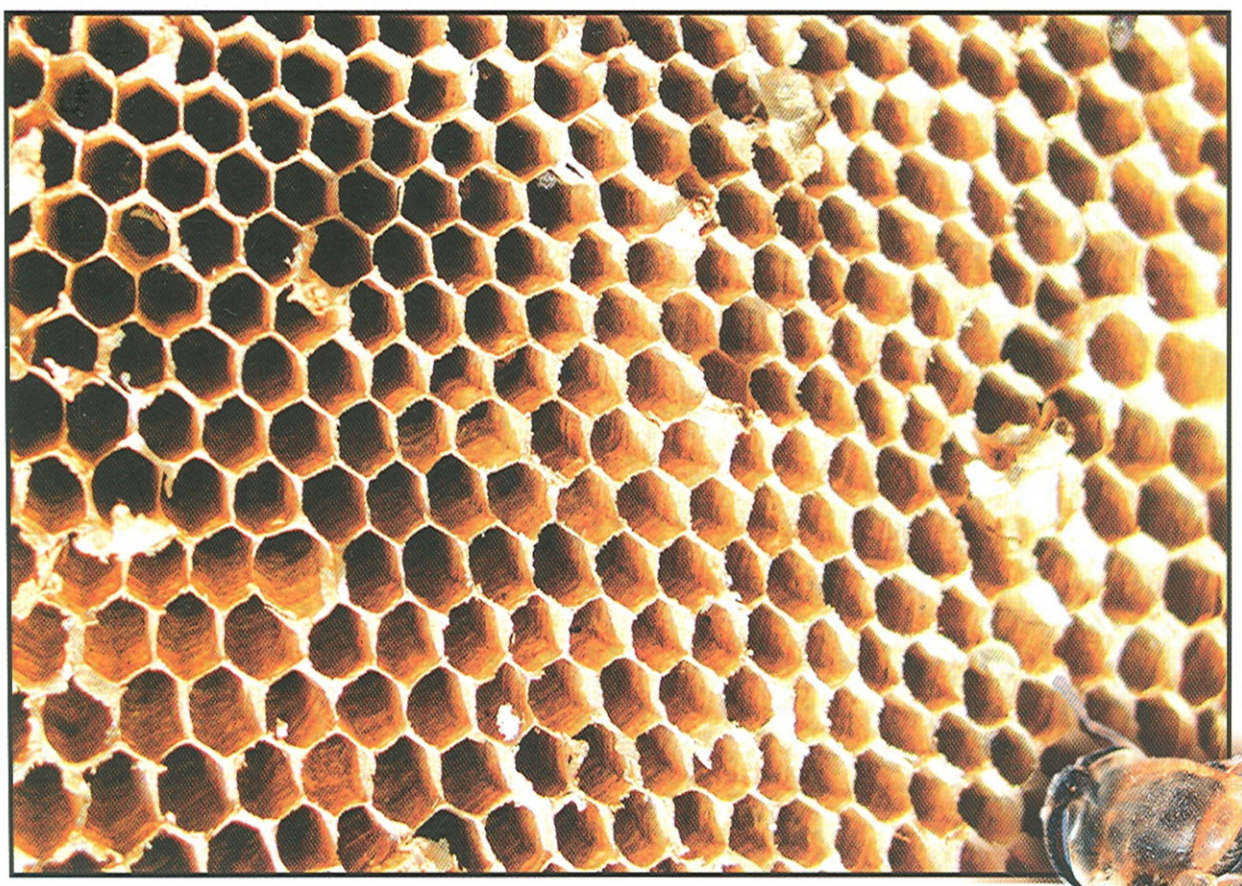
Programtervezési minták

Újrahasznosítható elemek objektumközpontú programokhoz



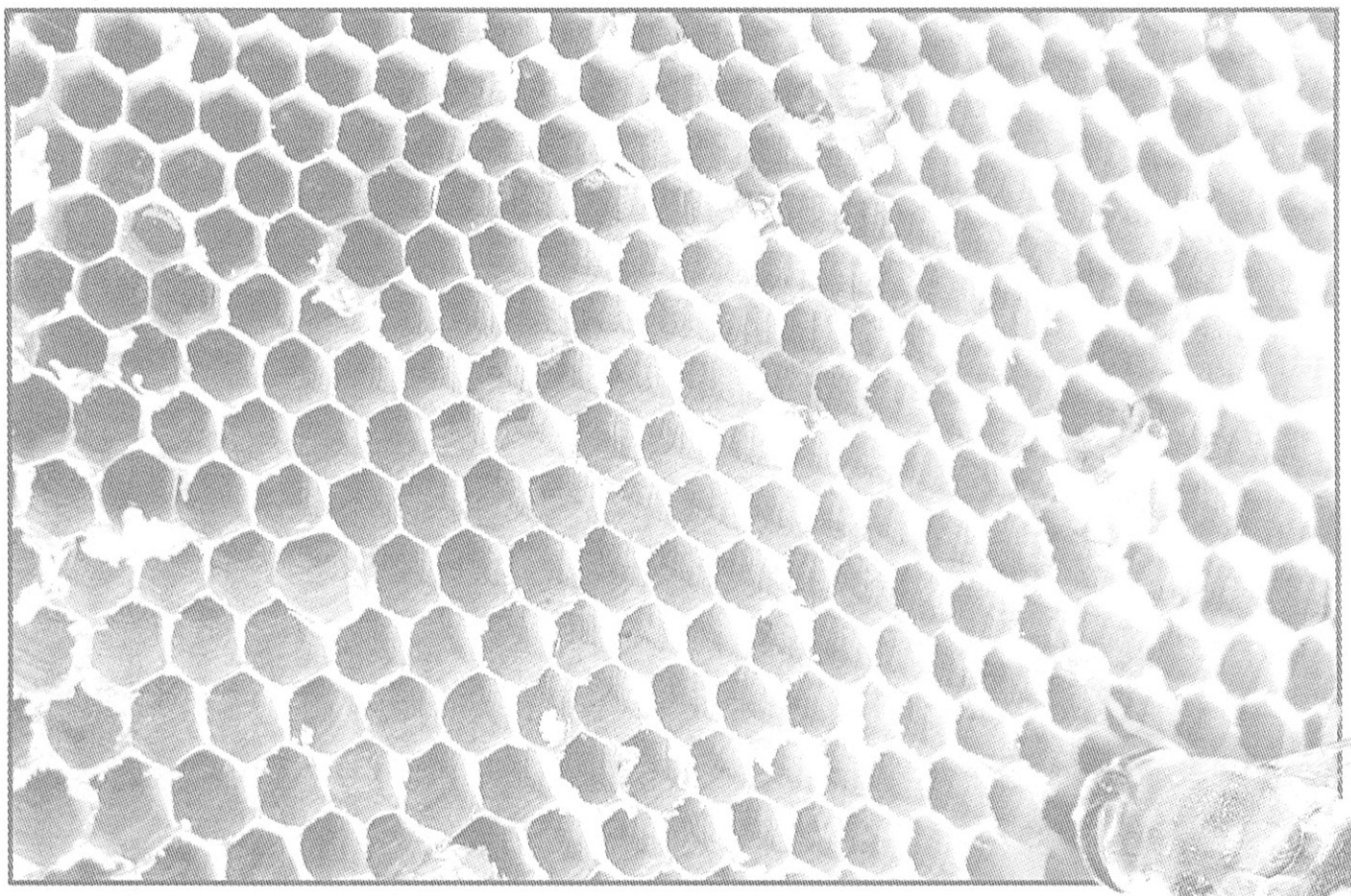
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

KISKAPU

Addison-Wesley

Programtervezési minták

Újrahasznosítható elemek objektumközpontú programokhoz



Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

KISKAPU

Addison-Wesley

A fordítás a következő angol eredeti alapján készült:

Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edition, ISBN 0201633612, Pearson Education Inc., Addison Wesley Professional

Nyelvi lektor: *Rézműves László*
Fordítás: *Antal Ferenc, Rézműves László*
Műszaki szerkesztő: *Csutak Hoffmann Levente*

Szakmai lektor: *Soczó Zsolt*
Borító: *Zsédely Teréz*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója
© 2004 Kiskapu Kft.
1081 Budapest, Népszínház u. 31. I. 7.
Telefon: (+36-1) 477-0443 Fax: (+36-1) 303-1619
http://kiado.kiskapu.hu/
e-mail: kiado@kiskapu.hu

ISBN: 963 9301 77 9

Készült a debreceni Kinizsi Nyomdában
Felelős vezető: *Bördős János*

Tartalomjegyzék

Karinnak
E.G.

Sylvie-nek
R.H.

Faith-nek
R.J.

Dru Ann-nek és Matthew-nak
Józsué 24:15b
J.V.

Ajánlások

„Jó ideje nem olvastam ilyen kitűnően megírt és mélyreható könyvet... A tervezési minták létjogosultságát a lehető legjobb módszerrel bizonyítja: nem a magyarázat, hanem a példák erejével."

Stan Lippman, C++ Report

„...Gamma, Helm, Johnson és Vlissides eme új könyvének hatása a szoftvertervezés elméletére bizonyosan hosszan tartó lesz. Kár, hogy a *Programtervezési minták* maga azt állítja, hogy csak az objektumközpontú programokkal foglalkozik, mert így az eme közösségen kívüli szoftverfejlesztők esetleg kézbe sem veszik – ami nagy kár lenne, mert a kötet mindenkinek nyújt valamit, aki szoftvert fejleszt. Minden fejlesztő használ tervezési mintákat, az újrahasznosítható elvont elemek jobb megértése pedig csak jobbá teheti munkánkat."

Tom DeMarco, IEEE Software

„Összességében úgy gondolom, a könyv hozzájárulása a szakterülethez egyedülálló és felbecsülhetetlen. Az objektumközpontú tervezés terén szerzett hatalmas tapasztalat áll mögötte, melyet tömör és újrahasznosítható formában tárnak elénk. Biztos, hogy gyakran fogom fellapozni, ha ötletekre lesz szükségem az objektumközpontú tervezést illetően – de hát éppen ez az újrahasznosítás lényege, nem?"

Sanjiv Gossain, Journal of Object-Oriented Programming

„Ez a régen várt könyv méltó az előzetes hírveréshez: mint egy építész tervei, úgy sorakoznak benne az idők során számtalanszor bizonyított, kipróbált módszerek. A szerzők az objektumközpontú tervezés terén szerzett több évtizedes tapasztalatukkal 23 mintát választottak ki, és önmagában ez a szám mutatja, milyen alapos és fegyelmezett munkát végeztek. A *Programtervezési minták* egy példányával minden programozónak rendelkeznie kell, aki szeretne még jobb lenni!"

Larry O'Brien, Software Development

„Kétségtelen tény, hogy a tervezési minták teljesen megváltoztathatják a szoftverfejlesztés módját, és a valódi, elegáns tervezés szintjére emelhetik. A *Programtervezési minták* az e területről írott könyvek közül messze kiemelkedik: olyan könyv, amelyet el kell olvasni, a mélyére kell ásni, meg kell tanulni kívülről – és szeretni kell. Mindörökre megváltoztatja a nézőpontunkat a programokat illetően."

Steve Billow, Journal of Object-Oriented Programming

„A *Programtervezési minták* igen erőteljes könyv. Rövid ismerkedés után a legtöbb C++-programozó képes lesz hasznosítani a benne bemutatott mintákat, hogy jobb programokat készítsen. A kötet intellektusunkat tökéletesen kielégíti: a gyakorlatban használható eszközöket nyújt, amelyek elgondolkodtatnak és segítenek, hogy hatékonyabban fejezhessük ki magunkat. Alapvetően megváltoztatja, amit a programozásról gondolunk."

Tom Cargill, C++ Report

Bevezetés

Ennek a könyvnek nem az a célja, hogy bevezetést adjon az objektumközpontú (objektumorientált) megoldásokhoz és az objektumközpontú programok fejlesztéséhez. A témáról számos igen jó könyvet írtak már. Kötetünk azt feltételezi, hogy az Olvasó jártas legalább egy objektumközpontú nyelvben, és rendelkezik valamennyi tapasztalattal az objektumközpontú tervezés terén is. Vagyis a leghatározottabban nem azoknak szól, akiknek a szótárért kell nyúlniuk, amikor típusokról vagy többalakúságról beszélünk, vagy ha szembeállítjuk a felületöröklést a megvalósítás-örökléssel.

Másrészről nem célunk az sem, hogy haladó szintű, technikai jellegű tárgyalását adjuk a témának. Ez csupán egy tervezési mintákat tartalmazó könyv, ami egyszerű és elegáns megoldásokat ír le egy-egy adott – az objektumközpontú szoftvertervezésen belüli – problémára. A tervezési minták olyan, hosszú évek alatt kidolgozott és továbbfejlesztett megoldásokat írnak le, amelyek az idők során már bizonyítottak, vagyis nem a „kezdeti ötletek" kategóriájába tartoznak. Többszöri újratervezés, újrakódolás áll mögöttük, amelynek célja a jobb újrahasznosíthatóság és a nagyobb rugalmasság volt. A tervezési minták ezeket a megoldásokat foglalják össze tömör, könnyen alkalmazható formában.

A tervezési minták sem „rejtett" nyelvi szolgáltatások, sem mások elkápráztatását szolgáló programozási trükkök ismeretét nem igénylik. Mindegyik megvalósítható szabványos objektumközpontú nyelveken, bár talán egy kicsivel több munkát igényelnek, mint az „alkalmi" megoldások. A befektetett erőfeszítések azonban megtérülnek, ahogy programjaink rugalmassága és újrahasznosíthatósága nő.

Ha egyszer megértjük a tervezési mintákat és nem csak kérdőn nézünk, amikor szóba kerülnek, soha többé nem fogunk az objektumközpontú tervezésre ugyanúgy gondolni. Ösztönösen olyan programokat fogunk készíteni, amelyek rugalmasabbak, modulárisabbak és átláthatóbbak – de hát éppen ezért választottuk az objektumközpontú programozást, nem igaz?

Figyelmeztetésképpen, de egyszersmind bátorításként fel kell hívnunk a figyelmet, hogy nem szabad aggódni, ha nem értünk meg mindent az első olvasásra. Mi sem értettünk mindent, amikor először papírra vetettük... Ez a kötet nem egyszeri elolvasásra és aztán a polcra való – sűrűn fogjuk kézbe venni és fellapozni, hogy ötleteket gyűjtsünk a tervezéshez.

A könyv nehezen született meg. Négy országot, három szerző házasságát, és két (rokonságban nem levő) utód születését látta, és rengeteg ember vette ki a részét a szerkesztéséből. Külön köszönet jár Bruce Andersonnak, Kent Becknek, és André Weinandnak biztatásukért és tanácsaikért. Azoknak is meg szeretnénk köszönni, akik átnézték a kézirat piszkozatát: Roger Bielefeld, Grady Booch, Tom Cargill, Marshall Cline, Ralph Hyre, Brian Kerninghan, Thomas Laliberty, Mark Lorenz, Arthur Riel, Doug Schmidt, Clovis Tondo, Steve Vinoski és Rebecca Wirsf-Brock – köszönjük. Hálásak vagyunk az Addison-Wesley kiadó csapatának (Kate Habib, Tiffany Moore, Lisa Raffaele, Pradeepa Siva és John Wait) is segítségükért, fáradhatatlanságukért és türelmükért, amellyel támogatták ezt a munkát. Külön köszönet Carl Kesslernek, Danny Sabbah-nak és Mark Wegmannak az IBM kutatóintézetéből.

Végül, de nem utolsósorban, mindenkinek köszönetet mondunk az Internetről, akik megjegyzéseket fűztek a minták különböző változataihoz, bátorító szavakat mondtak, vagy megerősítettek minket abban, hogy amit csinálunk, az értékes. Ezen emberek közé tartoznak (a teljesség igénye nélkül): Jon Avotins, Steve Berczuk, Julian Berdych, Matthias Bohlen, John Brant, Allan Clarke, Paul Chisholm, Jens Coldewey, Dave Collins, Jim Coplien, Don Dwiggins, Gabriele Elia, Doug Felt, Brian Foote, Denis Fortin, Ward Harold, Hermann Hueni, Nayeem Islam, Bikramjit Kalra, Paul Keefer, Thomas Kofler, Doug Lea, Dan LaLiberte, James Long, Anne Louise Luu, Pundi Madhavan, Brian Marick, Robert Martin, Dave McComb, Carl McConnell, Christine Mingins, Hanspeter Mössenböck, Eric Newton, Marianne Ozkan, Roxsan Payette, Larry Podmolik, George Radin, Sita Ramakrishnan, Russ Ramirez, Alexander Ran, Dirk Riehle, Bryan Rosenburg, Aamod Sane, Duri Schmidt, Robert Seidl, Xin Shu és Bill Walker.

Nem gondoljuk, hogy a minták gyűjteménye teljes és megváltoztathatatlan; inkább csak a tervezésről alkotott jelenlegi gondolataink tárháza. Szívesen fogadjuk a megjegyzéseket, legyen az bár a példák kritikája, olyan vonatkozások és ismert felhasználási módok, amelyeket kihagytunk, vagy olyan tervezési minták, amelyeket bele kellett volna foglalnunk a könyvbe. A leveleket a gondunkat viselő Addison-Wesley kiadó fogadja, de elektronikus levélben, a `design-patterns@cs.uiuc.edu` címen is elérhetők vagyunk. Ha a példakódokból elektronikus példányra lenne szükségünk, elég, ha elküldjük a „send design pattern source" üzenetet a `design-patterns@cs.uiuc.edu` címre. A legfrissebb hírek és javítások a `http://st-www-cs.uic.edu/users/patterns/DPBook/DPBook.html` weboldalon érhetők el.

E. G. Mountain View, California, USA
R. H. Montreal, Quebec, Kanada
R. J. Urbana, Illinois, USA
J. V. Hawthorne, New York, USA
1994. augusztus

Előszó

Minden jól szerkesztett objektumközpontú rendszer tele van mintákkal. Tulajdonképpen az egyik módja annak, ahogyan megítélem az objektumközpontú rendszerek minőségét az, hogy a tervezői vajon eléggé odafigyeltek-e az általános együttműködési lehetőségekre a nyelv objektumai között. Az ilyen szolgáltatásokra való összpontosítás a rendszertervezés folyamán olyan felépítményt eredményezhet, ami kisebb, egyszerűbb, és sokkal érthetőbb, mint ha ezen mintákat mellőzik.

Az összetett rendszerek esetében a minták fontosságát már régen felismerték. Christopher Alexander és munkatársai voltak talán az elsők, akik felvetették egy mintanyelv használatának ötletét épületek és városok építésénél. Gondolataik gyökeret vertek az objektumközpontú szoftverfejlesztők között is: röviden, a tervezési minták révén a programozás az építészek tudományának szintjére emelkedett.

Ebben a könyvben Erich Gamma, Richard Helm, Ralph Johnson és John Vlissides bemutatják a tervezési minták alapelveit, majd rendszerezik is a mintákat. Így a kötet két szempontból is nélkülözhetetlen: először is, megmutatja, milyen szerepet játszhatnak a minták az összetett rendszerek felépítésében, másodszor, a jól felépített minták bemutatása révén a gyakorlatban hasznosítható kézikönyvet nyújt, amit a gyakorló tervező munkája során felhasználhat.

Megtisztelve érzem magam, amiért lehetőségem volt tervezőként együtt dolgozni a könyv több szerzőjével is: sokat tanultam tőlük, és gyanítom, a kötetet elolvasva az Olvasó is fog.

Grady Booch
Vezető tudós, Rational Software Corporation

Útmutató az olvasónak

A kötet két fő részre oszlik. Az első rész (1. és 2. fejezet) leírja, mik is azok a tervezési minták és hogyan segítenek objektumközpontú programokat tervezni, majd egy esettanulmány segítségével bemutatja, hogyan alkalmazhatók a tervezési minták a gyakorlatban. A könyv második része (3., 4. és 5. fejezet) a tulajdonképpeni tervezési minták katalógusa.

E gyűjtemény teszi ki a könyv jelentős részét. A második rész fejezetei három típusra osztják a tervezési mintákat: létrehozási, szerkezeti és viselkedési mintákra. A katalógus többféle módon is használható. Elolvashatjuk az elejétől a végéig, de böngészhetünk mintáról mintára is. Más megközelítés, ha valamelyik fejezetet mélyrehatóan tanulmányozzuk. Ha ezt választjuk, látni fogjuk, hogy a közeli kapcsolatban álló minták hogyan különböztetik meg magukat egymástól.

A gyűjteményben való haladáshoz logikai útirányjelzőként használhatjuk a minták közötti hivatkozásokat. Ez a megközelítés bepillantást enged abba, hogyan kapcsolódnak a minták egymáshoz, hogyan kombinálhatók más mintákkal, illetve mely minták működnek jól együtt. Az 1.1 táblázat grafikusan mutatja be ezeket a hivatkozási kapcsolatokat.

Egy másik módja a katalógus elolvasásának a problémaközpontú megközelítés. Ugorjuk az 1.6. alfejezetre, ahol néhány általános problémáról olvashatunk az újrahasznosítható objektumközpontú programtervezés területéről, majd keressük meg és tanulmányozzuk az adott problémákra megoldást nyújtó mintákat. Van, aki először végigolvassa a katalógust, és utána tér át a problémaközpontú megközelítésre, hogy megkeresse a saját munkája során alkalmazható mintát.

Ha nincs túl nagy tapasztalatunk az objektumközpontú tervezésben, kezdjük a legegyszerűbb és legáltalánosabb mintákkal:

- Elvont gyár (Abstract Factory)
- Illesztő (Adapter)
- Összetétel (Composite)
- Díszítő (Decorator)
- Gyártófüggvény (Factory Method)
- Megfigyelő (Observer)
- Stratégia (Strategy)
- Sablonfüggvény (Template Method)

Olyan objektumközpontú rendszer szinte nincs is, amelyik nem használ legalább egy párat az itt felsorolt minták közül, a nagy rendszerek pedig csaknem mindet használják. E leggyakrabban alkalmazott minták segítenek, hogy megértsük a jó objektumközpontú tervezés lényegét, illetve az egyes tervezési minták sajátosságait.

1

Bevezetés

Objektumközpontú programot tervezni nehéz, *újrahasznosíthatót* még nehezebb. Meg kell találni a megfelelő objektumokat, helyesen kell osztályokba rendezni azokat, meg kell határozni az osztályok felületeit és az öröklési viszonyokat, és létre kell hozni a kulcsfontosságú kapcsolatokat közöttük. A tervnek igazodnia kell az adott problémához, de eléggé általánosnak kell lennie ahhoz, hogy később más feladatokhoz és követelményekhez is illeszthető legyen. Általában arra is törekszünk, hogy ne legyen szükség újratervezésre, vagy legalábbis a lehető legkisebb mértékben. Bármelyik tapasztalt tervező megmondhatja, hogy egy újrahasznosítható és rugalmas objektumközpontú szoftvert nagyon nehéz, ha nem lehetetlen elsőre pontosan megtervezni. Mielőtt egy terv kész lenne, általában többször megpróbálják újrafelhasználni, mindannyiszor módosítva rajta valamit.

Mégis, a tapasztalt tervezők igenis készítenek jó terveket, míg a kezdő tervezők elvesznek a lehetőségek dzsungelében, és hajlamosak visszatérni a korábban megismert, nem objektumközpontú megoldásokhoz. Hosszú időbe telik, amíg a kezdők megtanulják, miről is szól a jó objektumközpontú tervezés. Vagyis a tapasztalt tervezők nyilván tudnak valamit, amit a tapasztalatlanok nem. De mi lehet az?

A szakértő tervezők például tudják, hogy *nem* szabad minden problémára új megoldásokat kidolgozni. Célszerűbb újrahasznosítani azon megoldásokat, amelyek korábban már működőképesnek bizonyultak. Vagyis ha találunk egy jó megoldást, újra és újra felhasználhatjuk – ez is hozzájárul ahhoz, hogy szakértőkké válhassunk. Ebből következően a különböző objektumközpontú rendszerekben visszatérő osztálymintákat és együttműködő objektumokat találhatunk: ezek a minták adott tervezési problémákra adnak megoldást és rugalmassá, elegánssá, végső soron újrahasznosíthatóvá teszik az objektumközpontú rendszereket. Segítenek a tervezőknek újrahasznosítani a sikeres terveket, azzal, hogy az új terveket a megelőző tapasztalat alapjaira helyezik. Az a tervező, akinek ismerősek az ilyen minták, rögtön képes lesz alkalmazni azokat más feladatokra is, anélkül, hogy újra fel kellene fedeznie őket.

A lényeget egy hasonlattal ragadhatjuk meg. Regény- és színdarabírók ritkán rögtönzik a cselekményt; ehelyett olyan mintákat követnek, mint a „tragikusan esendő hős" (Macbeth, Hamlet stb.) vagy „a romantikus regény" (erre számtalan példa akad). Hasonló módon, az objektumközpontú programokat készítő tervezők olyan mintákat követnek, mint az „ábrázoljunk állapotokat objektumokkal", „készítsünk „díszíthető" objektumokat, amelyekhez könnyen új jellemzőket adhatunk, vagy éppen elvehetünk belőlük". Ha ismerjük a mintát, számos tervezési döntés automatikusan adódik.

Mindannyian ismerjük a tervezési tapasztalat értékét. Hányszor volt tervezés közben *déja vu* érzésünk – hogy egyszer már megoldottuk a problémát ezelőtt, de nem tudjuk, mikor és hogyan? Ha emlékeznénk az előző probléma részleteire és arra, hogyan oldottuk meg, újrahasznosíthatnánk az akkori megoldást, ahelyett, hogy újra rá kellene jönnünk. Sajnos azonban tervezési tapasztalatainkat ritkán rögzítjük mások számára.

E könyv célja, hogy rögzítse az objektumközpontú tervezési tapasztalatokat, **tervezési mintákként**. Mindegyik tervezési minta rendszerez, elnevez, megmagyaráz és értékel egy fontos, az objektumközpontú rendszereken belül gyakran ismétlődő megoldást. Célunk az, hogy mások számára hatékonyan alkalmazható formában adjuk át tervezési tapasztalatainkat, ezért összegyűjtöttünk és leírtunk néhányat a legfontosabb tervezési minták közül – ezeket mutatjuk be a kötetben.

A tervezési minták könnyebbé teszik a sikeres tervek és szerkezetek újbóli felhasználását. A már bizonyított módszerek tervezési mintákként való rögzítése elérhetőbbé teszi e megoldásokat az új rendszerek tervezői számára. A tervezési minták segítenek kiválasztani azt a megoldást, amelyik újrahasznosíthatóvá teszi a rendszert, és segítenek elkerülni azokat, amelyek nem. A tervezési minták a létező rendszerek dokumentálását és karbantartását is segíthetik, azzal, hogy pontosan leírják a mögöttük megbúvó osztályokat és objektumkapcsolatokat. Hogy egyszerűen fogalmazzunk, a tervezési minták segítenek a tervezőnek, hogy hamarabb leljen rá a helyes útra.

Az e könyvben szereplő tervezési minták egyike sem ír le új vagy nem bizonyított terveket. Csak olyanokat vettünk bele, amelyeket már többször is alkalmaztak különböző rendszerekben. E tervek legtöbbjét még sohasem dokumentálták: az objektumközpontú közösség „folklórjának" részei csupán, vagy elemei egy sikeres objektumközpontú rendszernek, de egy kezdőnek ezek egyikéből sem egyszerű tanulni. Tehát, bár ezek a tervek nem újak, új és elérhető formában, egyfajta tervezésiminta-katalógusként tárjuk az olvasók elé.

A könyv mérete ellenére csak a töredékét képes bemutatni egy szakember valószínűsíthető tudásának. Nincs benne egyetlen minta sem, ami az egyidejű (párhuzamos vagy elosztott) vagy a valósidejű programozással foglalkozik, és nincsenek benne kifejezetten egy adott alkalmazástípusra vonatkozó minták. Nem mondja meg, hogyan készítsünk felhasználói felületeket, hogyan írjunk meghajtóprogramokat, vagy hogyan használjunk objektumközpontú adatbázisokat. Természetesen ezen területek mindegyikének is megvannak a saját mintái, és érdemes lenne valakinek ezeket is katalogizálni.

1.1 Mi is az a tervezési minta?

Christopher Alexander azt mondja: „Minden minta olyan problémát ír le, ami újra és újra felbukkan a környezetünkben, s aztán leírja hozzá a megoldás magját, oly módon, hogy a megoldás milliószor felhasználható legyen, anélkül, hogy valaha is kétszer ugyanúgy csinálnánk." [AIS+77, *x*. oldal] Bár Alexander épületek és városok mintáiról beszélt, amit mond, az igaz az objektumközpontú tervezési mintákra is. A mi megoldásaink falak és ajtók helyett objektumok és felületek szintjén fejeződnek ki, de alapjában véve mind a kettő megoldás egy adott problémára az összefüggéseiben nézve.

A mintáknak általában négy lényeges elemük van:

1. A **minta neve** hivatkozási eszköz, amit arra használhatunk, hogy egy-két szóval leírjunk egy tervezési problémát, megoldásait és a következményeket. A minta elnevezése azonnal növeli tervezési szókincsünket és lehetővé teszi, hogy magasabb elvonatkoztatási szinten tervezzünk. Az, hogy vannak szavaink a mintákra, lehetővé teszi, hogy beszéljük róluk a kollégáinkkal (vagy akár „magunkkal"), és használjuk őket a dokumentációban. Könnyebbé teszi, hogy gondolkodjunk a tervezésről, hogy beszéljünk a tervekről és használatuk pozitív és negatív oldalairól. A jó nevek megtalálása gyűjteményünk összeállításának egyik legnehezebb része volt.
2. A **probléma** írja le, mikor alkalmazzuk a mintát, vagyis elmagyarázza a problémát és összefüggéseit. Leírhat konkrét tervezési problémákat is, mint például „hogyan ábrázoljunk algoritmusokat objektumokként", de olyan osztály- vagy objektumszerkezeteket is, amelyek rugalmatlan tervezésre utalnak. A probléma néha feltételek listáját is tartalmazhatja, amelyeknek teljesülniük kell, mielőtt értelme lenne alkalmazni a mintát.
3. A **megoldás** azokat az elemeket írja le viszonyaikkal, hatáskörükkel és együttműködési lehetőségeikkel együtt, amelyek felépítik a tervet. Nem konkrét tervet vagy megvalósítást ad meg, mivel a minta olyan, mint egy sablon, ami különféle helyzetekben alkalmazható. Helyette csupán egy tervezési probléma elvont leírását biztosítja, és hogy az elemek (esetünkben osztályok és objektumok) általános elrendezése hogyan oldja azt meg.
4. A **következmények** a minta alkalmazásának előnyei, illetve hátrányai. Bár a következményekről ritkán esik szó a tervezési döntések indoklásánál, mégis létfontosságúak a tervezési alternatívák értékelésénél és annak megértésében, hogy mi a haszna és mi a hátránya a minta alkalmazásának. A „következmények" gyakran a tárhely- és idő-felhasználást érintő hátrányokra vonatkoznak, de érinthetnek nyelvi és megvalósítási kérdéseket is. Miután az újrahasznosíthatóság gyakran fontos tényező az objektumközpontú tervezésnél, a következmények hasznos információkat szolgáltatnak a rendszer rugalmasságáról, bővíthetőségéről vagy hordozhatóságáról. A következmények felsorolása határozottan segít megérteni és értékelni az előnyöket és hátrányokat.

Nézőpontunk határozza meg, mit tekintünk mintának és mit nem. Ami az egyiknek minta, a másiknak csupán elemi építőkocka; a kötetben ezért igyekeztünk a mintákra az elvonatkoztatás egy bizonyos szintjén összpontosítani. A *Programtervezési minták* nem az olyan

szerkezetekről szól, mint a láncolt listák vagy a hasítótáblák (kivonatok), amelyeket osztályokként kódolhatunk és így újra felhasználhatunk, de nem is összetett, teljes alkalmazások vagy alrendszerek építéséhez használható tervezetekről. A könyvben bemutatott tervezési minták *egymással együttműködő objektumok és osztályok leírásai, amelyek testreszabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben.*

A tervezési minták megnevezik, elvonatkoztatják és azonosítják egy-egy mindennapos szerkezet kulcsfontosságú tulajdonságait, amelyek lehetővé teszik, hogy újrahasznosítható objektumközpontú tervet állítsunk elő. A tervezési minta azonosítja a részt vevő osztályokat és példányokat, szerepüket és kapcsolataikat, illetve a felelősségi körök eloszlását. Mindegyik tervezési minta egy bizonyos objektumközpontú tervezési kérdésre összpontosít. Leírja, mikor alkalmazható, más tervezési megszorításokat figyelembe véve lehet-e alkalmazni, továbbá ismerteti használatának következményeit és az előnyöket, illetve hátrányokat. Mivel a terveket előbb-utóbb meg is kell valósítanunk, a minták mellett példákat is nyújtunk C++ és (időnként) Smalltalk nyelven, hogy illusztráljuk a megvalósítás módját.

Bár a tervezési minták objektumközpontú szerkezeteket írnak le, gyakorlati megoldásokon alapulnak, amelyeket az olyan általános objektumközpontú programozási nyelvek, mint a Smalltalk és a C++ beépítve tartalmaznak, szemben az eljárásközpontú (procedurális) nyelvekkel (Pascal, C, Ada) vagy a dinamikusabb objektumközpontú nyelvekkel (CLOS, Dylan, Self). Gyakorlati okok miatt választottuk a Smalltalkot és a C++-t: népszerűségük állandó, és mindennapos tapasztalataink vannak velük.

A programnyelv választása fontos, mivel befolyásolja a programozó nézőpontját. A mi mintáink Smalltalk, illetve C++ szintű nyelvi tulajdonságokat feltételeznek, és ez a választás meghatározza, mit lehet és mit nem lehet könnyen megvalósítani. Ha eljárásközpontú nyelveket feltételeztünk volna, a kötetbe a következő nevű mintákat is bele kellett volna foglalnunk: „Öröklés", „Betokozás (egységbe zárás)" „Többalakúság" stb. Néhány mintánkat viszont közvetlenül támogatja egy-egy kevésbé mindennapi objektumközpontú programozási nyelv. A CLOS-nak például vannak úgynevezett multi-metódusai, amelyek csökkentik az igényt az olyan mintákra, mint a Látogató (Visitor). Persze számos különbség van a Smalltalk és a C++ között is, ami annyit tesz, hogy néhány minta könnyebben kifejezhető az egyik nyelven, mint a másikon: ilyen például a Bejáró (Iterator).

1.2 Tervezési minták a Smalltalk MVC-ben

A Model-View-Controller (MVC) osztályhármas [KP88] arra használatos, hogy felhasználói felületeket építsünk fel a Smalltalk-80-ban. Az MVC-n belüli tervezési mintákra pillantva jobban megérthetjük, mit értünk a „minta" kifejezés alatt.

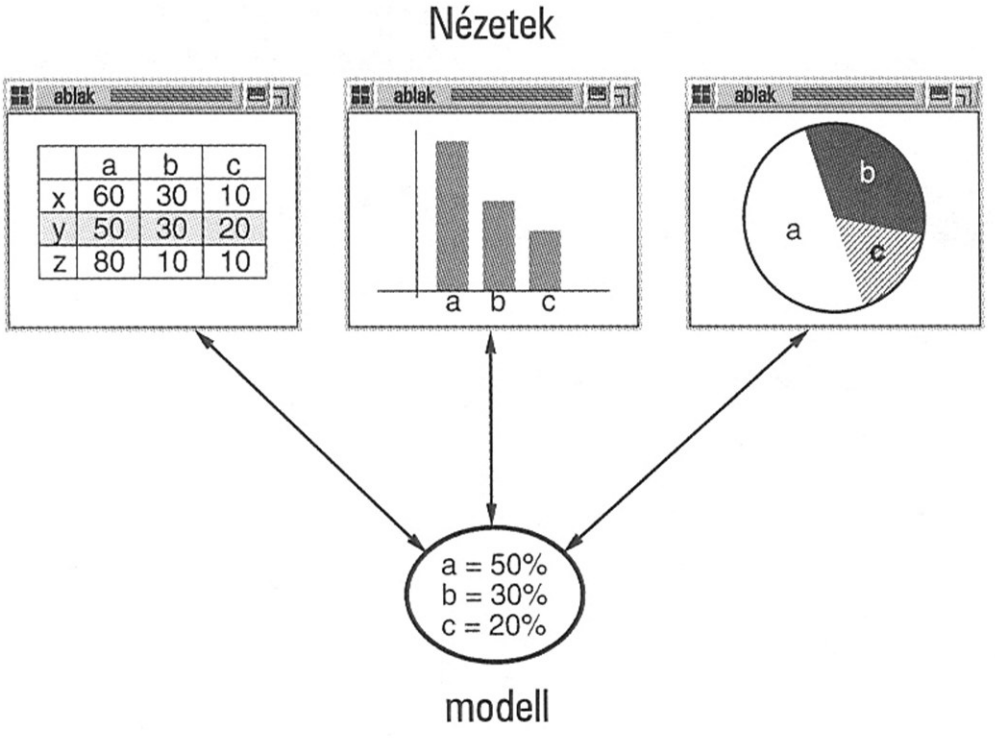
Az MVC háromfajta objektumot tartalmaz. A Model (Modell) az alkalmazás objektum, a View (Nézet) annak ábrázolása a képernyőn, a Controller (Vezérlő) pedig annak módját határozza

meg, ahogyan a felhasználói felület válaszol a felhasználói bemenetre. Az MVC előtt a felhasználói felületek tervezésekor hajlamosak voltak ezeket az objektumokat egymásba olvasztani. Az MVC szétválasztja őket, hogy növelje a rugalmasságot és az újrahasznosíthatóságot.

Az MVC úgy választja szét a nézeteket és modelleket, hogy egy előfizetés–értesítés protokollt hoz létre közöttük. A nézetnek gondoskodnia kell arról, hogy megjelenése tükrözze a modell állapotát. Valahányszor a modell adatai megváltoznak, a modell értesíti azokat a nézeteket, amelyek rá támaszkodnak, azok pedig lehetőséget kapnak rá, hogy frissítsék magukat. Ez a megközelítés lehetővé teszi, hogy többféle nézetet kapcsoljunk egy modellhez, hogy különböző megjelenítéseket biztosítsunk, emellett pedig anélkül készíthetünk új nézeteket a modellhez, hogy újraírnánk azt.

A következő diagram egy modellt és három nézetet mutat. (Az egyszerűség kedvéért kihagytuk a vezérlőket.) A modell néhány adatértéket tartalmaz, a nézetek pedig egy táblázatot, egy hisztogramot és egy tortadiagramot határoznak meg, amelyek ezeket az adatokat jelenítik meg különféle módokon.

A modell értesíti a nézeteket, amikor az értékei megváltoznak, a nézetek pedig kapcsolatot tartanak a modellel, hogy elérjék ezeket az értékeket.



Felszínesen értékelve, ez a példa egy olyan szerkezetet tükröz, ami elválasztja a nézeteket és a modellt, de alkalmazható egy jóval általánosabb problémára is: objektumok elválasztására. Így az egy objektumot érintő változások több másikra kihathatnak, anélkül, hogy a megváltozott objektum ismerné a többiek belső szerkezetét. Ezt az általánosabb szerkezetet írja le a Megfigyelő (Observer) tervezési minta.

Egy másik tulajdonsága az MVC-nek, hogy a nézetek beágyazhatók. Például egy gombokból álló vezérlőpult megvalósítható összetett nézetként, ami beágyazott gombnézeteket tartalmaz. Egy objektumvizsgáló felhasználói felülete szintén állhat beágyazott nézetekből, amelyeket újra felhasználhatunk egy hibakeresőben. Az MVC a CompositeView (Összetett nézet) osztállyal támogatja a beágyazott nézeteket, amely a View alosztálya. A CompositeView objektumok úgy viselkednek, mint a View objektumok; egy összetett nézet bárhol használható, ahol egy sima nézet (view), de ezen felül tartalmazza és kezeli a beágyazott nézeteket.

Mindezt tekinthetjük úgy, mint egy olyan szerkezetet, ami lehetővé teszi számunkra, hogy az összetett nézetek egészét ugyanúgy kezeljük, mint az egyes elemeiket, de ismét csak azt mondhatjuk, hogy a terv egy ennél jóval általánosabb problémára is alkalmazható, ami mindig felbukkan, valahányszor csoportba akarunk foglalni objektumokat és úgy kezelni ezen csoportot, mint egy egységes objektumot. Ezt az általánosabb szerkezetet a Összetétel (Composite) tervezési minta írja le. Segítségével osztályhierarchiát hozhatunk létre, amelyben egyes alosztályok elemi objektumokat írnak le (pl. Gomb), más osztályok pedig összetett objektumokat (CompositeView), amelyek az elemi objektumokból állnak össze.

Az MVC azt is lehetővé teszi, hogy megváltoztassuk azt, ahogyan a nézet reagál a felhasználói bemenetre, anélkül, hogy módosítanánk a megjelenését. Például lehet, hogy változtatni szeretnénk azon, ahogyan a billentyűleütésekre reagál, vagy helyi előugró menüt szeretnénk használni billentyűparancsok helyett. Az MVC a válaszrendszert egy vezérlő (Controller) objektumba zárja; e vezérlők pedig osztályhierarchiát alkotnak, megkönnyítve az új vezérlők készítését a már létezők alapján.

A nézetek valamelyik Controller alosztály egy példányát használják, hogy megvalósítsák a kívánt válaszstratégiát; eltérő reagálás megvalósításához egyszerűen lecseréljük a példányt egy másfajta vezérlőre. Az is lehetséges, hogy futásidőben változtassuk meg a nézet vezérlőjét, így lehetővé tehetjük a nézet számára, hogy a felhasználói bemenetre másképpen válaszoljon. Egy nézet például letiltható egyszerűen úgy, hogy olyan vezérlőhöz rendeljük, amely nem veszi figyelembe a bemeneti eseményeket, így a nézet nem fogadja el a bemeneteket.

A nézet–vezérlő viszony a Stratégia (Strategy) tervezési minta példája. A stratégia egy objektum, ami egy algoritmust képvisel. Akkor vesszük hasznát, amikor statikusan vagy dinamikusan ki akarjuk cserélni az algoritmust, amikor az algoritmusnak számos változatával rendelkezünk, vagy amikor az algoritmusnak összetett adatszerkezetei vannak, amelyeket egységbe szeretnénk zárni.

Az MVC más tervezési mintákat is használ, például a Gyártófüggvényt (Factory Method), hogy megadja az alapértelmezett vezérlőosztályt egy nézet számára, vagy a Díszítő (Decorator) mintát, hogy gördítősávot adjon egy nézethez, de a fő összefüggéseket az MVC-ben a Megfigyelő, az Összetétel és a Stratégia tervezési minták adják meg.

1.3 A tervezési minták leírása

Hogyan írunk le tervezési mintákat? A grafikus jelölésmódok, habár fontosak és hasznosak, nem elegendőek, hiszen csak a tervezési folyamat végterméket mutatják, az osztályok és objektumok viszonyaként. Ahhoz, hogy újrahasznosíthassuk a tervet, muszáj rögzítenünk a döntéseket, az alternatívákat és döntéseink következményeit is. A konkrét példák is fontosak, mivel segítenek abban, hogy a tervet működés közben lássuk.

A könyvben a tervezési mintákat egységes formában írtuk le. Mindegyik mintát szakaszokra osztottuk, amelyek a következő sablonhoz igazodnak. A sablon egységes szerkezetet kölcsönöz az információknak, egyszerűbbé téve a tervezési minták megtanulását, összehasonlítását és használatát.

A minta neve és besorolása

A minta neve a minta lényegét közvetíti rövid formában. A jó név létfontosságú, mivel a tervezési szókincs részévé válik. A minta besorolásának alapját az 1.5 részben található vázlat adja meg.

Cél

Rövid leírás, amely a következő kérdésekkel foglalkozik: Mit csinál az adott tervezési minta? Mi az értelme és a célja? Milyen sajátos tervezési problémára ad választ?

Egyéb nevek

A minta más, jól ismert nevei, ha vannak ilyenek, illetve a minta angol neve.

Feladat

Forgatókönyv, amely bemutatja a tervezési problémát és azt, hogy az osztályok és az objektumok hogyan oldják azt meg. A forgatókönyv segít a minta következő, elvontabb leírásának megértésében.

Alkalmazhatóság

Melyek azok a helyzetek, ahol az adott tervezési minta alkalmazható? Melyek azok a rossz tervek, amelyek leváltását a minta megcélozza? Hogyan ismerhetők fel ezek?

Szerkezet

A minta osztályainak grafikus szemléltetése, az OMT-n (Object Modeling Technique [RPB+91]) alapuló jelölést használva. Együttműködési diagramokat („interakció-diagramokat") is használunk [JCJO92, Boo94], hogy szemléltessük a kérelmek és együttműködések sorozatát az objektumok között. A jelölésrendszert részletesen a B függelék ismerteti.

Résztvevők

Az osztályok, illetve objektumok, amelyek részt vesznek a tervezési mintákban, valamint azok feladatai.

Együttműködés

Hogyan működnek együtt az objektumok, hogy végrehajtsák feladataikat?

Következmények

Hogyan támogatja az adott minta a kívánt célokat? Mik a minta használatának előnyei és hátrányai? A rendszer mely szerkezeti elemeit változtathatjuk szabadon az adott mintát használva?

Megvalósítás
Milyen buktatókra kell ügyelni, milyen módszereket érdemes használni a minta megvalósításakor? Vannak nyelvi sajátosságok?

Példakód
Programkód-töredékek, amelyek illusztrálják, hogyan valósítható meg a minta C++ vagy Smalltalk nyelven.

Ismert felhasználások
Valós rendszerekből vett példák a mintára. Legalább két példát szerepeltetünk, különböző területekről.

Kapcsolódó minták
Mely tervezési minták kapcsolódnak szorosan az adott mintához? Mik a legfontosabb különbségek? Milyen más mintákkal együtt célszerű használni az adott mintát?

A függelékekben kiegészítő információkat találhatunk, amelyek segítenek megérteni a mintákat és a hozzájuk kapcsolódó magyarázó leírásokat. Az A függelék az általunk használt szakkifejezések jegyzéke. Már említettük a B függeléket, ami a különféle jelöléseket mutatja be. Végezetül, a C függelék azon alaposztályok forráskódját tartalmazza, melyeket a példakódokban használtunk.

1.4 A tervezési minták katalógusa

Gyűjteményünk 23 tervezési mintát tartalmaz. Ezek neve és célja áttekintésképpen a következő listában szerepel.

Elvont gyár (Abstract Factory) Kapcsolódó vagy egymástól függő objektumok családjának létrehozására szolgáló felületet biztosít a konkrét osztályok megadása nélkül.

Illesztő (Adapter) Az adott osztály felületét az ügyfelek által igényelt felületté alakítja. E módszerrel az egyébként összeférhetetlen felületű osztályok együttműködését biztosíthatjuk.

Híd (Bridge) Az elvont ábrázolást elválasztja a megvalósítástól, hogy a kettő egymástól függetlenül módosítható legyen.

Építő (Builder) Az összetett objektumok felépítését függetleníti az ábrázolásuktól, így ugyanazzal az építési folyamattal különböző ábrázolásokat hozhatunk létre.

Felelősséglánc (Chain of Responsibility) Arra szolgál, hogy elkerüljük a kérelem küldőjének a fogadóhoz való kötését. Ezt úgy érjük el, hogy több objektumnak is jogot adunk a kérelem kezelésére. A fogadó objektumokat láncba állítjuk, amelyen a kérelem addig halad, amíg el nem ér egy objektumot, ami képes a kezelésére.

Parancs (Command) A kérelmeket objektumba zárja, aminek célja, hogy az ügyfeleknek paraméterként különböző kérelmeket adjunk át, ezeket sorba állítsuk vagy naplózzuk, illetve támogassuk a műveletek visszavonását.

Összetétel (Composite) Az objektumokat faszerkezetbe rendezi, hogy ábrázolhassuk a rész–egész viszonyokat. A módszer révén az önálló objektumokat és az objektumösszetételeket egységesen kezelhetjük.

Díszítő (Decorator) Az objektumokhoz dinamikusan további felelősségi köröket rendel. A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.

Homlokzat (Facade) Egy alrendszerben felületek egy halmazához egységes felületet biztosít. A módszerrel magasabb szintű felületet határozunk meg, amelynek révén az adott alrendszer könnyebben használhatóvá válik.

Gyártófüggvény (Factory Method) Felületet határoz meg egy objektum létrehozásához, de az alosztályokra bízza, melyik osztályt példányosítják. A gyártófüggvények megengedik az osztályoknak, hogy a példányosítást az alosztályokra ruházzák át.

Pehelysúlyú (Flyweight) Megosztás révén támogatja a nagy finomságú objektumok tömegeinek hatékony felhasználását.

Értelmező (Interpreter) Egy adott nyelv nyelvtanát ábrázolja, illetve ehhez az ábrázoláshoz értelmezőt biztosít, amely annak alapján képes az adott nyelv mondatait megérteni.

Bejáró (Iterator) Az összetett objektumok elemeinek soros elérését a háttérben megbúvó ábrázolás felfedése nélkül biztosító módszer.

Közvetítő (Mediator) Egy objektumot határoz meg, amely objektumok egy halmazának együttműködését irányítja. (Vagyis ezeket egyetlen objektumba tokozzuk be.) A módszerrel laza csatolást hozunk létre, amelyben az egyes objektumok közvetlenül nem hivatkozhatnak egymásra, a köztük levő kapcsolatok pedig egymástól függetlenül módosíthatók.

Emlékeztető (Memento) Az egységbe zárás (betokozás, enkapszuláció) megsértése nélkül kinyeri és rögzíti egy objektum belső állapotát, hogy az később ebbe az állapotba visszaállítható legyen.

Megfigyelő (Observer) Objektumok között egy–sok függőségi kapcsolatot hoz létre, így amikor az egyik objektum állapota megváltozik, minden tőle függő objektum értesül erről és automatikusan frissül.

Prototípus (Prototype) Prototípus példány használatával meghatározza, milyen típusú objektumokat kell létrehozni, az új objektumok létrehozását pedig ennek a prototípusnak a lemásolásával állítja elő.

Helyettes (Proxy) Adott objektumot képviselőn vagy helyőrzőn keresztül irányítunk, hogy szorosabban felügyelhessük működését.

Egyke (Singleton) Egy osztályból csak egy példányt engedélyez, és ehhez globális hozzáférési pontot ad meg.

Állapot (State) Adott objektum számára engedélyezi, hogy belső állapotának megváltozásával megváltoztathassa viselkedését is. Az objektum ekkor látszólag módosítja az osztályát.

Stratégia (Strategy) Algoritmus-családot határoz meg, melyben az algoritmusokat egyenként egységbe zárjuk és egymással felcserélhetővé tesszük. E módszer révén az algoritmus a felhasználó ügyféltől függetlenül módosítható.

Sablonfüggvény (Template Method) Adott művelet algoritmusának vázát készíti el, amelynek egyes lépéseit alosztályokra ruházza át. Így az alosztályok az algoritmus egyes lépéseit felülbírálhatják, anélkül, hogy az algoritmus szerkezete módosulna.

Látogató (Visitor) Objektumszerkezet elemein végrehajtandó műveletet ábrázol: a Látogató minta segítségével anélkül határozhatunk meg egy új műveletet, hogy a benne részt vevő elemek osztályát meg kellene változtatnunk.

1.5 A katalógus rendszerezése

A tervezési minták részletességükben és elvonatkoztatási szintjük szerint különbözőek. Mivel sok tervezési minta van, szükséges őket valamilyen úton-módon rendszerezni. Ebben a részben osztályokba soroljuk a tervezési mintákat, így hivatkozhatunk a rokon minták családjaira. Az osztályba sorolás segíti a katalógusban levő minták gyorsabb megtanulását és új minták felfedezésére is ösztönözhet.

		Cél		
		Létrehozási	**Szerkezeti**	**Viselkedési**
Hatókör	**Osztály**	Gyártófüggvény	(Osztály)illesztő	Értelmező Sablonfüggvény
	Objektum	Elvont gyár Építő Prototípus Egyke	(Objektum)illesztő Híd Összetétel Díszítő Homlokzat Pehelysúlyú Helyettes	Felelősséglánc Parancs Bejáró Közvetítő Emlékeztető Megfigyelő Állapot Stratégia Látogató

1.1 táblázat
Tervezési minták hatókör szerint.

A tervezési mintákat két szempont szerint osztályozhatjuk (1.1 táblázat). Az első a **cél**, ami azt tükrözi, mit csinál a minta. A mintáknak **létrehozási**, **szerkezeti** vagy **viselkedési** célja lehet. A létrehozási minták (vagy alkotó minták) az objektum-létrehozás folyamatában érdekeltek. A szerkezeti minták az objektumok és osztályok felépítésével foglalkoznak. A viselkedési minták azon tulajdonságokat írják le, ahogyan az objektumok kölcsönösen együttműködnek és felosztják a felelősségi köröket.

A második szempont a **hatókör**, ami meghatározza, hogy a minta objektumra vagy osztályra alkalmazható-e. Az osztályminták az osztályok és az alosztályok viszonyával foglalkoznak. Ezen viszonyok örökléssel létesítődnek, ezért statikusak, vagyis fordításkor rögzítettek. Az objektumminták objektumkapcsolatokra vonatkoznak, amelyek futásidőben megváltoztathatók, ezért jóval dinamikusabbak. Majdnem minden minta használ valamilyen szintű öröklést, így csak azokat a mintákat hívjuk „osztálymintáknak", amelyek az osztályok közötti viszonyokra összpontosítanak. A legtöbb minta az „objektum" hatókörbe tartozik.

A létrehozási osztályminták az objektum-létrehozás feladatát részben az alosztályokra ruházzák át, míg a létrehozási objektumminták egy másik objektumra hagyják. A szerkezeti osztályminták öröklést használnak az osztályok előállításához, míg a szerkezeti objektumminták objektumépítési módokat írnak le. A viselkedési osztályminták öröklést használnak az algoritmusok és vezérlési folyamatok leírásához, míg a viselkedési objektumminták azt írják le, hogyan dolgozik együtt objektumok egy csoportja, hogy végrehajtsanak egy olyan feladatot, amelyet egyetlen objektum nem tudna önmagában végrehajtani.

A minták csoportosításának két másik módja lehet. Néhány mintát gyakran együtt használunk: ilyen például az Összetétel, amelyet gyakran használunk együtt a Bejáró és a Látogató mintákkal. Néhány minta vagylagos: a Prototípus mintát például gyakran az Elvont gyár helyett használhatjuk. Néhány minta hasonló felépítést eredményez, holott más-más célt szolgálnak: például az Összetétel és a Díszítő minták szerkezeti diagramjai egyformák.

A tervezési minták rendszerezésének másik módja, amikor aszerint csoportosítjuk őket, hogy a „Kapcsolódó minták" leírásban mely más mintákra hivatkoznak. Az 1.1 ábra ezeket a viszonyokat festi le grafikusan.

Nyilvánvalóan sokféle módon csoportosíthatjuk még a tervezési mintákat, és a több szinten való gondolkodás elmélyítheti rálátásunkat arra, hogy mit csinálnak, miben hasonlítanak egymásra, és mikor kell alkalmazni őket.

1.6 Hogyan oldják meg a tervezési minták a tervezési problémákat?

A tervezési minták számos olyan mindennapos problémát oldanak meg, amelyekkel a tervezők szembetalálkoznak, méghozzá sokféleképp. Alább felsoroltunk párat ezen problémák közül, és leírtuk, hogyan oldják meg ezeket a tervezési minták.

Megfelelő objektumok keresése

Az objektumközpontú programok objektumokból épülnek fel. Az **objektumok** magukba foglalják mind az adatokat, mind az adatokon dolgozó eljárásokat, amelyeket **(tag)függvényeknek**, metódusoknak, vagy **műveleteknek** hívunk. Az objektum akkor hajt végre egy műveletet, ha az **ügyféltől kérelem** (vagy **üzenet**) érkezik.

Egy objektum *kizárólag* akkor hajthat végre műveletet, ha kérelem érkezik hozzá, belső adatai pedig *kizárólag* műveletekkel változtathatók meg. Ezek miatt a megszorítások miatt az objektum belső állapota *egységbe van zárva*, vagyis nem érhető el közvetlenül, és az objektumon kívülről láthatatlan.

Az objektumközpontú tervezés nehézsége az adott rendszer objektumokra való felbontásában rejlik. A feladat azért nehéz, mert sok tényező jön számításba: az egységbe zárás (betokozás), a „finomság" (részletesség, „szemcsézettség"), a függőségek, a rugalmasság, a teljesítmény, a továbbfejleszthetőség, az újrahasznosíthatóság és így tovább. Ezek mind befolyásolják a felbontást, sokszor egymásnak ellent mondva.

Az objektumközpontú tervezés módszertana számos megközelítést támogat. Az egyik, hogy körülírjuk a feladatot, majd főneveket és igéket társítunk hozzá, végül ezek segítségével megalkotjuk a megfelelő osztályokat és műveleteket. De helyezhetjük a hangsúlyt a rendszer elemeinek együttműködésére és a felelősségi körökre is. Vagy modellezhetjük a valódi világot, és az annak elemzése közben „talált" objektumokat foglalhatjuk rendszerbe. Arról, hogy melyik megközelítés a legjobb, valószínűleg mindig vita fog folyni.

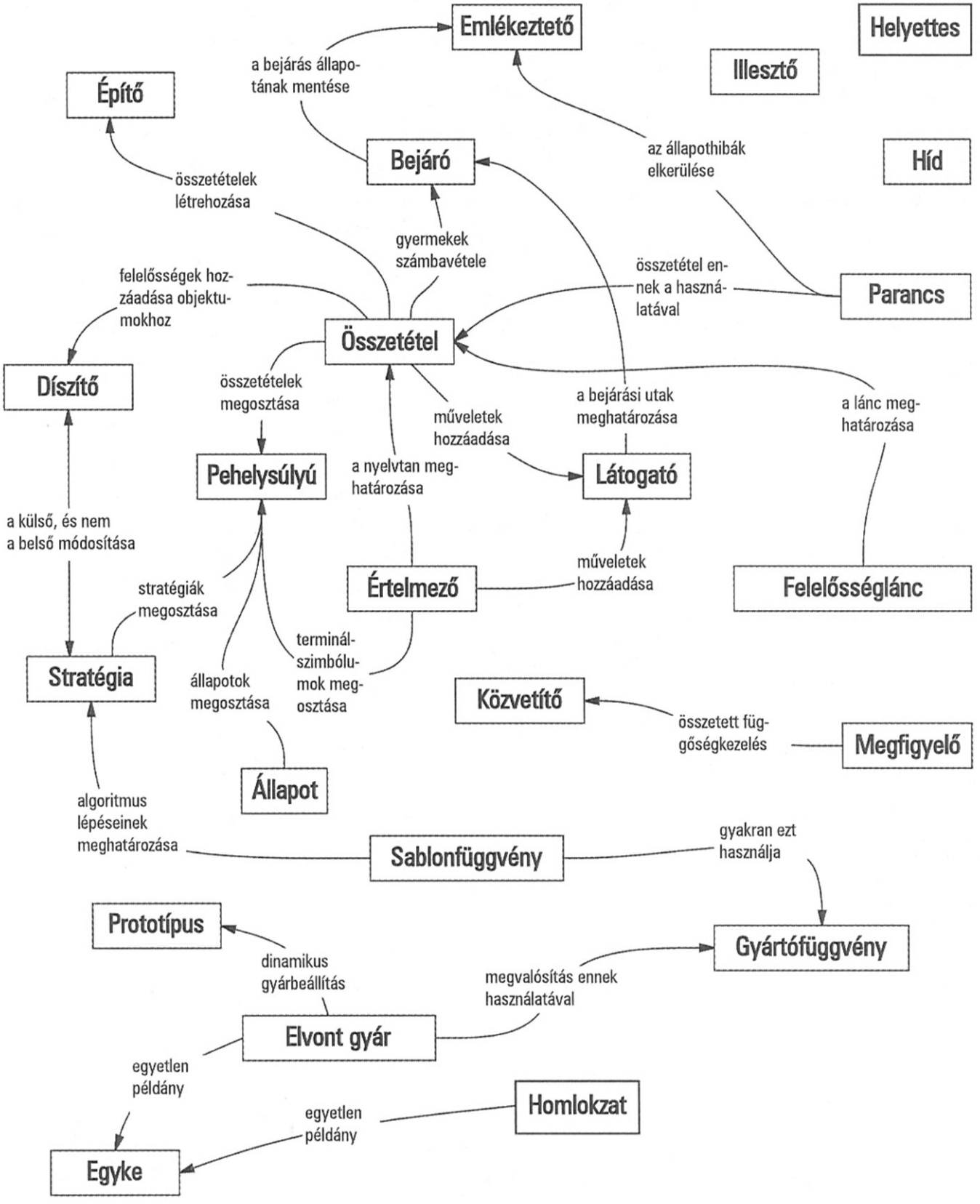
A tervezés során számos objektumot az elemző modellből származtatunk, de az objektumközpontú rendszerekben végül gyakran olyan osztályok szerepelnek, amelyeknek a valódi világban nincs megfelelőjük. Ezek egy része alsó szintű osztály, amilyen például a tömb. Mások sokkal magasabb szintűek. Az Összetétel minta például egy olyan elvont rendszert ír le, amelyben fizikai megfelelővel nem rendelkező objektumokat kezelhetünk egységesen. A valódi világ szigorú modellezése olyan rendszerhez vezet, ami ugyan bemutatja a ma valóságát, de a holnapét már nem feltétlenül. A tervezés során felmerülő elvont fogalmak („absztrakciók") kulcsfontosságúak a terv rugalmassága szempontjából.

A tervezési minták segítenek a kevésbé egyértelmű fogalmak és az azokat ábrázoló objektumok felismerésében. Például azok az objektumok, amelyek folyamatokat vagy algoritmusokat képviselnek, a természetben nem fordulnak elő, mégis fontos részei a rugalmas mintáknak. A Stratégia minta azt írja le, hogyan valósítsunk meg felcserélhető algoritmuscsaládokat. Az Állapot minta egy adott egyed összes állapotát objektumként ábrázolja. Ezek az objektumok ritkán találhatók meg a valóság elemzésével vagy a tervezés korai szakaszában; később kerülnek napfényre, amikor a tervet igyekszünk rugalmasabbá és újrahasznosíthatóbbá tenni.

A szükséges objektumok megkeresése

Az objektumok mérete és száma a különböző rendszerekben nagyfokú eltérést mutathat. Képviselhetnek mindent, egészen a hardvereszközökig, de átfoghatnak nagyobb egységeket is, a teljes alkalmazások szintjéig. Hogyan döntsük el, hogy mi is legyen egy objektum?

A tervezési minták erre a kérdésre is választ adnak. A Homlokzat minta azt írja le, hogyan ábrázoljunk objektumokként egész alrendszereket, míg a Pehelysúlyú minta azt, hogy ho-



1.1 ábra

Tervezésiminta-kapcsolatok.

gyan támogassunk nagyon sok, „nagy finomságú" (kis részleteket leíró) objektumot. Más tervezési minták különböző módokat írnak le az objektumok kisebb objektumokra való szétbontására. Az Elvont gyár és az Építő minták olyan objektumokat eredményeznek, amelyeknek egyetlen feladata más objektumok létrehozása, a Látogató és a Parancs minták pedig olyanokat, amelyek egyetlen célja kérelmeket intézni objektumokhoz vagy objektumcsoportokhoz.

Az objektumfelületek meghatározása

Minden objektum által leírt művelet megadja a művelet nevét, az objektumokat, amiket paraméterként kap, és a művelet visszatérési értékét. Ezt hívjuk a művelet **aláírásának** („szignatúra"). Az objektum műveletei által meghatározott összes ilyen aláírás halmaza az objektum felülete. A **felület** jellemzi az objektumnak küldhető kérelmek teljes halmazát. Bármely, az objektum felületében lévő aláírásnak megfelelő kérelem elküldhető az objektumnak.

A **típus** egy név, ami egy bizonyos felületet jelent. Azt mondjuk, hogy egy objektum „ablak" típusú, ha az „ablak" nevű felületben értelmezett műveleteknek címzett kérést fogad. Egy objektumnak több típusa is lehet, és sokban különböző objektumok is osztozhatnak egy típuson. Az objektum felületének egy részét egy típus jellemezheti, míg a többit más típusok. Két azonos típusú objektumnak csak felületük egy részében kell egyeznie. A felületek más felületeket is tartalmazhatnak részhalmazként. Egy típus akkor **altípusa** egy másiknak, ha a felülete tartalmazza az **őstípus** felületét. Sokszor úgy mondjuk, hogy az altípus *örökli* az őstípus felületét.

A felületek alapvető fontosságúak az objektumközpontú rendszerekben. Az objektumokat csak a felületeiken keresztül ismerjük. Nincs rá mód, hogy bármit is megtudjunk egy objektumról, vagy megkérjük, hogy tegyen valamit, anélkül, hogy a felületén át ne haladnánk. Az objektum felülete semmit nem mond a megvalósításáról – a különböző objektumok különböző módokon teljesíthetik a kéréseket. Vagyis két objektum, amelynek teljesen más a megvalósítása, felületében teljesen megegyezhet.

Ha egy objektumnak kérelmet küldünk, a művelet, ami végrehajtódik, függ mind a kérelemtől, mind a fogadó (vevő) objektumtól. Különböző objektumok, amelyek ugyanolyan kéréseket támogatnak, különbözhetnek az ezeket végrehajtó műveletek megvalósításában. A kérelmek objektumokhoz és azoknak egy műveletéhez való futásidőben történő hozzárendelését dinamikus kötésnek vagy **késői kötésnek** nevezzük.

A késői kötés azt jelenti, hogy egy kérelem kiadása nem határozza meg a konkrét megvalósítást, egészen a program futásáig. Következésképp írhatunk olyan programokat, amelyek egy bizonyos felülettel rendelkező objektumot várnak, tudván, hogy bármelyik olyan objektum elfogadja majd a kérelmet, amelynek megfelelő a felülete. A késői kötés emellett megengedi, hogy egyforma felülettel rendelkező objektumokat futásidőben egymással helyettesítsünk. Ennek a helyettesíthetőségnek a neve **többalakúság** (polimorfizmus), és kulcs-

fontosságú elv az objektumközpontú rendszerekben. Leszűkíti a feltételezések körét, amit az ügyfélobjektumnak más objektumokról tennie kell azon kívül, hogy támogatják az adott felületet. A többalakúság leegyszerűsíti az ügyfelek meghatározását, szétválasztja az objektumokat, és lehetővé teszi, hogy azok kapcsolata futásidőben módosuljon.

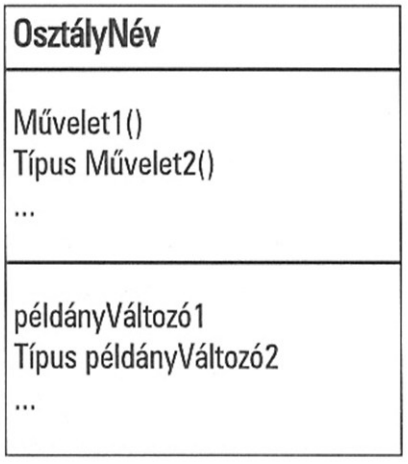
A tervezési minták azzal segítenek a felületek meghatározásában, hogy azonosítják kulcselemeiket és a felületen áthaladó adatokat. Egy tervezési minta azt is megmondhatja, hogy mit *ne* tegyünk a felületbe. Az Emlékeztető minta jó példa erre. Leírja, hogyan zárjuk egységbe és mentsük egy objektum belső állapotát, hogy azt majd később visszaállíthassuk ebbe az állapotába. A minta előírja, hogy az emlékeztető objektumoknak két felületet kell meghatározniuk: egy korlátozottat, amivel az ügyfelek tárolhatnak és másolhatnak emlékeztetőket, és egy kitüntetettet, amit csak az eredeti objektum használhat az állapot elraktározására és visszaállítására.

A tervezési minták a felületek közötti kapcsolatokat is meghatározzák. Sokszor elvárják, hogy egyes osztályoknak hasonló legyen a felülete, vagy megszorításokat adnak az osztályok felületére. Például mind a Díszítő, mind a Helyettes minta elvárja a Díszítő és Helyettes objektumok felületétől, hogy megegyezzenek a díszített, illetve helyettesített objektumok felületével. A Látogató mintában a Látogató felületének minden objektumosztályt mutatnia kell, amit a látogatók látogathatnak.

Az objektummegvalósítások meghatározása

Eddig keveset mondtunk arról, hogy ténylegesen hogyan is határozunk meg egy objektumot. Nos, az objektum megvalósítását az **osztálya** határozza meg. Az osztály adja meg az objektum belső adatait és ábrázolását, illetve a műveleteket, amelyeket az objektum végre tud hajtani.

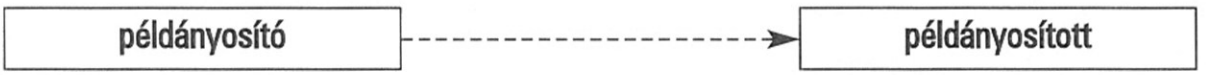
OMT alapú jelölésünk (amit a B függelékben foglaltunk össze) az osztályokat téglalappal ábrázolja, félkövérrel szedett névvel. A műveletek normál betűkkel szedve, az osztály neve alatt találhatók. Bármely adat, amit az osztály határoz meg, a műveletek után következik. Vonalak választják el az osztály nevét a műveletektől, és a műveleteket az adatoktól:



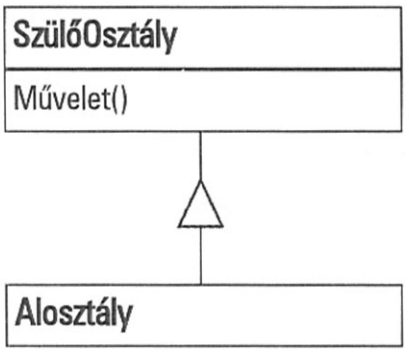
A visszatérési típusok és változópéldány-típusok elhagyhatók, mivel nem tételezünk fel statikusan típusos megvalósítási nyelvet.

Az objektumok egy osztály **példányosításával** jönnek létre. Az objektum az osztály egy **példánya**. Az osztály példányosításának folyamata során az osztály tárhelyet juttat az objektum belső adatainak (amelyek **változópéldányokból** épülnek fel) és hozzárendeli a műveleteket ezekhez az adatokhoz. Az osztály példányosításával sok hasonló objektumpéldány jöhet létre.

Az alábbi ábrán a szaggatott, nyílhegyű vonal egy olyan osztályt jelöl, ami egy másik osztály objektumát példányosítja. A nyíl a példányosított objektumok osztályára mutat.



A már létező osztályokból az **osztályöröklés** segítségével hozhatunk létre új osztályokat. Ha egy **alosztály** örököl egy **szülőosztálytól**, akkor a szülő által leírt minden adatot és műveletet meghatároz. Az alosztály objektumainak példányai minden, az alosztály és a szülőosztályok által meghatározott adatot tartalmaznak, és minden műveletet végre tudnak majd hajtani, amit az alosztály és a szülőosztályai tudnak. Az alosztály kapcsolatot függőleges vonallal és egy háromszöggel ábrázoljuk:

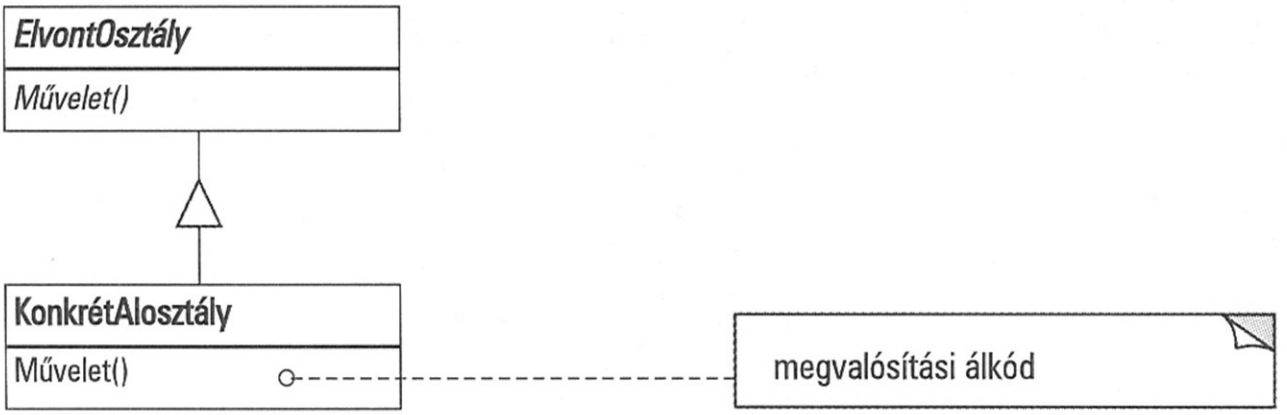


Az **elvont osztály** (absztrakt osztály) lényege, hogy általános felületet ír le az alosztályai számára. Az elvont osztály megvalósítása egy részét vagy egészét olyan műveletekre ruházza át, amelyeket alosztályai határoznak majd meg, vagyis az elvont osztályból nem készíthetünk példányt. Azokat a műveleteket, amelyeket egy elvont osztály bevezet, de meg nem valósít, **elvont műveleteknek** hívjuk. Azon osztályok neve, amelyek nem elvontak, **konkrét osztály**.

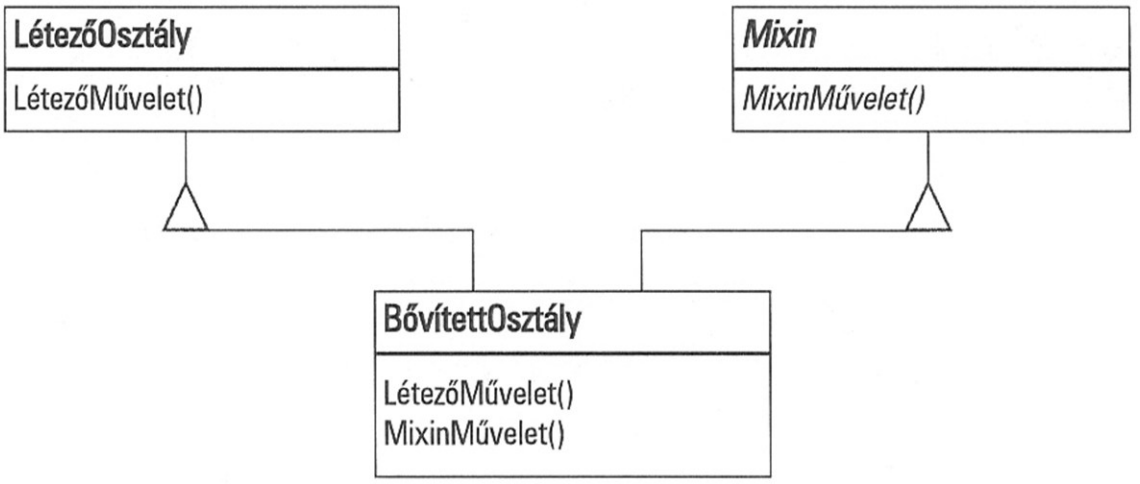
Az alosztályok finomíthatják vagy felülbírálhatják szülőosztályuk viselkedését. Konkrétabban, egy osztály **felülírhat** egy műveletet, amit a szülőosztálya határozott meg. A felülírás (felülbírálás) megadja az esélyt az alosztályoknak, hogy maguk kezeljék a kérelmeket a szülő-

osztályaik helyett. Az öröklés lehetővé teszi az osztályok más osztályok egyszerű kibővítésével való meghatározását, így könnyen alkothatunk objektumcsaládokat, amelyeknek rokon feladatuk van.

Az elvont osztályok nevét dőlt betűvel szedtük, hogy megkülönböztethetőek legyenek a konkrét osztályoktól. Dőlt betű jelzi az elvont műveleteket is. Néha az ábrán egy művelet megvalósításaként álkód (pszeudokód) tűnik fel, ilyenkor a kód egy szamárfüles dobozban lesz, amit szaggatott vonal köt össze az őt megvalósító művelettel.



A mixin („bekeveredő", „bekevert" osztály) olyan osztály, ami másik választható felületet vagy szolgáltatást nyújt más osztályoknak. Az elvont osztályhoz hasonlóan ezt sem lehet példányosítani. A mixin osztályokhoz többszörös öröklés szükséges:



Osztály- és felületöröklés

Fontos, hogy megértsük a különbséget az objektum *osztálya* és *típusa* között. Az osztály az adott objektum megvalósításának módját, vagyis az objektum belső állapotát és műveleteinek megvalósítását határozza meg. Ezzel szemben az objektum típusa csak a felületére utal, vagyis azoknak a kérelmeknek a halmazára, amire válaszolni tud. Egy objektumnak több típusa lehet, és különböző osztályú objektumok is tartozhatnak egy típusba.

Természetesen közeli rokonság van az osztály és a típus között. Mivel az osztály meghatározza az objektum által végrehajtható műveleteket, az objektum típusát is meghatározza. Ha azt mondjuk, hogy egy objektum egy osztály példánya, beleértjük azt is, hogy az objektum támogatja az osztály által értelmezett felületet.

Az olyan nyelvek, mint a C++ vagy az Eiffel, osztályokkal határozzák meg az objektum típusát és megvalósítását is. A Smalltalk programok nem adják meg a változók típusát; vagyis a fordító nem ellenőrzi, hogy egy adott változóhoz rendelt objektumok típusai altípusai-e a változó típusának. Üzenet küldéséhez meg kell néznünk, hogy a fogadó osztályának megvalósítása támogatja-e az üzenetet, de nem kell megnéznünk, hogy a fogadó példánya-e egy bizonyos osztálynak.

Fontos megérteni továbbá az osztályöröklés és a felületöröklés (vagy altípus-létrehozás) közötti különbséget is. Az osztályöröklés egy objektum megvalósítását egy másik objektum megvalósításának segítségével adja meg. Röviden, ez egy olyan módszer, amivel kód és ábrázolás osztható meg. Ezzel szemben a felületöröklés azt írja le, hogy egy objektumot mikor használhatunk egy másik helyett.

Könnyű összekeverni ezt a két fogalmat, mivel sok nyelv nem tesz egyértelmű különbséget. Az olyan nyelvekben, mint a C++ és az Eiffel, az öröklődés egyszerre jelenti a felület és a megvalósítás öröklését. Az általános felületöröklési mód a C++-ban az, ha nyilvánosan olyan osztálytól öröklünk, amelynek (tisztán) virtuális tagfüggvényei vannak. A tiszta felületöröklést úgy közelíthetjük meg e nyelvben, hogy nyilvánosan öröklünk tisztán elvont osztályokból. A tiszta megvalósítás- vagy osztályöröklés privát örökléssel közelíthető. A Smalltalkban az öröklés csak megvalósítás-öröklést jelent. Bármely osztály példányát hozzá lehet rendelni egy változóhoz, amíg a példány támogatja a változó értékén végzett műveletet.

Bár a legtöbb programozási nyelv nem tesz különbséget a felület- és megvalósítás-öröklés között, a gyakorlatban mégis van megkülönböztetés. A Smalltalk programozók általában úgy tesznek, mintha az alosztályok altípusok lennének (bár van pár jól ismert kivétel [Coo92]); a C++ programozók pedig elvont osztályokban meghatározott típusokon keresztül kezelik az objektumokat.

Sok tervezési minta függ ettől a különbségtől. Például a Felelősséglánc minta objektumainak közös típusuk kell, hogy legyen, de megvalósításuk általában különböző. Az Összetétel mintában az Elem (Component) közös felületet határoz meg, de az Összetétel (Composite) gyakran közös megvalósítást. A Parancs, a Megfigyelő, az Állapot és a Stratégia mintákat sokszor elvont osztályokkal valósítják meg, amelyek tisztán felületek.

Megvalósítás helyett felületre programozás

Az osztályöröklés alapjában véve csak egy módszer, amivel az alkalmazás szolgáltatásait a szülő szolgáltatásaival bővítjük ki. Segítségével gyorsan hozhatunk létre új objektumokat egy régi alapján. Különösebb munka nélkül kaphatunk új megvalósításokat, egyszerűen örökölve a létező osztályokból, amire szükségünk van.

Mindazonáltal a megvalósítás újrahasznosítása még nem minden. Az öröklés azon tulajdonsága, hogy *egyező* felületű objektumok családját képes meghatározni (általában egy elvont osztályból örökölve) szintén fontos. Miért? Mert ezen alapul a többalakúság.

Ha figyelmesen használjuk az öröklést (egyesek szerint *szabályosan*), akkor minden, adott elvont osztályból származó osztály osztozik majd a szülő felületén. Ebből az következik, hogy egy alosztály a műveletekhez csak adhat, vagy felülírhatja azokat, de szülőosztályának műveleteit nem tudja elrejteni. Így *minden* alosztály tud majd válaszolni az elvont osztály felületében szereplő kérelmekre, amik így az elvont osztály altípusává válnak.

Két előny is származik abból, ha az objektumokat kizárólag az elvont osztályban meghatározott felület szintjén kezeljük:

1. Az ügyfelek egészen addig nem veszik figyelembe a felhasznált objektumok típusát, amíg az objektumok felülete az ügyfelek által vártnak megfelel.
2. Az ügyfelek nem veszik figyelembe az ezeket az objektumokat megvalósító osztályokat. Csak a felületet meghatározó elvont osztály(oka)t ismerik.

Ez annyira lecsökkenti az alrendszerek közötti megvalósítás-függőséget, hogy az az újrahasznosítható objektumközpontú terv első alapelvéhez vezet:

Programozzunk a felületre a megvalósítás helyett.

Ne konkrét osztályok példányaiként vezessük be változóinkat, csak az elvont osztályban meghatározott felületet bővítsük. Ahogy látni fogjuk, ez a könyvben lévő tervezési minták egyik közös elve.

Persze valahol a rendszerünkben példányosítanunk kell majd konkrét osztályokat (vagyis meg kell adnunk a tényleges megvalósítást), és a létrehozási minták (Elvont gyár, Építő, Gyártófüggvény, Prototípus és Egyke) éppen ezt teszik lehetővé. Az objektum-létrehozás folyamatának elvonatkoztatásával ezek a minták különböző módokat nyújtanak arra, hogy egy felületet a megvalósításával anélkül kapcsolhassuk össze a példányosításnál, hogy a megvalósítás módjáról tudomásunk lenne. A létrehozási minták biztosítják, hogy a rendszerünk felület-, és ne megvalósítás-központú legyen.

Az újrahasznosítási szerkezetek használata

A legtöbben megértik az objektumok, felületek, osztályok és az öröklés fogalmát. A kihívás igazán abban rejlik, hogy alkalmazásukkal rugalmas, újrahasznosítható programokat építsünk, és a tervezési minták megmutatják, hogy ezt hogyan is kell.

Öröklés vagy összetétel?

Az objektumközpontú rendszerekben a képességek újrahasznosításának két leggyakrabban használt módszere az osztályöröklés és az **objektum-összetétel** (objektumkompozíció). Ahogy azt már említettük, az osztályöröklés arra ad módot, hogy egy osztály megvalósítását egy másik osztály segítségével határozzuk meg. Az alosztályokon keresztül történő újrahasznosítást **fehér dobozos újrahasznosításnak** nevezzük. A „fehér doboz" a láthatóságra utal: az örökléssel az alosztályok gyakran látják a szülő osztály belső részeit.

Az objektum-összetétel az osztályöröklés alternatívája. Itt az új szolgáltatások úgy jönnek létre, hogy kisebb részekből építünk fel objektumokat, hogy több szolgáltatással rendelkezzenek. Az objektum-összetételnél az összeépített objektumoknak jól meghatározott felülettel kell rendelkezniük. Az ilyen újrahasznosítást **fekete dobozos újrahasznosításnak** nevezzük, mert az objektumok belső részei láthatatlanok. Az objektumok „fekete dobozokként" jelennek meg.

Az öröklésnek és az összetételnek egyaránt megvannak a maga előnyei és hátrányai. Az öröklődés statikusan, fordításkor történik, és használata egyértelmű, mivel közvetlenül a programnyelv támogatja; továbbá az osztályöröklés könnyebbé teszi az újrahasznosított megvalósítás módosítását is. Ha egy alosztály felülírja a műveletek némelyikét, de nem mindet, akkor a leszármazottak műveleteit is megváltoztathatja, feltételezve, hogy azok a felülírt műveleteket hívják.

De az osztályöröklésnek vannak hátrányai is. Először is, a szülőosztályoktól örökölt megvalósításokat futásidőben nem változtathatjuk meg, mivel az öröklés már fordításkor eldől. Másodszor – és ez sokkal rosszabb –, a szülőosztályok gyakran alosztályaik fizikai megjelenését is meghatározzák, legalább részben. Mivel az öröklés megengedi, hogy egy alosztály betekintést nyerjen szülője megvalósításába, gyakran mondják, hogy „az öröklés megszegi az egységbe zárás szabályát" [Sny86]. Az alosztály megvalósítása annyira kötődik a szülőosztály megvalósításához, hogy a szülő megvalósításában a legkisebb változtatás is az alosztály változását vonja maga után.

A megvalósítási függőségek gondot okozhatnak az alosztályok újrahasznosításánál. Ha az örökölt megvalósítás bármely szempontból nem felel meg az új feladatnak, arra kényszerülünk, hogy újraírjuk vagy valami megfelelőbbel helyettesítsük a szülőosztályt. Ez a függőség korlátozza a rugalmasságot, és végül az újrahasznosíthatóságot. Ezt úgy orvosolhatjuk, ha csak elvont osztályoktól öröklünk, mivel azok általában semennyi vagy csak kevés megvalósításra vonatkozó részt tartalmaznak.

Az objektum-összetétel dinamikusan, futásidőben történik, olyan objektumokon keresztül, amelyek hivatkozásokat szereznek más objektumokra. Az összetételhez szükséges, hogy az objektumok figyelembe vegyék egymás felületét, amihez figyelmesen megtervezett felületek kellenek, amelyek lehetővé teszik, hogy az objektumokat sok másikkal együtt használjuk. A módszer előnye viszont, hogy mivel az objektumokat csak a felületükön keresztül érhetjük el, nem szegjük meg az egységbe zárás elvét. Bármely objektumot lecserélhetünk egy másikra futásidőben, amíg a típusaik egyeznek. Továbbá, mivel az objektumok megvalósítása objektumfelületek segítségével épül fel, sokkal kevesebb lesz a megvalósítási függőség.

Az objektum-összetételnek még egy hatása van a rendszer szerkezetére: az osztályörökléssel szemben segít az osztályok egységbe zárásában és abban, hogy azok egy feladatra összpontosíthassanak. Az osztályok és osztályhierarchiák kicsik maradnak, és kevésbé valószínű, hogy kezelhetetlen szörnyekké duzzadnak. Másrészről az objektum-összetételen alapuló tervezésben több objektumunk lesz (ha osztályunk kevesebb is), és a rendszer viselkedése ezek kapcsolataitól függ majd, nem pedig egyetlen osztály határozza meg.

Ez vezet el minket az objektumközpontú tervezés második alapelvéhez:

Használjunk objektum-összetételt osztályöröklés helyett, amikor csak lehet.

A legjobb az lenne, ha nem kellene új elemeket létrehoznunk, hogy valamit újra felhasználhassunk, és minden szükséges képességre szert tehetnénk, ha összepárosítanánk a létező objektumokat. De ez ritkán történik meg, mivel a rendelkezésünkre álló elemek tárháza a gyakorlatban soha nem elég gazdag. Az öröklésből eredő újrahasznosíthatóság viszont könnyebbé teszi új elemek építését a régiekből: az öröklés és az összetétel tehát együtt használatosak.

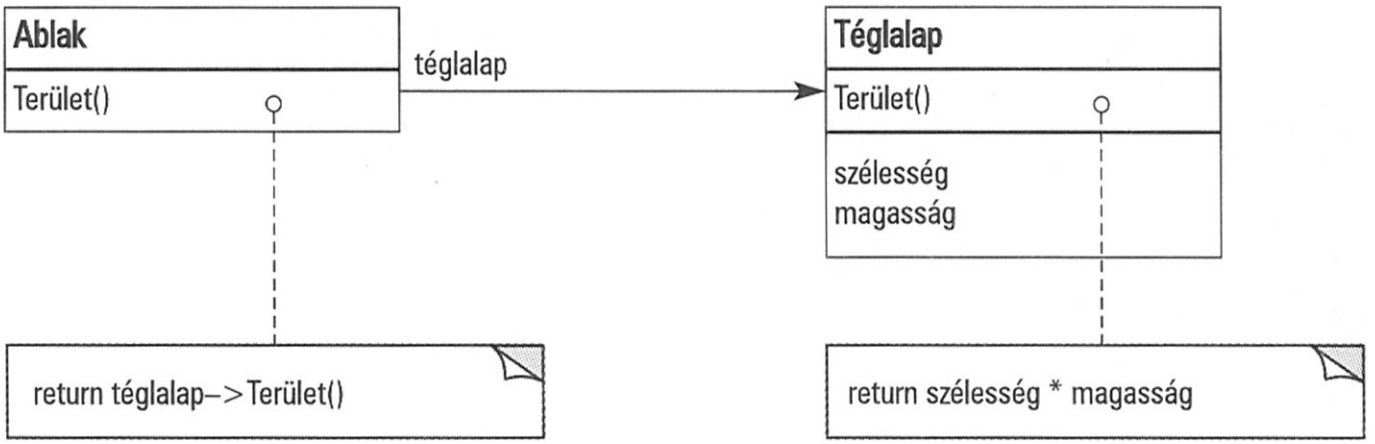
A gyakorlat azonban azt mutatja, hogy a tervezők túlzott előszeretettel használják az öröklést, mint újrahasznosítási módszert, és a tervek sokszor újrahasznosíthatóbbá (és egyszerűbbé) válnának, ha gyakrabban fordulnánk az objektum-összetételhez. A bemutatott tervezési mintákban, mint majd láthatjuk, az objektum-összetétel újra és újra felbukkan.

Képviselet

A **képviselet** (delegáció) az összetétel olyan erejű újrahasznosítási módszerré tétele, mint az öröklés [Lie86, JZ91]. A kérelmek kezelésében ekkor *két* objektum vesz részt: egy fogadó objektum, és a **képviselője** (delegáltja), amelyre a fogadó műveleteket ruház át. Ez az alosztályok szülőknek címzett kérelem-átirányításához hasonló. De az öröklésnél az örökölt műveletek mindig utalhatnak a fogadó objektumra, amit a C++-ban a `this`, a Smalltalkban a `self` változó valósít meg. Hogy ugyanezt a hatást érjük el a képviselettel, a fogadó „átadja magát" a képviselőnek, hogy az átruházott művelet a fogadóra utalhasson.

Például ahelyett, hogy az Ablak alosztálya lenne a Téglalapnak (mivel az ablakok téglalap alakúak), az Ablak osztály felhasználhatná a Téglalap viselkedését azzal, hogy egy saját Téglalap példányváltozóra Téglalap-szerű viselkedést ruház. Más szavakkal, ahelyett, hogy egy Ablak Téglalap lenne, egyszerűen lenne egy Téglalapja. Az Ablaknak mostantól kifejezetten továbbítania kell a kérelmeket a Téglalap-példányának, pedig eddig örökölte volna ezeket a műveleteket.

A következő ábra azt mutatja, ahogyan az Ablak osztály átadja a Terület műveletét a Téglalap-példánynak.



Az egyszerű nyílban végződő vonal azt mutatja, hogy egy osztály egy másik osztály egy példányára hivatkozik. A hivatkozásnak külön neve is lehet, ebben az esetben „teglalap".

A képviselet legfőbb előnye az, hogy segítségével könnyen alakíthatunk ki viselkedéseket futásidőben, és megváltoztathatjuk összetételük módját. Ablakunk kör alakúvá válhat futásidőben, egyszerűen kicserélve a Téglalap-példányt egy Kör-példányra, amennyiben a Téglalap és a Kör azonos típusúak.

A képviseletnek ugyanaz a hátránya, mint általában a többi objektum-összetételi rugalmasságnövelő eljárásnak: a dinamikus, erősen paraméterezett programokat nehezebb megérteni, mint a statikusabbakat. Vannak ugyan futásidejű hátrányai is, de az emberi tényezőt érintő hátrányok hosszú távon sokkal fontosabbak. A képviselet csak akkor jó választás, ha többet egyszerűsít, mint bonyolít. Nem könnyű szabályokat adni rá, hogy mikor érdemes használni, mivel hatékonysága azon múlik, milyen környezetben használjuk, és mennyi tapasztalatunk van a használatában. A képviselet legjobban erősen stilizált formában használható – vagyis a szabványos mintákban.

Több tervezési minta használja a képviseletet. Az Állapot, a Stratégia és a Látogató minták erősen függnek tőle. Az Állapot mintában az objektumok egy Állapot objektumnak adnak át kérelmeket, ami az adott objektum aktuális állapotát tükrözi. A Stratégia mintában az ob-

jektumok adott kérelmet ruháznak át egy objektumra, ami a kérelem teljesítésének módját képviseli. Egy objektumnak csak egy állapota, de több módszere (stratégiája) is lehet a különböző kérelmekhez. Mindkét minta célja, hogy megváltoztassa egy objektum viselkedését azzal, hogy megváltoztatja az objektumokat, amelyeknek a kérelmeket átadja. A Látogató mintában az egy adott objektumszerkezet minden elemén végrehajtott műveletet mindig a Látogató objektum veszi át.

Más minták kevésbé használják a képviseletet. A Közvetítő minta egy objektumot vezet be, ami közvetít más objektumok között. Néha a közvetítő objektum úgy valósítja meg a műveleteket, hogy egyszerűen továbbítja azokat más objektumoknak; máskor hivatkozást is küld magára, vagyis valódi képviseletet használ. A Felelősséglánc minta úgy kezeli a kérelmeket, hogy objektumok láncán keresztül továbbítja azokat egyik objektumtól a másikig. Néha ez a kérelem magával viszi az eredeti fogadó objektumra való hivatkozást, és ilyenkor a minta képviseletet használ. A Híd minta szétválasztja az elvont fogalmat a megvalósításától. Ha a fogalom és egy adott megvalósítása szorosan összekapcsolódnak, az elvont ábrázolás egyszerűen ruházhat át műveleteket az adott megvalósításra.

A képviselet szélsőséges példa az objektum-összetételre, ami azt mutatja, hogy az öröklés, mint kód-újrahasznosítási megoldás, mindig helyettesíthető objektum-összetétellel.

Öröklés és paraméterezett típusok

Egy másik (nem szigorúan objektumközpontú) eljárás a szolgáltatások újrahasznosítására a **paraméterezett típusok** használata, amiket generikus vagy **általános programozási alegységeknek** (Ada, Eiffel), illetve **sablonoknak** (template, C++) hívunk. Ez a módszer egy típus meghatározását teszi lehetővé anélkül, hogy minden általa felhasznált típust megadnánk. A meghatározatlan típusokat *paraméterekként* látjuk a használat helyén. Például egy Lista osztályt elemeinek típusával paraméterezhetünk. Ahhoz, hogy egészek egy listáját vezessük be, az „integer" típust adjuk át paraméterként a paraméterezett típusú Listának. Ha egy karakterlánc objektumokból álló listát szeretnénk megadni, a „String" típust adjuk át paraméterként. A nyelvi megvalósítás a Lista osztálysablon testreszabott változatát készíti el minden elemtípushoz.

A paraméterezett típusokkal egy harmadik módszert kaptunk (az osztályöröklés és az objektum-összetétel mellé), amellyel objektumközpontú rendszerek viselkedését építhetjük fel. Sokféle tervet megvalósíthatunk e három módszer használatával. Ha egy rendező eljárásnak paraméterként meg szeretnénk adni, hogy milyen műveletet használjon az elemek összehasonlítására, ezt a következő eszközökkel valósíthatjuk meg:

1. alosztályokkal megvalósított művelet (Sablonfüggvény felhasználása),
2. egy objektum kötelezettsége, amit a rendező eljárásnak átadunk (Stratégia), vagy
3. egy C++ sablon vagy Ada általános programozási alegység argumentuma, ami meghatározza az elemek összehasonlítására szolgáló függvény nevét.

A fenti módszerek között lényeges különbségek vannak. Az objektum-összetétel segítségével megváltoztathatjuk a futásidőben létrejövő viselkedést, de közvetettségre van szükség, ami néha kevésbé hatékony. Az öröklés segítségével alapértelmezett megvalósítással láthatjuk el a műveleteket, amit majd az alosztályok felülírhatnak. A paraméterezett típusok segítségével megváltoztathatók az osztályok által használt típusok. De sem az öröklés, sem a paraméterezett típus nem változhat futásidőben. Hogy melyik megközelítés a legjobb, az a terven és a megvalósítás megszorításain múlik.

Az e könyvben leírt minták egyike sem foglalkozik a paraméterezett típusokkal, bár esetenként használjuk minták C++ megvalósításának testreszabására. A paraméterezett típusokra semmi szükség az olyan nyelvekben, mint a Smalltalk, amelyekben nincs fordítási idejű típusellenőrzés.

Futás- és fordítási idejű szerkezetek összehasonlítása

Egy objektumközpontú program futásidejű szerkezete gyakran kevés hasonlatosságot mutat a kód szerkezetével. Fordításkor a kód szerkezete statikus; meghatározott öröklési kapcsolatban álló osztályokból áll. A program futásidejű szerkezete egymással társalgó objektumok gyorsan változó hálózatából áll. Valójában a két szerkezet egymástól nagyban független. Ha az egyikből szeretnénk megérteni a másikat, az olyan, mintha az élővilág dinamikáját szeretnénk megérteni pusztán az állatok és növények rendszertani leírásából, és fordítva.

Vessünk egy pillantást az objektumok **összesítése** (aggregáció) és **ismeretsége** (acquaintance) közötti különbségre, és hogy mennyire máshogyan nyilvánulnak meg fordítási és futásidőben. Az összesítés azt sugallja, hogy az egyik objektum a másik tulajdonában van vagy felelős érte. Általában úgy mondjuk, hogy egy objektum egy másiknak *tulajdonosa* vagy *része*. Az összesítés arra utal, hogy az összesített objektumnak és tulajdonosának egyforma az életideje.

Az ismeretség arra utal, hogy egy objektum egyszerűen *tud* egy másikról. Néha az ismeretséget „asszociációnak" vagy „használó" kapcsolatnak (using) is hívják. Az ismerős objektumok kérhetnek műveleteket egymástól, de nem felelősek egymásért. Az ismeretség gyengébb kapcsolat az összesítésnél, és sokkal lazább összefüggést feltételez az objektumok között.

Ábráinkban a sima nyílhegyű vonal ismeretséget jelent. Az a nyíl viszont, amelynek végén rombusz található, összesítést jelöl:

Az összesítést és az ismeretséget könnyű összekeverni, mivel gyakran ugyanúgy valósítjuk meg őket. A Smalltalkban minden változó hivatkozás más objektumokra, a programozási nyelv nem tesz különbséget összesítés és ismeretség között. A C++-ban az összesítés megvalósítása tagváltozó-példányokkal történik, de gyakrabban történik a megvalósítás olyan mutatókkal és hivatkozásokkal, amelyek példányokra mutatnak. Az ismeretséget is mutatókkal és hivatkozásokkal valósítjuk meg.

Végső soron az ismeretséget és az összesítést inkább a cél határozza meg, nem kifejezett nyelvi eszközök. Talán nehéz felismerni a különbséget a fordítási idejű szerkezetben, de mégis fontos. Összesítő kapcsolatból általában kevesebb van, de azok tartósabbak az ismeretségeknél. Az ismeretségek ezzel szemben sokkal gyakrabban létesülnek újra és újra, néha csak egy művelet erejéig létezve. Az ismeretségek dinamikusabbak is, így nehezebb azokat megtalálni a forráskódban.

Ilyen különbségekkel a program futásidejű és fordítási idejű szerkezetében világos, hogy a kód nem fogja felfedni a rendszer teljes működését. A rendszer futásidejű szerkezetét inkább a tervező, mint a nyelv adja meg. Az objektumok és típusaik közti kapcsolatokat nagy gonddal kell megterveznünk, mert ezek határozzák meg, mennyire jó vagy rossz a futásidejű szerkezet.

Sok tervezési mintában (főleg amelyek objektumszinten működnek) látható a fordítási és futásidejű szerkezetek közti különbség. A Összetétel és a Díszítő minták különösen hasznosak a bonyolult futásidejű szerkezetek építésében. A Megfigyelő olyan futásidejű szerkezeteket tartalmaz, amelyeket gyakran nehéz megérteni a minta ismerete nélkül. A Felelősséglánc szintén olyan kommunikációs mintákat eredményez, amiket az öröklés nem mutat be. A futásidejű szerkezetek általában nem látszanak tisztán a kódból, amíg meg nem értjük a mintákat.

Változásra tervezve

Az újrahasznosíthatóság kulcsa az új igények és a létező igények változásának megérzésében rejlik, és abban, hogy úgy tervezzük meg rendszerünket, hogy az ennek megfelelően fejlődhessen.

Ahhoz, hogy olyan rendszert tervezzünk, ami a változásoknak is megfelel, figyelembe kell vennünk, hogy a rendszer milyen változásokon kell majd, hogy átessen életében. Az olyan terv, ami nem veszi figyelembe a változásokat, az alapos újratervezés későbbi szükségességének kockázatát rejti. A változások között előfordulhat osztályok új meghatározása és új megvalósítása, az ügyfél változása, és újratesztelés. Az újratervezés a programrendszer számos részét érinti, és a váratlan változások kivétel nélkül költségesek.

A tervezési minták segítségével ez elkerülhető, segítségükkel biztosíthatjuk, hogy a rendszer meghatározott módokon módosítható legyen. Minden tervezési mintában vannak a rendszerszerkezetnek olyan részei, amelyek a többitől függetlenül változtathatók, ezzel téve ellenállóbbá a rendszert egyes változásfajtákkal szemben.

Az alábbaikban leírtuk az újratervezés néhány gyakori okát, valamint azt, hogy az egyes tervezési minták milyen megoldást adnak ezekre a problémákra:

1. *Objektum létrehozása kifejezett osztálymegadással.* Ha egy objektum létrehozásánál megadjuk az osztály nevét, az egyetlen megvalósításra korlátoz, és nem egyfajta felületre. Ez a korlátozás bonyolíthatja a későbbi változtatásokat. Elkerüléséhez közvetve hozzuk létre az objektumokat.
   Tervezési minták: Elvont gyár, Gyártófüggvény, Prototípus.
2. *Konkrét műveletekre támaszkodás.* Ha konkrét műveleteket adunk meg, a kérelem egyfajta kielégítési módjára szorítkozunk. A megváltoztathatatlanul kódolt kérelmek mellőzésével könnyebbé válik a kérelem kielégítésének megváltoztatása mind fordításkor, mind futásidőben.
   Tervezési minták: Felelősséglánc, Parancs.
3. *Függőség a hardver- és programkörnyezettől.* A külső, operációs rendszeri és alkalmazás-programozási felületek (API-k) különböznek a különböző hardver- és programkörnyezetekben. A környezetfüggő programot nehezebb más környezetekhez illeszteni. Még az is lehet, hogy nehézzé válik a programot naprakészen tartani az eredeti környezetében. Ezért fontos, hogy rendszerünk minél környezetfüggetlenebbé váljon.
   Tervezési minták: Elvont gyár, Híd.
4. *Függőség az objektumok ábrázolásától és megvalósításától.* Azok az ügyfelek, akik tudják, hogy az objektumot hogyan ábrázoljuk, tároljuk, helyezzük el, vagy valósítjuk meg, lehet, hogy változásokra szorulnak, ha maga az objektum is változik. Ha ezeket az információkat elrejtjük az ügyfelek elől, nem kell annyit változtatnunk.
   Tervezési minták: Elvont gyár, Híd, Emlékeztető, Helyettes.
5. *Algoritmikus függőségek.* Az algoritmusok hatékonyságát gyakran utólag finomhangolják, kibővítik őket, vagy lecserélik a fejlesztés és újrahasznosítás során. Azok az objektumok, amelyek egy ilyen algoritmustól függnek, kénytelenek lesznek megváltozni, ezért azokat az algoritmusokat, amelyek valószínűleg meg fognak változni, el kell különíteni.
   Tervezési minták: Építő, Bejáró, Stratégia, Sablonfüggvény, Látogató.
6. *Szoros csatolás.* A szorosan összekapcsolt osztályokat nehéz elkülönítve felhasználni, mivel függnek egymástól. A szoros csatolás oszthatatlan rendszerekhez vezet, ahol nem változtathatunk vagy törölhetünk egy osztályt a rendszerből számos más osztály megértése és átírása nélkül. A rendszer nehezen átlátható, más rendszerre nehezen átvihető és alig kezelhető masszává válik.
   A laza csatolás növeli annak esélyét, hogy egy osztály önmagában is felhasználható lesz, és hogy a rendszert könnyebb lesz megtanulni, más rendszerre átvinni, módosítani és kibővíteni. A tervezési minták olyan eljárásokkal segítik a laza csatolású rendszereket, mint az elvont csatolás és a rétegezés.
   Tervezési minták: Elvont gyár, Híd, Felelősséglánc, Parancs, Homlokzat, Közvetítő, Megfigyelő.

7. *A működés kibővítése alosztályokkal.* Egy objektum testreszabása alosztályokkal nem mindig könnyű feladat. Minden új osztálynak egy rögzített megvalósításhoz kell igazodnia (előkészítéskor, lezáráskor stb.). Egy alosztály meghatározásához a szülő mélyreható megértése szükséges. Például egy művelet felülírása talán egy másik felülírását is maga után vonhatja, vagy lehet, hogy egy felülírt művelet szükséges egy örökölt művelet hívásához. Továbbá az alosztályok létrehozása az osztályok számának robbanásszerű növekedéséhez is vezethet, mivel már egy egyszerű bővítés is lehet, hogy több új alosztály bevezetését követeli meg.
   Az objektum-összetétel általában, a képviselet teljes egészében rugalmas alternatívákat kínálnak az öröklés helyett a viselkedések kialakítására. Egy alkalmazáshoz már létező objektumok összeépítésével, új alosztályok meghatározása nélkül adhatunk új szolgáltatásokat. Másrészről az objektum-összetétel túlzott használata megnehezítheti a szerkezet megértését. Több tervezési minta olyan terveket eredményez, amelyekben testhez álló szolgáltatásokat hozhatunk létre csupán egy alosztály meghatározásával, míg a példányokat létező osztályok összetételével állítjuk elő.
   Tervezési minták: Híd, Felelősséglánc, Összetétel, Díszítő, Megfigyelő, Stratégia.
8. *Osztályok kényelmetlen módosítása.* Van, amikor olyan osztályt kell megváltoztatnunk, amit kényelmesen lehetetlen. Talán a forráskódra lenne szükség, de nem áll rendelkezésünkre (például egy megvásárolható osztálykönyvtárnál), vagy bármilyen változás rengeteg alosztály megváltoztatását vonná maga után. A tervezési minták ezekre az esetekre is adnak útmutatást.
   Tervezési minták: Illesztő, Díszítő, Látogató.

Ezen példák jól mutatják, hogy a tervezési minták hogyan segíthetnek a program rugalmasabbá tételében. Azt, hogy ez a rugalmasság mennyire fontos, az építendő program jellege dönti el. Lássuk, hogyan működnek a tervezési minták három elég tág programcsoportban: az alkalmazásokban, az elemkészletekben és a keretrendszerekben.

Alkalmazások

Ha olyan alkalmazást írunk, mint például egy szövegszerkesztő vagy egy táblázatkezelő, akkor a *belső* újrahasznosíthatóság, a karbantarthatóság és a fejleszthetőség nagyon fontosak. A belső újrahasznosíthatóság biztosítja, hogy ne kelljen annál többet tervezni és megvalósítani, mint amit muszáj. Azok a tervezési minták, amelyek csökkentik a függőségeket, növelik a belső újrahasznosíthatóságot. A laza csatolás megnöveli annak esélyét, hogy egy objektum osztálya együtt tud működni több másikkal. Például amikor kizárjuk az adott műveletektől való függőségeket azzal, hogy minden műveletet elkülönítünk és egységbe zárunk, könnyebbé tesszük egy művelet különböző környezetekben történő felhasználását. Ugyanez történhet, ha eltávolítjuk az algoritmikus és ábrázolási függőségeket.

A tervezési minták továbbá könnyebben karbantarthatóbbá tesznek egy alkalmazást azzal, hogy csökkentik a környezeti függőségeket, és rétegezik a rendszert. Javítják a fejleszthetőséget azzal, hogy megmutatják, hogyan kell az osztályhierarchiákat kibővíteni, és hogyan kell az objektum-összetételt kihasználni. A fejleszthetőséget a laza csatolás is segíti. Egy elkülönített osztály fejlesztése könnyebb, mert nem kell sok függőséget figyelembe venni.

Elemkészletek

Az alkalmazások gyakran előre elkészített könyvtárakból használnak egy vagy akár több osztályt is. Ezeket az előre létrehozott osztályokat hívjuk **elemkészleteknek** (eszközkészlet, toolkit). Az elemkészlet kapcsolódó és újrahasznosítható osztályokból áll, amelyek hasznos, általános célú szolgáltatásokat nyújtanak. Egy példa az elemkészletre a gyűjteményosztályok készlete, amely a listák, asszociatív táblák, vermek és hasonlók osztályait tartalmazza. A C++ I/O (kimeneti–bemeneti) könyvtára is jó példa. Az elemkészletek nem erőszakolnak egy meghatározott tervet az alkalmazásra, csak olyan képességekkel látják el, amelyek megkönnyítik a munkáját, a mi számunkra pedig lehetővé teszik a megvalósítást az általános szolgáltatások újrakódolása nélkül. Az elemkészletek nagy hangsúlyt fektetnek a *kód újrahasznosítására*: tulajdonképpen az alprogram-könyvtárak objektumközpontú megfelelői.

Elképzelhető, hogy az elemkészlet-tervezést nehezebbnek találjuk, mint az alkalmazástervezést, mivel az elemkészleteknek sok alkalmazásban kell tudniuk működni, hogy használhatók legyenek. Továbbá az elemkészlet írója nincs abban a helyzetben, hogy tudná, melyek lesznek a felhasználó alkalmazások, vagy mik lesznek a különleges igényeik. Ez még fontosabbá teszi, hogy elkerüljük a feltevéseket és függőségeket, amik korlátozzák az elemkészlet rugalmasságát, és ebből következően az alkalmazás hatékonyságát.

Keretrendszerek

A **keretrendszer** (framework) együttműködő osztályok összessége egy bizonyos programtípus számára [Deu89, JF88], amelyek egy újrahasznosítható szerkezetben egyesülnek. Például egy keretrendszer irányulhat arra, hogy grafikus szerkesztőket építsünk belőle olyan különböző területekre, mint például a művészi rajzolás, a zeneszerkesztés, és a gépészeti CAD [VL90, Joh92]. Egy másik keretrendszer segíthet különböző programozási nyelvek és célgépek fordítójának megépítésében [JML92]. Megint egy másik pénzügyi modellező alkalmazások készítését segítheti [BE93]. A keretrendszert úgy kell testreszabnunk egy adott alkalmazáshoz, hogy alkalmazásfüggő alosztályokat származtatunk a keretrendszer elvont osztályaiból.

A keretrendszer meghatározza az alkalmazás felépítését. Meghatározza az általános szerkezetet, annak osztályokra és objektumokra bontását, illetve a kulcskötelességeket, vagyis hogy hogyan működnek együtt az objektumok és osztályok, valamint a vezérlés haladási irányát. A keretrendszer mindeme tervezési paramétereket előre megadja, hogy nekünk, az alkalmazás tervezőinek, illetve megvalósítóinak csak az alkalmazás lényegére kelljen figyelnünk. A keretrendszer azokat a tervezési döntéseket foglalja magába, amelyek az adott felhasználási területen általánosak. A keretrendszerek tehát a *tervezési újrahasznosítást* részesítik előnyben a kód újrahasznosításával szemben, bár egy adott keretrendszer konkrét alosztályokat is biztosít számunkra, amelyeket azonnal használhatunk is.

Az ilyen szintű újrahasznosítás a vezérlés irányának megfordulásához vezet az alkalmazás és a program között, amelyen alapul. Ha elemkészletet használunk (vagy éppen egy hagyományos alprogram-könyvtárat), megírjuk az alkalmazás vázát, és meghívjuk a kódot,

amit fel akarunk használni. Ha keretrendszert használunk, a vázat használjuk fel, és megírjuk a kódot, amit hív. Konkrét nevekkel és hívási módokkal rendelkező műveleteket kell írnunk, de ez lecsökkenti a tervezési döntések számát, amiket meg kell hoznunk.

Eredményképp nem csak gyorsabban építhetjük fel alkalmazásainkat, de azoknak hasonló lesz a szerkezetük is, egyszerűbb lesz a karbantartásuk, és következetesebbnek tűnnek a felhasználóknak. Másrészről veszítünk a kreatív szabadságból, hiszen már sok tervezői döntést meghoztak helyettünk.

Ha az alkalmazásokat nehéz megtervezni és az elemkészleteket még nehezebb, akkor a keretrendszereket a legnehezebb. A keretrendszer-tervező arra törekszik, hogy az adott szerkezet a terület minden alkalmazásának esetében működőképes legyen. Minden jelentős változtatás a keretrendszer szerkezetében jelentősen csökkenti a keret előnyeit, mivel annak lényege éppen az a felépítés, amit az alkalmazások számára meghatároz. Ezért életbevágó a keretrendszerek olyan rugalmasságúra és bővíthetőre írása, amennyire csak lehet.

Továbbá, mivel az alkalmazások szerkezete annyira függ a keretrendszertől, azok különösen érzékenyek a keretrendszer felületében végrehajtott változásokra. Ahogy egy keretrendszer fejlődik, az alkalmazásoknak is követniük kell. Ez teszi a laza csatolást a legfontosabbá; máskülönben a keretrendszer kis változtatása is jelentős kellemetlen utóhatásokat váltana ki.

Az imént említett tervezési kérdések nagyon fontosak a keretrendszerek tervezésében. Egy tervezési mintán keresztül dolgozó keretrendszer sokkal valószínűbb, hogy magas szintű tervezést és kód-újrahasznosítást ér el, mint egy olyan, amelyik nem. A kész keretrendszerekben általában több tervezési minta is megtestesül. A minták segítségével a keretrendszer felépítése számos különböző alkalmazáshoz megfelel újratervezés nélkül.

Még jobb, ha a keretrendszert a benne foglalt tervezési mintákkal együtt dokumentálják [BJ94]. Azok, akik ismerik a mintákat, így gyorsabban nyernek betekintést a keretrendszerbe, de azok is nyerhetnek a keretrendszer dokumentációjához csatolt szerkezetből, akik nem ismerik a mintákat. A dokumentáció elkészítése mindenféle program esetében fontos, de különösen a keretrendszereknél. A keretrendszerek működésének megértése gyakran igen nehéz, de e tudás elsajátítása szükséges ahhoz, hogy hasznosíthassuk őket. Bár a tervezési minták nem mentesítenek eme erőfeszítéstől, megkönnyíthetik a dolgunkat azzal, hogy egyértelműbbé teszik a keretrendszer szerkezetének elemeit.

Mivel a minták és keretrendszerek között elég sok a közös vonás, sokan elgondolkodhatnak azon, hogy mik is a különbségek, ha vannak ilyenek egyáltalán. Nos, három fő vonásban különböznek:

1. *A tervezési minták elvontabbak a keretrendszereknél.* A keretrendszerek megtestesülhetnek programkód formájában, de a mintáknál csak a *példa* jelenhet meg kódként. A keretrendszerek erőssége, hogy leírhatók programozási nyelvek segítségé-

vel, és nem csak tanulhatók, hanem közvetlenül futtathatók és felhasználhatók. Ezzel szemben az ebben a könyvben szereplő tervezési mintákat mindig alkalmaznunk kell felhasználásukhoz, emellett a tervezési minták elmagyarázzák a terv céljait, előnyeit és következményeit is.

2. *A tervezési minták kisebb szerkezeti elemek a keretrendszereknél.* Egy keretrendszer általában több tervezési mintát tartalmaz, de ez fordítva soha nem igaz.
3. *A tervezési minták kevésbé specializáltak a keretrendszereknél.* A keretrendszerek mindig egy bizonyos felhasználási területre vonatkoznak. Egy grafikus szerkesztő keretrendszerét fel lehet használni egy gyár modellezésében, de ettől még nem keverhető össze egy szimulációs keretrendszerrel. Ezzel szemben az itt leírt tervezési mintákat szinte bármilyen alkalmazásban használhatjuk. Bár természetesen lehetne a mieinknél szakosodottabb tervezési mintákat készíteni (mondjuk elosztott rendszerekhez vagy párhuzamos programozáshoz), még ezek sem jelölnék ki úgy az alkalmazás felépítését, mint egy keretrendszer.

A keretrendszerek egyre gyakoribbá és fontosabbá válnak. Az objektumközpontú rendszerek így érik el a legnagyobb felhasználhatóságot. A nagyobb objektumközpontú alkalmazások keretrendszerek egymással együttműködő rétegeiből állnak. Az alkalmazás szerkezetének és kódjának nagy része keretrendszerekből származik, vagy legalábbis ezek erősen hatnak rá.

1.7 Hogyan válasszunk tervezési mintát?

A katalógusban található, húsznál is több választható tervezési mintából nem könnyű megtalálni, melyikre van éppen szükségünk egy adott probléma megoldásához, különösen, ha a gyűjtemény új és ismeretlen számunkra. Ezért az alábbiakban néhány tanácsot adunk a megfelelő tervezési minta kiválasztásához:

- *Vegyük figyelembe, hogy a tervezési minták hogyan oldják meg a tervezési problémákat.* Az 1.6-os rész azt tárgyalja, hogy a tervezési minták hogyan lehetnek segítségünkre a megfelelő objektumok megtalálásában, az objektum-részletezettség és az objektumfelületek meghatározásában és más módszerekben, amelyekkel a tervezési minták megoldhatják gondjainkat. Ezek áttekintése segíthet a helyes minta megtalálásában.
- *Nézzük át a célról szóló részeket.* Az 1.4 rész felsorolja a könyvben szereplő minták céljait. Olvassuk át ezt a részt, azok után a célok kutatva, amelyek kapcsolódnak az adott problémához. Az 1.1-es táblázatban található osztályozó rendszer használatával leszűkíthetjük a keresést.
- *Tanulmányozzuk a minták kapcsolatát.* Az 1.1-es ábra grafikusan mutatja be a tervezési minták közti kapcsolatokat. Ezen kapcsolatok tanulmányozása segíthet a jó minta vagy mintacsoport megtalálásában.
- *Tanulmányozzuk a hasonló célú mintákat.* A katalógus három fejezetből áll: az első a létrehozási mintákról szól, a második a szerkezeti mintákról, a harmadik pedig a viselkedési mintákról. Minden fejezet a mintákat bemutató megjegyzésekkel indít, és egy olyan résszel zárul, ami összehasonlítja a mintákat. Ezek a részek a hasonló célú minták közti hasonlatosságokba és különbségekbe engednek betekintést.

- *Vizsgáljuk meg az újratervezés okait.* Vizsgáljuk meg ismét az újratervezés okait a Változásra tervezve részben, hogy lássuk, melyik vonatkozik ránk. Aztán nézzük át a mintákat, amelyek segíthetnek elkerülni az újratervezést.
- *Gondoljuk át, mit tegyünk változtathatóvá rendszerünkben.* Ez a megközelítés az újratervezés okaira való összpontosítás ellentéte. Ahelyett, hogy azt néznénk, mi miatt *kellhet* megváltoztatnunk a tervet, azon gondolkodunk el, hogy mit akarunk majd újratervezés nélkül *módosíthatóvá* tenni. Itt a változó elemek egységbe zárására összpontosítunk, ami számos tervezési minta témája. Az 1.2-es táblázat azokat az elemeket sorolja fel, amelyeket az egyes tervezési minták használatával függetlenül, vagyis újratervezés nélkül megváltoztathatunk.

Cél	Tervezési minta	Változtatható elemek
Létrehozási	Elvont gyár	Leszármazott objektumok családjai
	Építő	Hogyan készül az összetett objektum
	Gyártófüggvény	Egy példányobjektum alosztálya
	Prototípus	Egy példányobjektum osztálya
	Egyke	Egy osztály egyetlen példánya
Szerkezeti	Illesztő	Felület egy objektumhoz
	Híd	Egy objektum megvalósítása
	Összetétel	Egy objektum szerkezete és összetétele
	Díszítő	Egy objektum kötelességei leszármaztatás nélkül
	Homlokzat	Felület egy alrendszerhez
	Pehelysúlyú	Objektumok tárolásának költsége
	Helyettes	Hogyan érünk el egy objektumot; az objektum helyzete
Viselkedési	Felelősséglánc	Az objektum, ami a kérelmeket teljesíti
	Parancs	Mikor és hogyan teljesül egy kérelem
	Értelmező	Egy nyelv szabályai és értelmezése
	Bejáró	Hogyan érjük el és járjuk be egy aggregátum elemeit
	Közvetítő	Mely objektumok hatnak egymásra, és hogyan
	Emlékeztető	Milyen privát információ tárolódik az objektumon kívül, és mikor
	Megfigyelő	Számos más objektumtól függő objektum; hogyan maradnak a függő objektumok naprakészek
	Állapot	Egy objektum állapotai
	Stratégia	Egy algoritmus
	Sablonfüggvény	Egy algoritmus lépései
	Látogató	Olyan műveletek, amelyek alkalmazhatók objektum(ok)ra az osztályuk megváltoztatása nélkül

1.2 táblázat

A tervezési minták által megengedett változtatható elemek.

1.8 Hogyan használjuk a tervezési mintákat?

Ha már kiválasztottuk a tervezési mintát, hogyan használjuk? Íme egy útmutató, amely lépésről lépésre megmutatja, hogyan használhatjuk fel hatékonyan a mintákat:

1. *Olvassuk át a mintát, hogy legyen róla elképzelésünk.* Legyünk különös figyelemmel az Alkalmazhatóság és a Következmények részekre, hogy biztosak lehessünk benne, hogy ez a megfelelő minta a problémánk megoldására.
2. *Térjünk vissza a Szerkezet, a Résztvevők és az Együttműködés fejezetekre.* Győződjünk meg róla, hogy értjük a mintában szereplő osztályokat és objektumokat, és hogy azok hogyan kapcsolódnak egymáshoz.
3. *Nézzük át a Példakód részt, hogy lássunk egy konkrét példát a minta kódba ágyazására.* A kód tanulmányozása segít a minta megvalósításának megtanulásában.
4. *Válasszunk olyan neveket a minta résztvevőinek, amelyek értelmesek lesznek az alkalmazott környezetben.* A tervezési mintákban található nevek általában túl elvontak ahhoz, hogy egy alkalmazásban használhatók legyenek. Mindazonáltal hasznos a résztvevő nevének belefoglalása abba a névbe, ami majd az alkalmazásban megjelenik. Ennek segítségével nyilvánvalóbbá válik a minta alkalmazása a megvalósításban. Például ha a Stratégia mintát használjuk egy szövegszerkesztő algoritmushoz, akkor olyan osztályaink lehetnek, mint például az EgyszerűElrendezésStratégia vagy a TeXElrendezésStratégia.
5. *Határozzuk meg az osztályokat.* Adjuk meg a felületüket, és hozzuk létre az öröklődési kapcsolataikat, majd határozzunk meg olyan példányváltozókat, amik mutatják az adat- és objektumhivatkozásokat. Azonosítsuk az alkalmazás azon létező osztályait, amelyekre hatással lesz a minta, és ennek megfelelően módosítsuk azokat.
6. *Adjunk az alkalmazásra jellemző neveket a mintában szereplő műveleteknek.* Itt a nevek újra csak az alkalmazástól függnek. Használjuk a műveletekhez rendelt feladatokat és együttműködési kapcsolatokat útmutatóként. Továbbá legyünk következetesek az elnevezési rendszert illetően. Például egy gyártófüggvény nevében használjuk mindig a „Létrehoz-" (vagy Create) előtagot.
7. *Valósítsuk meg a mintában található feladatokat és együttműködéseket ellátó műveleteket.* A Megvalósítás rész tippeket ad a megvalósításra, de a Példakód részben leírt példák is segíthetnek.

Ezek persze csak tanácsok a kezdéshez. Idővel mindenkinek kifejlődik a saját munkamódszere a tervezési minták használatára.

A tervezési minták használatának tárgyalása nem lenne teljes anélkül, hogy arról is írnánk, hogyan *ne* használjuk őket. Nos, ne használjuk őket megfontolatlanul. Gyakran úgy érnek el rugalmasságot és változtathatóságot, hogy további közvetítő szinteket vezetnek be, ám ez bonyolíthatja a szerkezetet, illetve csökkentheti a hatékonyságot. Egy tervezési mintát csak akkor alkalmazzunk, ha a rugalmasság, amit nyújt, valóban szükséges. A Következmények rész segíthet a legtöbbet egy minta előnyeinek és hátrányainak kiszámításában.

2

Esettanulmány: szövegszerkesztő tervezése

Ebben a fejezetben egy grafikus (WYSIWYG, „Azt kapod, amit látsz") szövegszerkesztő felépítését vizsgáljuk meg, amelynek neve **Lexi**[1]. Elemzésével látni fogjuk, milyen megoldásokat adnak a tervezési minták az ilyen és ehhez hasonló programok készítése során felmerülő tervezési problémákra. A fejezet végére nyolc minta gyakorlati használatában szerzünk jártasságot.

A 2.1 ábra a Lexi felhasználói felületét mutatja. Az ablak közepének nagy részét egy téglalap alakú terület foglalja el, amelyben a dokumentum grafikus megjelenítése kap helyet. A dokumentumban szöveges részek és képek vegyesen lehetnek, különféle formázási beállítások mellett. A dokumentumterületet a szokásos lenyitható menük és gördítősávok szegélyezik, illetve oldalikonok, amelyek azt szolgálják, hogy a dokumentum adott oldalára ugorhassunk.

2.1 Tervezési problémák

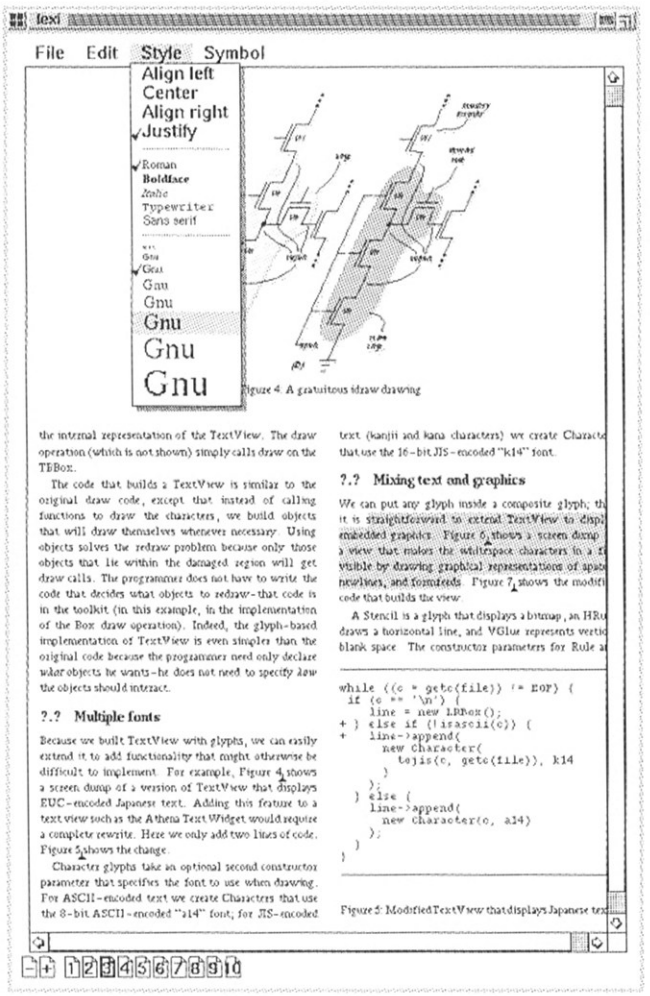
A Lexi szerkezetét hétféle szempontból vizsgáljuk:

1. *Dokumentumszerkezet.* A dokumentum belső ábrázolásának megválasztása a program szerkezetének szinte valamennyi elemét érinti, hiszen minden szerkesztési, formázási, megjelenítési és szövegelemzési művelet ezen ábrázolás bejárását igényli. Az, ahogyan a dokumentumban levő adatokat elrendezzük, az alkalmazás többi részének megtervezésére is kihat.
2. *Formázás.* Hogyan rendezi sorokba és hasábokba a Lexi a szöveget és képeket? Milyen objektumok felelnek a különböző formázási módok érvényre juttatásáért? Hogyan kapcsolódnak ezek a formázási módok a dokumentum belső ábrázolásához?
3. *A felhasználói felület finomítása.* A Lexi felhasználói felülete gördítősávokat, szegélyeket és árnyékolásokat tartalmaz, amelyek a grafikus dokumentummegjelenítést

[1] A Lexi szerkezete a Doc-on alapul, amely egy szövegszerkesztő alkalmazás, amelyet Calder [CL92] készített.

teszik szebbé. Ezek a díszítő elemek valószínűleg változni fognak, ahogy a felhasználói felületet továbbfejlesztjük, ezért fontos, hogy anélkül legyenek könnyen eltávolíthatók, illetve kiegészíthetők, hogy ez a program többi részét érintené.

4. *Több megjelenítési szabvány támogatása.* Célunk, hogy a Lexi könnyen alkalmazkodjon az olyan különböző megjelenítési szabványokhoz, mint a Motif vagy a Presentation Manager (PM), és ehhez ne legyen szükség nagyobb módosításokra.
5. *Több ablakkezelő rendszer támogatása.* A különféle megjelenítési szabványok általában többféle ablakkezelő rendszeren működnek. A Lexit annyira függetleníteni kell az ablakrendszerektől, amennyire csak lehetséges.
6. *Felhasználói műveletek.* A felhasználók különféle grafikus felületi elemeken – gombokon, lenyíló menükön stb. – keresztül vezérelhetik a programot, melyek működését az alkalmazás különböző részein szétszórt objektumok biztosítják. A kihívást az jelenti, hogy egységesen kezeljük ezeket az objektumokat, illetve egységesen vonhassuk vissza az általuk végrehajtott műveleteket.
7. *Helyesírás-ellenőrzés és elválasztás.* Hogyan támogatja a Lexi az olyan elemző műveleteket, mint a helytelenül írt szavak megkeresése vagy az elválasztási pontok meghatározása? Hogyan csökkenthetjük a lehető legkisebbre azon osztályok számát, amelyeket módosítanunk kell, ha új elemző művelettel szeretnénk kiegészíteni a programot?



2.1 ábra

A Lexi felhasználói felülete.

A fenti tervezési problémákat a következő részekben alaposabban megvizsgáljuk. Mindegyikhez célokat kapcsolunk, illetve megszorításokat, amelyek az említett célok elérésének módjára vonatkoznak. Először részletesen elmagyarázzuk a célokat és megszorításokat, és csak ezután térünk rá a megoldási javaslatokra. Egy-egy probléma és megoldása egy vagy több tervezési mintát mutat be; a problémák felvázolása minden esetben a vonatkozó tervezési minta rövid bevezetésével zárul.

2.2 Dokumentumszerkezet

A dokumentum végső soron nem más, mint alapvető grafikai elemek – karakterek, vonalak, sokszögek és más alakzatok – adott elrendezése. Ezek az elemek határozzák meg a dokumentum teljes információtartalmát, de a dokumentum készítője nem grafikai, hanem a fizikai szerkezethez kapcsolódó elemekként – sorok, hasábok, ábrák, táblázatok és más szerkezetek – látja azokat[2], amelyek maguk is kisebb hasonló elemekből állnak.

A Lexi felhasználói felületét úgy kell elkészítenünk, hogy a programot használók képesek legyenek ezeket az elemeket közvetlenül elérni. Nem árt például, ha a felhasználó egy diagramot egységként és nem önálló grafikai alapelemek (primitívek) halmazaként kezelhet, vagy ha egy táblázatra mint egészre és nem mint képekkel kevert szövegek alaktalan masszájára hivatkozhat. Ettől lesz a felület egyszerű és könnyen használható. Ahhoz, hogy ezt a Lexi esetében is elérhessük, olyan belső ábrázolást választunk, ami illeszkedik a dokumentum fizikai szerkezetéhez.

A belső ábrázolásnak konkrétan az alábbiakat kell támogatnia:

- A dokumentum fizikai szerkezetének megőrzése, vagyis a szövegek és képek sorokba, hasábokba, táblázatokba stb. rendezése.
- A dokumentum grafikus képének előállítása és megjelenítése.
- A megjelenített kép pontjainak megfeleltetése a belső ábrázolás elemeinek. Ez teszi lehetővé, hogy a Lexi meghatározza, mire is hivatkozik a felhasználó, amikor a dokumentum grafikus megjelenítésén belül valahová kattint.

E célok elérésében bizonyos megkötéseket kell tennünk. Először is, a képeket és szövegeket egységesen kell kezelnünk. Az alkalmazás meg kell, hogy engedje a felhasználónak, hogy szöveget illesszen be képbe és fordítva. El kell kerülnünk, hogy a képeket a szöveg különleges eseteként kezeljük, vagy a szövegeket különleges képként, másképp felesleges szerkesztő és formázó műveletekkel fogunk rendelkezni. Egyetlen művelethalmaz elegendő kell legyen mind a szövegekhez, mind a képekhez.

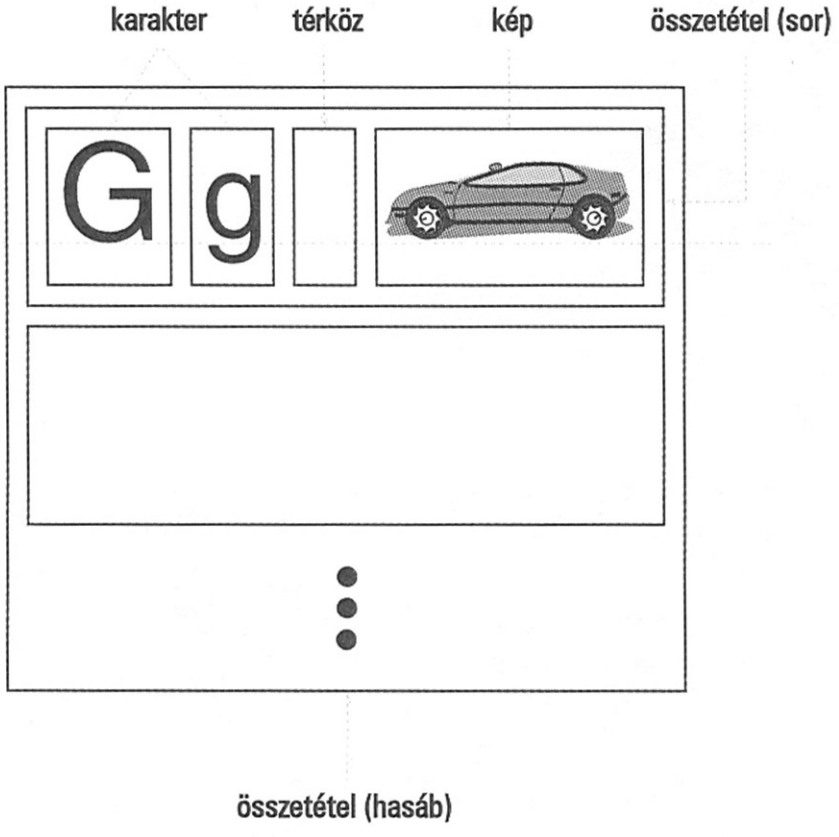
[2] A dokumentumok készítői emellett gyakran a logikai szerkezet alapján gondolnak a dokumentumra, vagyis mint mondatok, bekezdések, fejezetek és alfejezetek összességére. A példa egyszerűsítése érdekében az általunk használt belső ábrázolás kifejezetten nem tárol majd információt a logikai szerkezetről, bár a leírt tervezési megoldás az ilyen információk ábrázolására is tökéletesen alkalmas.

Másodszor, a belső ábrázolás megvalósításában az önálló elemek és elemcsoportok között nem szabad különbséget tennünk. A Lexi képes kell legyen arra, hogy az egyszerű, illetve összetett elemeket egységesen kezelje, és így támogassa a tetszőlegesen összetett dokumentumokat. „A második hasáb ötödik sorában a tizedik elem" lehessen egyetlen karakter, de számos alelemmel rendelkező bonyolult diagram is. Amíg az elem megrajzolhatja magát és meghatározhatjuk kiterjedését, bonyolultsága nem lesz hatással arra, hogy az oldalon hol és hogyan jelenik meg.

A második megszorítással szemben azonban a szöveget abban az esetben igenis elemeznünk kell, ha helyesírási hibákat és elválasztási pontokat keresünk. Sokszor nem számít, hogy egy sor egy eleme egyszerű vagy összetett objektum-e, máskor azonban az elemzés az adott objektum típusától függ. Nincs értelme például egy sokszög helyesírását ellenőrizni vagy elválasztani azt. A program belső szerkezetének ezeket és más ellentmondó megszorításokat is figyelembe kell vennie.

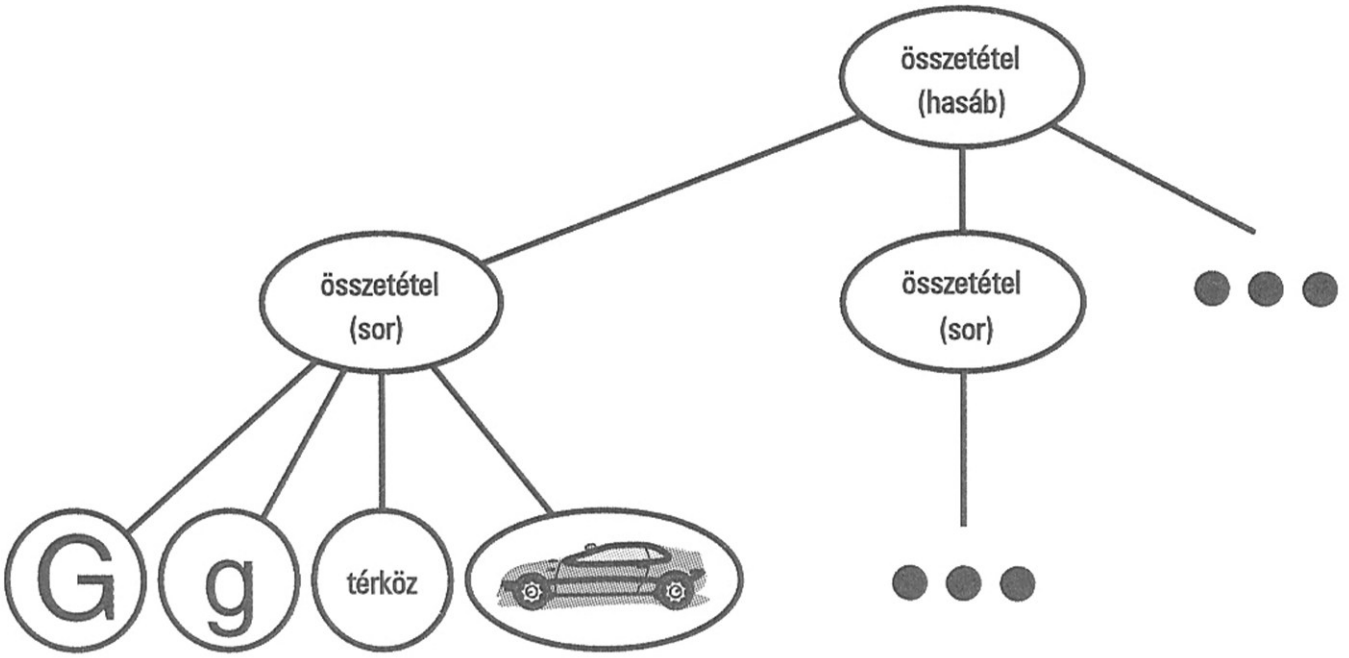
Önhívó felépítés

A hierarchikus felépítésű információk ábrázolásának egyik elterjedt módja az **önhívó felépítés** (rekurzív kompozíció), melynek során egyszerű elemekből egyre összetettebbeket építünk fel. E módszer révén a dokumentum egyszerű grafikai elemekből állítható össze. Első lépésként karakterek és képek halmazát rakhatjuk sorba balról jobbra, hogy kialakítsunk egy sort. Aztán több sort hasábba rendezhetünk, a hasábokból oldalakat építhetünk és így tovább (lásd a 2.2 ábrát).



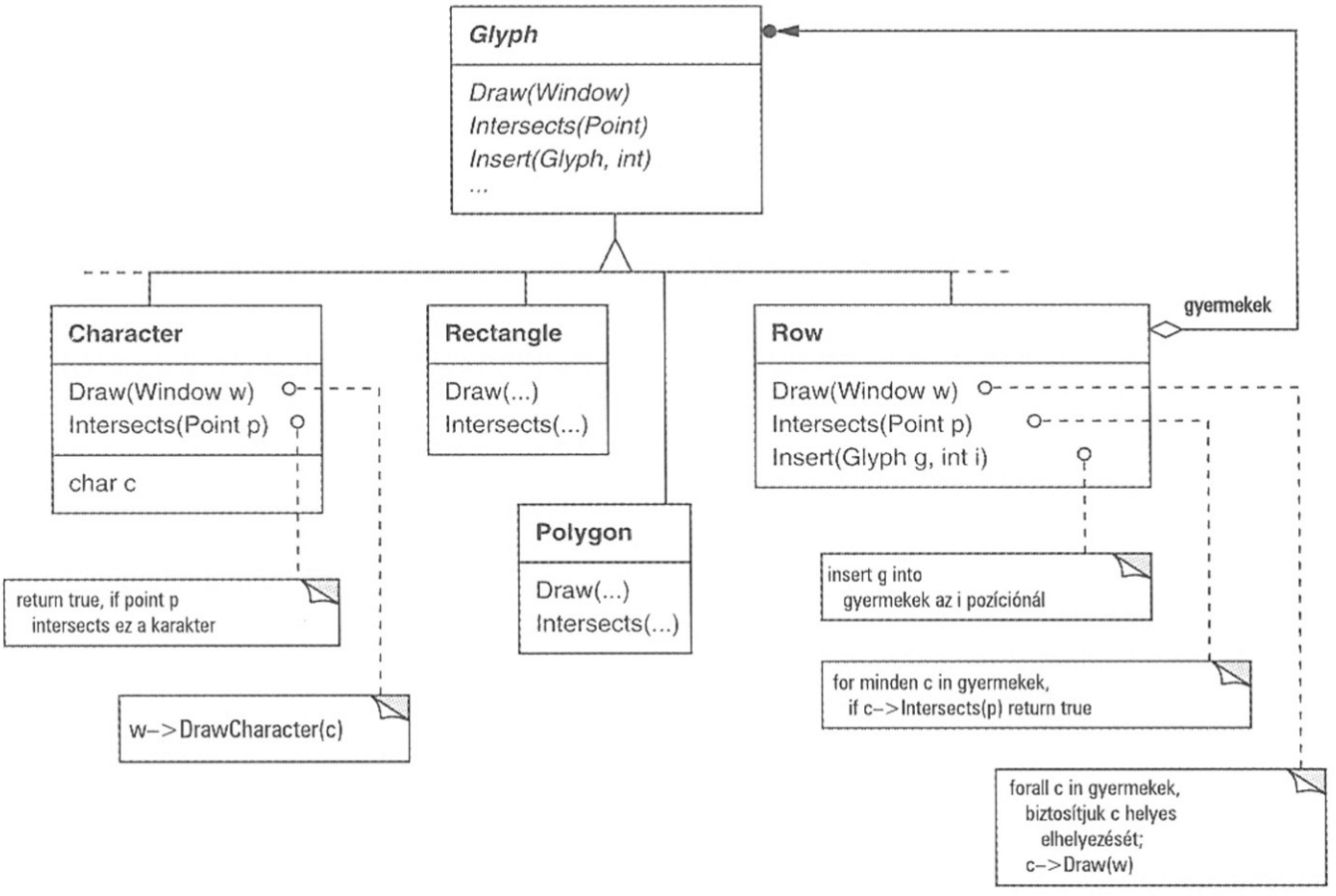
2.2 ábra

Képek és szöveg önhívó felépítése.



2.3 ábra

Képek és szöveg önhívó felépítésének objektumszerkezete.



2.4 ábra

Részleges képjel-osztályhiererchia.

A fizikai szerkezetet úgy ábrázolhatjuk, hogy minden lényeges eleméhez egy objektumot rendelünk. Tehát nem csak a látható elemekhez, amilyenek a karakterek és képek, hanem a láthatatlan szerkezeti elemekhez, a sorokhoz és hasábokhoz is. Az eredményt a 2.3 ábrán látható objektumszerkezet mutatja.

Azzal, hogy a dokumentum minden karakteréhez és grafikus eleméhez egy-egy objektumot társítunk, a Lexi szerkezetének minden szintjén biztosítjuk a rugalmasságot. A szövegeket és képeket megrajzolásukat, formázásukat és egymásba való ágyazottságukat figyelembe véve, egységesen kezelhetjük. A programot új karakterkészletek támogatásával is kiegészíthetjük, anélkül, hogy ez érintené a többi szolgáltatást. A Lexi objektumszerkezete a dokumentum fizikai szerkezetét modellezi.

Ez a megközelítés két dolgot von maga után. Az első nyilvánvaló: az objektumhoz megfelelő osztályokat kell létrehoznunk. A második, ami már nem biztos, hogy annyira magától értetődő, hogy ezek az osztályok összeegyeztethető felületeket igényelnek, hiszen az objektumokat egységesen szeretnénk kezelni. Összeegyeztethető felületeket pedig egy olyan nyelvben, mint a C++, úgy készíthetünk, ha az osztályokat öröklés útján származtatjuk.

Képjelek

A dokumentumszerkezetben megjelenő valamennyi objektum közös elvont osztálya a **Képjel** (Glyph[3]) lesz. Alosztályai mind egyszerű grafikai elemeket (karakterek, képek), mind szerkezeti elemeket (sorok, hasábok) leírnak majd. A 2.4 ábra a Képjel (az ábrán angolul Glyph) osztályhierarchia egy jellemző részletét mutatja, míg a 2.1 táblázatban a C++ jelölését használva részletesebben is bemutatjuk az alapvető képjelfelületet[4].

Felelősségi kör	Műveletek
Megjelenés	`virtual void Draw(Window*)` `virtual void Bounds(Rect&)`
Találat-érzékelés	`virtual bool Intersects(const Point&)`

Szerkezet	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

2.1 táblázat

Alapvető képjelfelület.

[3] A „glyph" kifejezést ebben az összefüggésben először Calder [CL90] használta; a mai szövegszerkesztők többsége nem használ minden karakterre külön objektumot, valószínűleg hatékonysági okokból. Calder tanulmányában [Cal93] bizonyította, hogy a megoldás működőképes. A mi képjeleink az övéinél kevésbé kifinomultak, mert az egyszerűség kedvéért szigorú hierarchiába rendeztük őket. Calder glyph-jei a tárolási költségek csökkentése végett megoszthatók, így irányított körmentes gráf szerkezetet formálnak. A Pehelysúlyú minta használatával hasonló eredményt érhetünk el, de ezt meghagyjuk gyakorló feladatnak.

[4] Az itt leírt felületet szándékosan egyszerűsítettük le a végletekig, hogy magyarázatunk követhető legyen. Egy teljes felület az olyan grafikus jellemzők kezelésére szolgáló műveleteket is tartalmazná, mint a szín vagy a betűtípus, illetve képes lenne koordináta-átalakításra, és magába foglalna bonyolultabb gyermekkezelő eljárásokat is.

A képjeleknek három alapvető feladatuk van. 1. Tudniuk kell, hogyan rajzolják meg önmagukat. 2. Tudniuk kell, mennyi helyet foglalnak. 3. Ismerniük kell a szülőjüket és gyermekeiket.

A Glyph alosztályai felülbírálják a `Draw` (Rajzol) műveletet, így rajzolják ki magukat az ablak felületére; a `Draw` hívása során egy `Window` (Ablak) objektumra való hivatkozást kapnak. A **Window** osztály határozza meg azokat a grafikai műveleteket, amelyek a szövegek és más alapvető alakzatok képernyőre rajzolásához szükségesek. A Glyph **Rectangle** (Téglalap) alosztálya például a következőképpen írhatja felül a `Draw` műveletet:

```
void Rectangle::Draw (Window* w) {
   w->DrawRect(_x0, _y0, _x1, _y1);
}
```

A fenti kódban az `_x0`, `_y0`, `_x1` és `_y1` a `Rectangle` adattagjai, amelyek a téglalap két ellentétes sarkának koordinátáit írják le, a `DrawRect` pedig a Window-nak azon művelete, amely a téglalapnak a képernyőn való megjelenítéséért felelős.

A szülő képjeleknek gyakran tudniuk kell, mekkora helyet foglal egy adott gyermekük, hogy azt más képjelekkel együtt úgy rendezhessék sorba, hogy egyik se fedje a másikat. (Erre mutat példát a 2.2 ábra.) A képjel által elfoglalt téglalap alakú területet a `Bounds` (Határok) művelet adja vissza, azon legkisebb befoglaló négyszög ellentétes sarkainak koordinátáival, amelyekbe a képjel még belefér. A Glyph alosztályai ezt a műveletet bírálják felül, hogy meghatározhassák a saját rajzolásukhoz szükséges területet.

Az `Intersects` (Metszi) művelet azt adja meg, hogy egy adott pont metszi-e a képjelet. Amikor a felhasználó valahol a dokumentumban kattint, a Lexi meghívja ezt a műveletet, hogy meghatározza, melyik képjelen, illetve annak melyik részén történt az egérkattintás. A Rectangle osztály ennek felülírásával számítja ki a téglalap és az adott pont metszéspontját.

Miután a képjeleknek gyermekeik is lehetnek, ezek hozzáadásához, elvételéhez, illetve eléréséhez közös felületre van szükségünk. A Row (Sor) gyermekei például azok a képjelek, amelyek a sorban találhatók. Az `Insert` (Beilleszt) művelet egy képjelet illeszt be egy egész sorszám által meghatározott helyen[5], míg a `Remove` (Eltávolít) eltávolítja a megadott képjelet, ha az valóban gyermek.

A `Child` (Gyermek) művelet a megadott sorszámon található gyermeket adja vissza (ha van ott olyan). A Row-hoz hasonló, gyermekekkel rendelkezni képes képjelek belsőleg kell, hogy használják a `Child` műveletet, vagyis nem jó, ha a gyermekek adatszerkezetét közvetlenül érik el. Így később nem kell módosítani a `Draw`-hoz hasonló gyermekbejáró műveleteket, ha az adatszerkezetet mondjuk tömbről láncolt listára változtatjuk. Ugyanígy

[5] Az egész sorszámok használata talán nem a legjobb módja annak, hogy meghatározzuk egy képjel gyermekeit, de ez a képjel által használt adatszerkezettől függ. Ha gyermekeit láncolt listában tárolja, egy, a listát címző mutató hatékonyabb lenne. A sorszámozás (indexelés) problémájára jobb megoldást is látunk majd a 2.8 részben, ahol a dokumentum elemzését tárgyaljuk.

a `Parent` (Szülő) a képjel szülőjéhez biztosít szabványos felületet. A Lexiben a képjelek hivatkozást tárolnak a szülőjükre, `Parent` műveletük pedig egyszerűen ezt a hivatkozást adja vissza.

Az Összetétel tervezési minta

Az önhívó összetétel módszere nem csak a dokumentumok esetében működik; bármilyen bonyolult, hierarchikus felépítményt ábrázolhatunk vele. A módszer objektumközpontú fogalmakkal vázolt lényegét az Összetétel (Composite) minta ragadja meg – tulajdonképpen itt is az ideje, hogy közelebbről megvizsgáljuk e mintát, szükség esetén visszautalva az itt bemutatott forgatókönyvre.

2.3 Formázás

Már eldöntöttük, hogyan *ábrázoljuk* a dokumentum fizikai szerkezetét; így a következő lépés, hogy kitaláljuk, hogyan építhetünk fel egy *konkrét* szerkezetet, ami megfelel egy helyesen formázott dokumentumnak. Az ábrázolás és formázás között különbséget kell tennünk: attól, hogy képesek vagyunk ábrázolni a dokumentum fizikai szerkezetét, még nem tudjuk, hogyan készíthetünk el egy adott szerkezetet. Ez nagyrészt a Lexi feladata lesz: a szöveget sorokra kell tördelnie, a sorokat hasábokba és így tovább, figyelembe véve a felhasználó kívánságait. A felhasználó például változtatható margószélességet, behúzást és térközöket, egyes és kettes sortávot, esetleg más formázási lehetőségeket szeretne[6], amelyeket a Lexi formázó algoritmusának mind figyelembe kell majd vennie.

„Formázás" alatt mi most csupán azt fogjuk érteni, hogy képjelek csoportját sorokba rendezzük, vagyis a „formázás" és „sortörés" kifejezéseket felcserélhetjük. A tárgyalt eljárás sorok hasábokba, illetve hasábok oldalakra rendezésére is ugyanúgy alkalmas.

Felelősségi kör	Műveletek
Mit formázunk?	`void SetComposition(Composition*)`
Mikor formázunk?	`virtual void Compose()`

2.2 táblázat

Egyszerű összeállító felület.

[6] A felhasználót még ennél is jobban érdekelheti a dokumentum *logikai* szerkezete, vagyis a mondatok, bekezdések, fejezetek, alfejezetek stb. A *fizikai* szerkezet ezzel összehasonlítva kevésbé lényeges: a legtöbb embert nem érdekli, hogy egy bekezdésben hová esnek a sortörések, amíg a bekezdés formázása megfelelő, és ugyanez érvényes a hasábok és oldalak formázására is. Tehát a felhasználó csupán magas szintű megkötéseket tesz a fizikai szerkezetet illetően, és a Lexire bízza, hogyan elégíti ki az igényeket.

A formázó algoritmus egységbe zárása

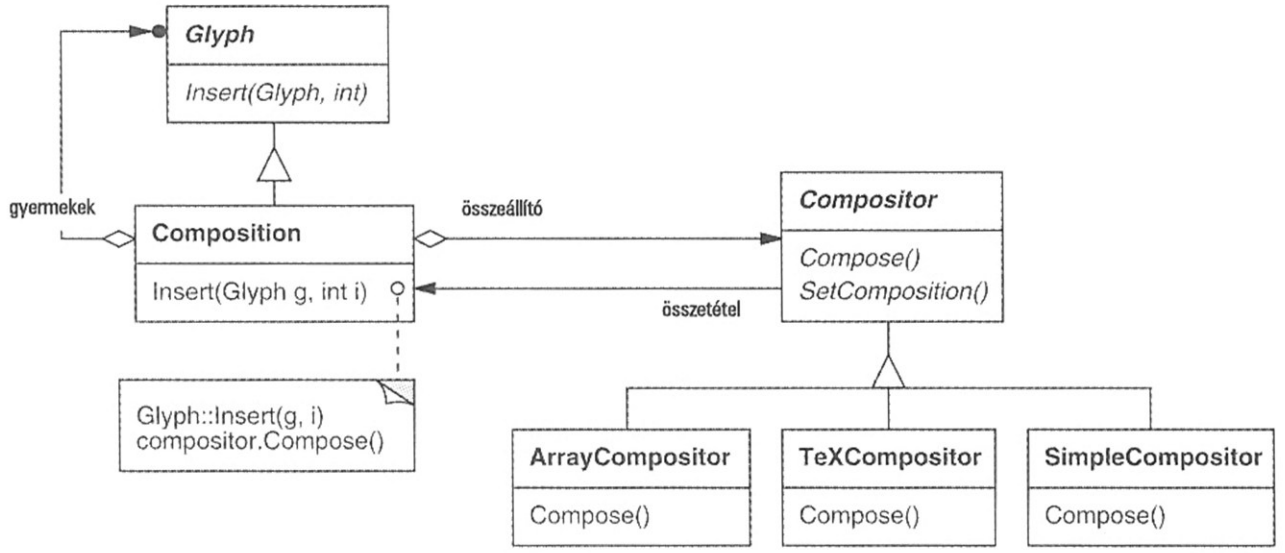
A formázási folyamatot, megszorításaival és más részleteivel együtt, nehéz automatizálni. A problémára számos megközelítés létezik, és a programozók különböző hatékonyságú formázó algoritmusok egész sorát dolgozták ki. Miután a Lexi vizuális szerkesztő, az egyik lényeges kérdés, amelyben kompromisszumot kell kötnünk, hogy melyik fontosabb: a formázás minősége vagy a sebessége? Általában azt várjuk el, hogy a szerkesztő viszonylag gyorsan hajtsa végre utasításainkat, de anélkül, hogy ez a dokumentum külalakján különösebben rontana. Ennek megoldása számos tényező függvénye, amelyek közül nem mindegyik befolyásolható fordítási időben. Előfordulhat, hogy a felhasználó a szebb megjelenítést részesíti előnyben, és ennek érdekében hajlandó elnézni a némileg lassabb működést, ami esetleg teljesen más algoritmus használatát igényli, mint az, amit éppen alkalmazunk. Egy másik, a megvalósítás módjára nagyobb hatást gyakorló döntés lehet, hogy a formázás sebességét vagy a tárigényt tartjuk inkább szem előtt. A formázás végrehajtására fordított idő csökkenthető, ha több információt tárolunk átmenetileg.

A formázó algoritmusok általában bonyolultak, ezért kívánatos, hogy minél jobban függetlenítsük azokat a dokumentum szerkezetétől. Ideális esetben anélkül hozhatunk létre új képjeleket, hogy a formázó algoritmusra figyelnünk kellene, és fordítva, egy új formázó algoritmus hozzáadása sem lenne szabad, hogy a meglevő képjelek módosításának szükségességét vonja maga után.

A fenti szempontokból következik, hogy a Lexit úgy kell megterveznünk, hogy a formázó algoritmust legalább fordításkor, de lehetőség szerint akár futásidőben lecserélhessük. Az algoritmus elszigetelését és egyszersmind könnyű cserélhetőségét úgy biztosíthatjuk, ha egy objektumba tokozzuk be, pontosabban ha külön osztályhierarchiát hozunk létre azon objektumok részére, amelyek formázó algoritmusokat zárnak egységbe. A hierarchia csúcsán egy felület fog állni, ami formázó algoritmusok széles körét támogatja, alosztályai pedig ezt a felületet valósítják majd meg egy-egy algoritmus működtetéséhez. Ezután már nincs akadálya, hogy bevezessünk egy Glyph alosztályt, amely gyermekeit egy adott algoritmus objektum használatával rendezi.

Összeállítók és összetételek

Először is készítünk egy **Összeállító** (Compositor) osztályt azon objektumok számára, amelyek egységbe zárhatnak egy formázó algoritmust. A felület (2.2 táblázat) tudatja az összeállítóval, *mely* képjeleket kell formázni, és *mikor*. A formázandó képjelek az **Összetétel** (Composition) nevű különleges Képjel (Glyph) alosztály gyermekei. Az Összetétel osztály példányai létrehozásukkor megkapják az Összeállító alosztály egy példányát (amelyik az adott sortörési algoritmusra szakosodott), amelyet arra utasítanak, hogy a `Compose` (Összeállít) művelettel szükség esetén – például ha a felhasználó módosította a dokumentumot – rendezzék a képjeleket. A 2.5 ábra az Összetétel (Composition) és az Összeállító (Compositor) osztályok közötti viszonyokat mutatja.



2.5 ábra

A Composition és Compositor osztályok kapcsolata.

A formázatlan Összetétel objektumok csak azokat a látható képjeleket tartalmazzák, amelyek a dokumentum alaptartalmát alkotják. Olyan képjelek nem kapnak benne helyet, amelyek a dokumentum fizikai szerkezetét, például a sorokat (Row) vagy hasábokat (Column) határozzák meg. Most az összetételnek közvetlenül a létrehozása utáni állapotáról beszélünk, amikor is kezdetben az általa formázandó képjelekkel feltöltődik. Amikor formázásra van szükség, az összetétel meghívja összeállítója `Compose` műveletét, az összeállító pedig végigjárja az összetétel gyermekeit és új Row és Column képjeleket illeszt be, sortörési algoritmusának megfelelően[7]. Az előálló objektumszerkezetet a 2.6 ábra mutatja; ezen az összeállító által létrehozott és a szerkezetbe beillesztett képjeleket szürke háttérrel ábrázoltuk.

Minden Összeállító alosztály más és más sortörési algoritmust valósíthat meg. Az Egyszerű-Összeállító (SimpleCompositor) nevű például gyors áttekintésre lehet alkalmas, ha nem nézi az olyan „elhanyagolható" jellemzőket, mint a dokumentum „színe". (A dokumentum színe a szöveg és a térközök eloszlására utal: a „jó szín" ezek egyenletességét jelenti.) A teljes TeX algoritmus megvalósítására használhatunk egy TeXÖsszeállító (TeXCompositor) nevű alosztályt, ami a „színt" is figyelembe veszi, viszont lassabban dolgozik.

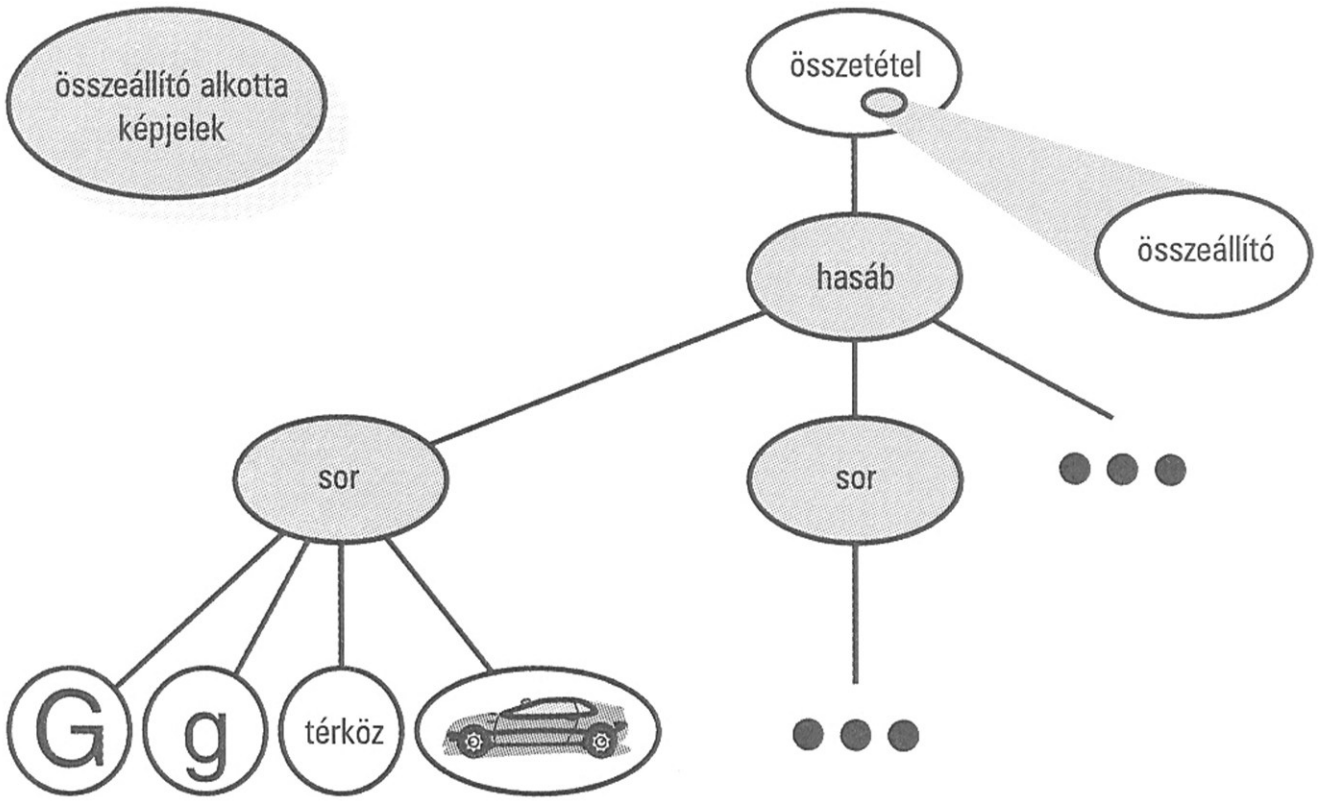
Az Összetétel–Összeállító osztálybontás biztosítja, hogy élesen elválaszthassuk egymástól a dokumentum fizikai szerkezetét alakító kódot a különböző formázó algoritmusok kódjától. Anélkül vehetünk fel a rendszerbe új Összeállító osztályokat, hogy hozzá kellene nyúlnunk a képjelosztályokhoz, és fordítva. Ha egy `SetCompositor` (Összeállító-beállító) műveletet adunk az Összetétel képjel-alapfelületéhez, még arra is lehetőségünk lesz, hogy futásidőben kicseréljük a sortörési algoritmust.

[7] A sortörések helyének kiszámításához az összeállítónak meg kell kapnia a karakter képjelek karakterkódjait. A 2.8 részben megnézzük, hogyan juthatunk hozzá ehhez az információhoz többféleképpen (a többalakúság segítségével), anélkül, hogy külön karakterműveletet adnánk a Glyph felülethez.

A Stratégia tervezési minta

A Stratégia minta lényege az algoritmusok objektumokba zárása. A minta kulcselemei a Stratégia (Strategy) objektumok (amelyek a különböző algoritmusokat egységbe zárják), és ezek működési környezete. Az összeállítók – lévén ezek is egy-egy formázó algorimust foglalnak magukba – szintén Stratégia objektumok, működési környezetük pedig az Összetétel.

A Stratégia minta alkalmazásának kulcsa olyan felületek tervezése a stratégia és környezete számára, amelyek elég általánosak ahhoz, hogy algoritmusok széles körét támogassák. Vagyis egy új algoritmus támogatásához nem szabad, hogy a stratégia vagy a környezet megváltoztatására legyen szükség. Példánkban a Képjel (Glyph) alapfelület kellően általánosan támogatta a gyermekek hozzáadását, eltávolítását és elérését ahhoz, hogy az Összeállító alosztályok az általuk használt algoritmustól függetlenül módosíthassák a dokumentum fizikai szerkezetét, és ehhez hasonlóan az Összeállító felület is megadott mindent az összetételeknek, hogy kezdeményezhessék a formázást.



2.6 ábra

Összeállító irányította sortörést tükröző objektumszerkezet.

2.4 A felhasználói felület finomítása

A Lexi felhasználói felületéhez két díszítő elemet adunk: egy szegélyt a szövegszerkesztő területhez, ami a szövegoldalt jelzi majd, illetve az oldal különböző részeinek megtekintését segítő gördítősávokat. Nem örökléssel adjuk ezeket a felülethez, mert azt szeretnénk, ha

minél könnyebben hozzáadhatók és elvehetők lennének (különösen futásidőben). Akkor érjük el a legnagyobb rugalmasságot, ha a többi felhasználói felületi elem nem is tud róla, hogy ezek a kiegészítő objektumok léteznek, így anélkül kapcsolhatjuk ki-be őket, hogy más osztályokat módosítanunk kellene.

Átlátszó befoglalás

Programozási szempontból a felhasználói felület finomítása a meglevő kód bővítését jelenti. Ha ezt öröklés révén érjük el, megfosztjuk magunkat attól a lehetőségtől, hogy a díszítő elemeket futásidőben átrendezhessük, de talán ennél is fontosabb, hogy az öröklés alapú megközelítés az osztályok számának nemkívánatos növekedését vonhatja maga után.

A szegélyt az Összetétel osztályból egy SzegélyesÖsszetétel (BorderedComposition) alosztályt létrehozva állíthatjuk elő, a gördítősávok felületét pedig ugyanígy, a GörgethetőÖsszetétel (ScrollableComposition) alosztály elkészítésével. Ha szegélyt és gördítősávokat is akarunk, egy SzegélyesGörgethetőÖsszetétel (BorderedScrollableComposition) osztályt is alkothatunk és így tovább. Szélsőséges esetben a díszítés minden változatára külön osztályunk lehet, mely megoldás a díszítő elemek változatosságának növekedésével hamar működésképtelenné válhat.

Az objektumösszetétel rugalmasabb és kezelhetőbb bővítési módot kínál. De milyen objektumokat párosítsunk össze? Tudjuk, hogy egy meglevő képjelet finomítunk, ezért magát a díszítést is objektummá tehetjük (mondjuk a **Szegély** – Border – osztály egy példányává), így az összetétel két elemből állhat elő: a képjelből és a szegélyből. A következő lépés annak eldöntése, hogy melyik elemet adjuk melyikhez. Az, hogy a szegély tartalmazza a képjelet, logikusnak tűnik, hiszen a képernyőn is a képjel lesz a szegély belsejében. De tehetjük ennek ellenkezőjét is, a szegélyt ágyazva a képjelbe, ám ekkor módosításokat kell végrehajtanunk a megfelelő Glyph alosztályon, hogy az tudomással bírjon a szegély létezéséről. Az első választás tehát jobbnak tűnik, hiszen így a szegélyrajzoló kódot teljes egészében a Border osztályon belül tarthatjuk, és nem zavarjuk a többi osztályt.

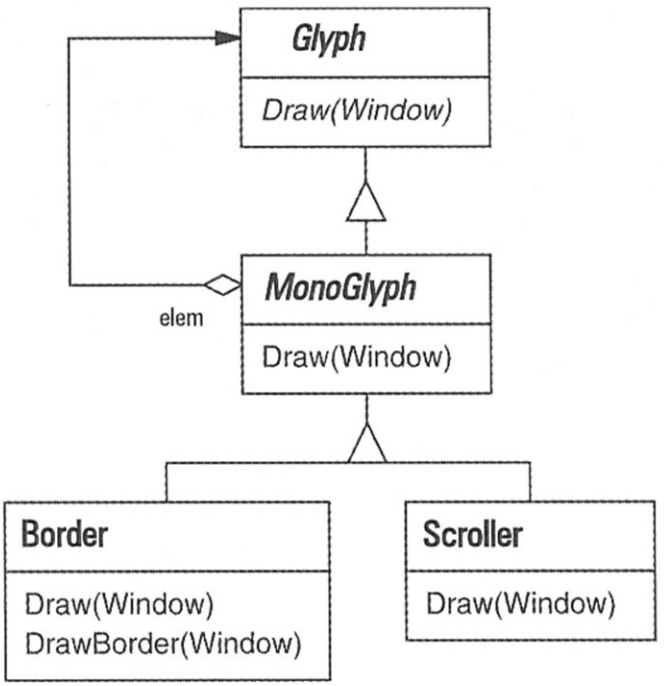
Hogyan nézzen ki a Border osztály? Az a tény, hogy a szegély grafikusan megjelenik a képernyőn, azt sugallja, hogy maga is képjel, így a Border a Glyph alosztálya kell legyen. Arra, hogy ezt tegyük, azonban van még egy okunk: a felhasználók nem törődnek azzal, hogy egy képjelnek van-e szegélye vagy sem, hanem egységesen szeretnék kezelni azokat. Ha egy szegély nélküli képjelnek azt mondják, hogy rajzolja meg magát, annak ezt mindenféle díszítés nélkül kell végrehajtania. Ha a képjel a szegély belsejében jön létre, a szegélyt is ugyanúgy kezelhetjük: arra, hogy rajzolja ki magát, ugyanúgy utasíthatjuk, mint bármely más képjelet. Ez mutatja, hogy a Border felületnek illeszkednie kell a Glyph felülethez; ezt a kapcsolatot azzal biztosítjuk, hogy a Border-t a Glyph alosztályaként hozzuk létre.

Mindez elvezet minket az **átlátszó befoglalás** (láthatatlan befoglalás, transparent enclosure) fogalmához, amelynek alapját az (1) egyetlen gyermekes (vagy egyelemű) összetétel, illetve az (2) összeegyeztethető felületek képezik. A felhasználó programok általában nem tud-

ják megállapítani, hogy magával az adott elemmel vagy a **befoglalójával** (vagyis a gyermek szülőjével) kerültek-e kapcsolatba, különösen ha a befoglaló egyszerűen átruházza műveleteit a befoglalt elemre. Emellett azonban a befoglaló ki is *bővítheti* elemének viselkedését, ha egy művelet átruházása előtt vagy után saját műveleteket is végez, illetve kiválóan alkalmas az elem állapotának beállítására is. Hamarosan meglátjuk, hogyan.

MonoGlyph

Az átlátszó befoglalás fogalmát bármely olyan képjellel kapcsolatban alkalmazhatjuk, ami egy másikat díszít. A fogalom konkretizálásához létrehozzuk a Glyph **MonoGlyph** nevű alosztályát, amely az olyan díszítő képjelek elvont osztálya lesz, mint a Border (lásd a 2.7 ábrát). A MonoGlyph egy hivatkozást tárol egy elemre, és minden kérelmet annak továbbít.



2.7 ábra
MonoGlyph osztálykapcsolatok.

Ez a MonoGlyph-et alapállapotban teljesen észrevehetetlenné teszi a felhasználó programok számára. A MonoGlyph például a következőképpen valósítja meg a `Draw` műveletet:

```
void MonoGlyph::Draw (Window* w) {
    elem->Draw(w);
}
```

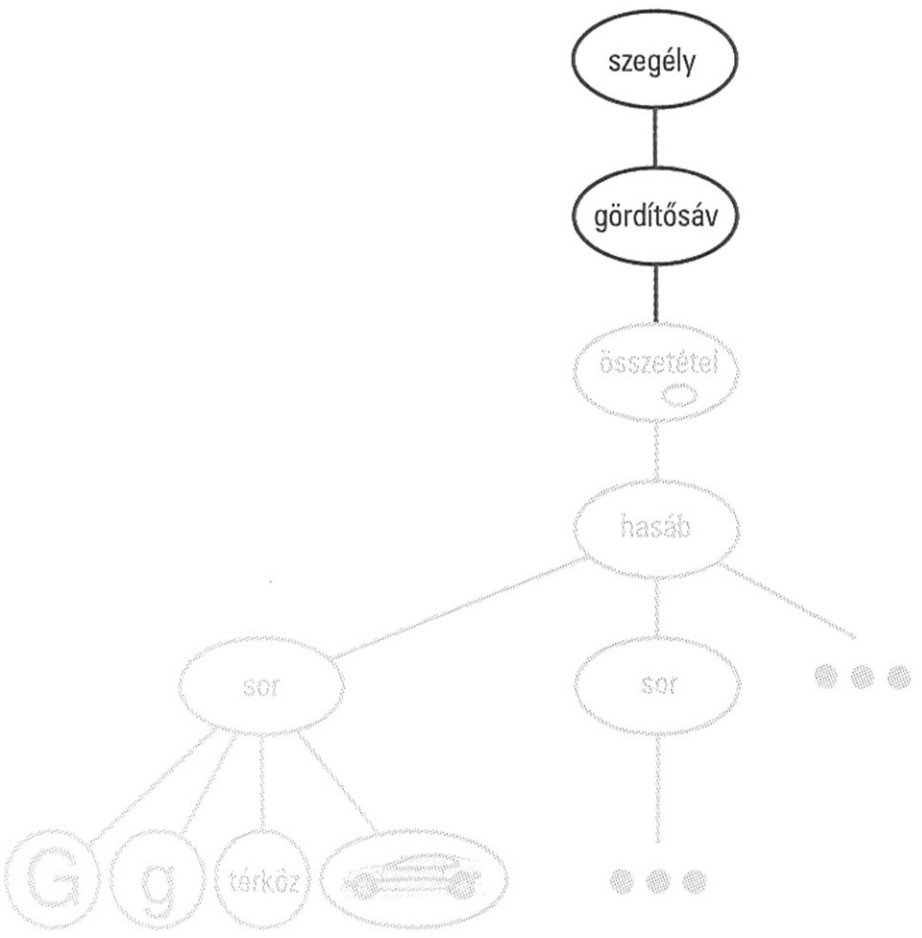
A MonoGlyph alosztályai ezen továbbító műveleteknek legalább az egyikét felülírják. A `Border::Draw` például először meghívja a szülőosztálybeli `MonoGlyph::Draw` műveletet az elemre, hogy az elvégezhesse a dolgát, vagyis a szegélyt leszámítva mindent kirajzoljon. Ezután a `Border::Draw` a `DrawBorder` (RajzolSzegély) nevű privát művelet meghívásával kirajzolja a szegélyt. (Ennek részleteibe most nem megyünk bele.)

```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

Megfigyelhetjük, milyen hatékonyan *bővíti ki* a `Border::Draw` a szülőosztálybeli műveletet a szegély megrajzolásához, ellentétben azzal, ha csupán *lecserélné* azt, amikor is a `MonoGlyph::Draw` hívás kimaradna.

Egy másik MonoGlyph alosztály a 2.7 ábrán látható. A **Scroller** (Görgető) olyan MonoGlyph, amely két gördítősáv alapján rajzolja ki magát különböző helyekre. Amikor a Scroller kirajzolja az elemét, a grafikus alrendszert arra utasítja, hogy a görgető határain túli részeket vágja le, az elem látható területről kigördített részei így nem jelennek meg a képernyőn.

Ezzel meg is vannak azok az elemek, amelyek segítségével a Lexi szövegszerkesztő területéhez szegélyt és gördítősávokat adhatunk. A meglevő Összetétel példányt a gördíthető felület kialakításához egy Görgető példányban hozzuk létre, azt pedig egy Szegély példányban. Az előálló objektumszerkezet a 2.8 ábrán látható.



2.8 ábra
Finomított objektumszerkezet.

Megjegyzendő, hogy az összetétel sorrendjét meg is fordíthatjuk, vagyis a szegéllyel ellátott Összetételt is tehetjük egy Görgető példányba, de ekkor a szöveggel együtt a szegély is továbbgördül, ami nem biztos, hogy a kívánt viselkedés. A lényeg azonban az, hogy az átlátszó befoglalás egyszerűvé teszi a különböző lehetőségekkel való kísérletezést, és a felhasználó programot megszabadítja attól, hogy a díszítő elemek kódjával foglalkoznia kelljen.

Megfigyelhetjük azt is, hogy a szegély csupán egyetlen képjelből áll, nem kettőből vagy annál is többől. Ez az eddig bemutatott összetételektől eltér, hiszen azoknál a szülőobjektumok tetszőleges számú gyermeket tartalmazhattak. A szegély azonban egyetlen elemet foglal magába, tehát csak egyetlen ilyen lehet. Egyszerre több elemhez is adhatnánk díszítést, de ekkor a díszítés fogalmát többféle összetétel fogalmával – sor, hasáb stb. – kellene kevernünk, ami nem szerencsés, hiszen ezekhez már rendelkezésre állnak a megfelelő osztályok. Jobb, ha az összetételhez a meglevő osztályokat használjuk, és a díszítést új osztályokra bízzuk. Ezzel a külalak finomítását elválaszthatjuk a többi összetételtől, ami egyszerre egyszerűsíti a díszítő osztályokat, tartja alacsonyan számukat, illetve akadályozza meg, hogy a már meglevő szolgáltatásokat ismételten elkészítsük.

A Díszítő minta

A Díszítő minta az átlátszó befoglalás segítségével ragadja meg a díszítést támogató osztály- és objektumkapcsolatokat. A „díszítés" fogalma persze szélesebb annál, mint ahogy eddig használtuk. A Díszítő mintában e fogalom mindenre vonatkozik, ami egy objektum feladatkörét kibővíti. Például egy elvont szintaxisfát szemantikai műveletekkel díszíthetünk, egy véges állapotú automatát új állapot-átmenetekkel, vagy maradandó objektumok hálóját jellemzőcímkékkel. A Díszítő minta általánosítja a Lexi kapcsán bemutatott megközelítést, hogy az szélesebb körben is alkalmazható legyen.

2.5 Több megjelenítési szabvány támogatása

A rendszertervezés egyik fő problémája, hogyan oldjuk meg a különböző hardver- és szoftverfelületek közötti átjárást. Ha a Lexit más rendszerre szeretnénk átültetni, nyilván nem akarjuk, hogy ez kimerítő munkával járjon, másképp nem érné meg az egész. Más szóval: az átültetést olyan egyszerűvé kell tenni, amennyire csak lehetséges.

A hordozhatóság egyik akadálya az alkalmazások egységes kinézetét biztosító megjelenítési szabványok (look-and-feel standard) sokasága. Ezek a szabványok adnak irányelveket arra vonatkozóan, hogy az egyes programok hogyan jelenjenek meg és hogyan reagáljanak a felhasználók tevékenységére. A ma létező szabványok persze nem térnek el túlságosan egymástól, mégis összetéveszthetetlenek. A Motif alkalmazások például nem egészen úgy néznek ki, mint más rendszereken futó társaik, és azokról is elmondható, hogy más hatást keltenek, mint a Motif programok. Egy olyan alkalmazásnak, amely képes több rendszeren is futni, igazodnia kell az adott rendszerek felhasználói felületéhez.

Tervezési célunk, hogy a Lexit képessé tegyük arra, hogy több meglevő megjelenítési szabványhoz alkalmazkodjon, illetve hogy könnyű legyen az esetleges új szabványok támogatását is beleépíteni. Emellett azt is szeretnénk, ha a rugalmasság csúcsaként futásidőben is módosíthatnánk a program megjelenését.

Az objektum-létrehozás elvonatkoztatása

Minden, amit látunk, és amivel kapcsolatba kerülünk a Lexi felhasználói felületén, egy képjel, amely más, láthatatlan képjelekbe (Sor, Hasáb stb.) ágyazódik. A láthatatlan képjelek láthatókat állítanak elő (Gomb, Karakter stb.), és megfelelően elrendezik azokat. A különböző megjelenítési stílusok útmutatói részletesebben is leírják a felület ezen vezérlő elemeinek („widget") – gombok, gördítősávok, menük – kinézetét. A vezérlők egyszerűbb képjelek, például karakterek, körök, négyszögek és sokszögek alapján építhetik fel magukat.

Tételezzük fel, hogy két vezérlőosztályunk van, amelyek a különböző megjelenítési szabványok támogatására hivatottak:

1. Elvont Képjel alosztályok halmaza minden vezérlőelem-kategóriához. Például egy elvont GördítőSáv (ScrollBar) osztállyal a képjel alapfelületét gördítőműveletekkel bővíthetjük ki, egy Gomb (Button) osztállyal gombműveleteket adhatunk hozzá és így tovább.
2. Konkrét alosztályok halmaza minden elvont alosztályhoz, amelyek megvalósítják a különböző megjelenítési szabványokat. A ScrollBar-nak például lehetnek MotifScrollBar és PMScrollBar alosztályai, amelyek Motif és PM (Presentation Manager) stílusú gördítősávokat hoznak létre.

A Lexinek meg kell tudnia különböztetni az egyes megjelenítési stílusokhoz kapcsolódó vezérlőket, például ha egy gombot kell kirajzolnia a program felületére, tudnia kell, melyik gombváltozat osztályát kell példányosítania (MotifButton, PMButton, MacButton stb.).

Világos, hogy a Lexi megvalósítása ezt nem tudja közvetlenül, például C++-ban egy konstruktorhívással megtenni. Ez a gombstílus kódban való rögzítését vonná maga után, így futásidőben nem választhatnánk stílust, ráadásul ha a programot másik rendszerre szeretnénk átültetni, az összes ilyen konstruktorhívást meg kellene keresnünk és meg kellene változtatnunk. És a gombok csupán a vezérlők egyik fajtáját jelentik a Lexi felhasználói felületén... Az adott megjelenítési módhoz kapcsolódó osztályok konstruktorhívásainak elszórása a kódban a karbantartók rémálma – ha csak egy felett is átsiklik a tekintetünk, ott találhatjuk magunkat egy Mac programban, aminek a közepén egy Motif stílusú menü éktelenkedik.

A Lexinek meg kell határoznia a megcélzott megjelenítési stílust, hogy létrehozhassa a megfelelő vezérlőelemet. Nem csak a kifejezett konstruktorhívásokat kell elkerülnünk, hanem azt is biztosítanunk kell, hogy képesek legyünk könnyedén egy egész vezérlőkészletet lecserélni. Mindkettő elérhető, ha *az objektum-létrehozás folyamatát elvonttá tesszük.* Egy példával mindjárt meg is világítjuk, mire gondolunk.

Gyárak és termékosztályok

Normális esetben a következő C++ kóddal hozhatunk létre egy Motif stílusú gördítősáv-példányt:

```
ScrollBar* sb = new MotifScrollBar;
```

Ezt a fajta kódot kell elkerülnünk, ha a Lexi megjelenítési szabványoktól való függőségét a lehető legkisebbre szeretnénk szorítani. De tegyük fel, hogy az `sb` gördítősávot az alábbiak szerint hozzuk létre:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

Itt a `guiFactory` a **MotifFactory** (MotifGyár) osztály egy példánya. A `CreateScrollBar` (LétrehozGördítőSáv) a megfelelő ScrollBar alosztály egy pédányát adja vissza, a kívánt megjelenítési stílusnak, ebben az esetben a Motifnak megfelelően. Ami a felhasználó programokat illeti, a fenti kód hatása megegyezik azzal, mintha közvetlenül a MotifScrollBar konstruktort hívnánk meg – de van egy lényeges különbség: a kódban immár nem utal semmi név szerint a Motif stílusra. A `guiFactory` objektum nem csak a Motif gördítősávok létrehozásának folyamatát teszi elvonttá, hanem *bármely* megjelenítési stílusnak megfelelő gördítősávét. Emellett a `guiFactory` nem szorítkozik a gördítősávok előállítására; vezérlők széles körét képes elkészíteni, így gombokat, beviteli mezőket, menüket és más vezérlőket is.

Mindez azért lehetséges, mert a MotifFactory a **GUIFactory** (GUIGyár) alosztálya, ami a vezérlőelemek létrehozására szolgáló általános felületet meghatározó elvont osztály. Olyan műveleteket tartalmaz, mint a `CreateScrollBar` vagy a `CreateButton` (LétrehozGomb), amelyek különféle vezérlőelemeket példányosítanak. A GUIFactory alosztályai ezen műveletek megvalósításával olyan képjeleket adnak vissza, mint a MotifScrollBar vagy a PMButton, amelyek egy bizonyos megjelenítési stílust képviselnek. A 2.9 ábra a `GUIFactory` objektumok osztályhierarchiáját mutatja.

Azt mondjuk, hogy ezek a „gyárak" **termék** (product, produktum) objektumokat állítanak elő. Az egy adott gyár által létrehozott termékek rokonságban állnak egymással; esetünkben minden termék egy adott megjelenítési stílushoz igazodó vezérlő. A gyárak számára a vezérlők működtetéséhez szükséges termékosztályok egy részét a 2.10 ábra mutatja.

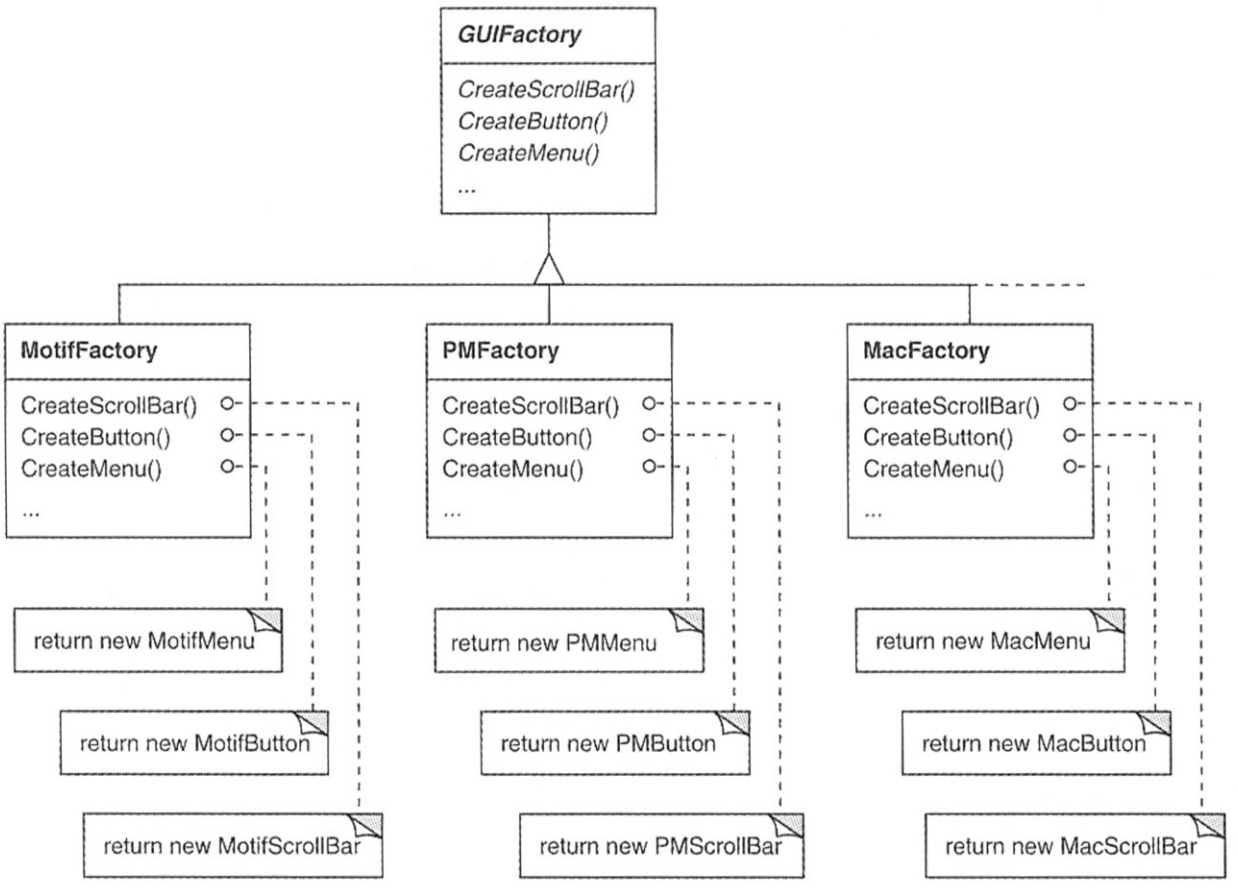
Az utolsó kérdés, amit meg kell válaszolnunk, hogy honnan származik a GUIFactory példány? A válasz pedig az, hogy bárhonnan, ha számunkra kényelmes. A `guiFactory` változó lehet általános (globális), lehet egy jól ismert osztály statikus tagja, de lehet helyi változó is, amennyiben a teljes felhasználói felületet egyetlen osztályon vagy függvényen belül hozzuk létre. Az ilyen objektumok kezelésére létezik egy tervezési minta is, az Egyke (Singleton). A lényeg az, hogy a `guiFactory` bevezetésére a program azon pontján van szükség, amikor *még* nem kell vezérlőelemeket létrehoznunk vele, de *már* döntés született a használni kívánt megjelenítési szabványról.

Ha a megjelenítési stílust fordításkor ismerjük, a `guiFactory` létrehozását a program elején, egyszerűen egy új gyárpéldányt hozzárendelve elintézhetjük:

```
GUIFactory* guiFactory = new MotifFactory;
```

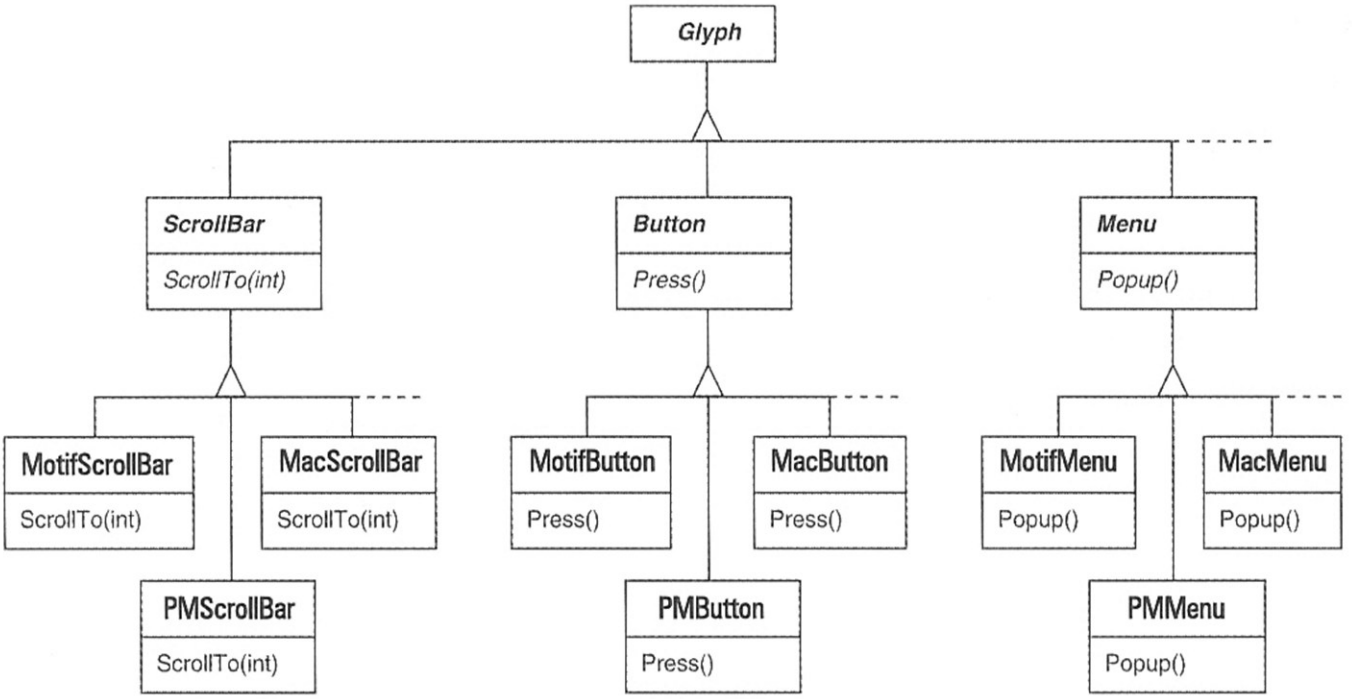
Ha a felhasználó adhatja meg a program indításakor a megjelenítési stílust – mondjuk egy karakterlánc beírásával –, a gyárat létrehozó kód a következő lehet:

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
   // indításkor adja meg a környezet vagy a felhasználó

if (strcmp(styleName, "Motif") == 0) {
   guiFactory = new MotifFactory;

} else if(strcmp(styleName, "Presentation_Manager") == 0) {
   guiFactory = new PMFactory;

} else {
   guiFactory = new DefaultGUIFactory;
}
```



2.9 ábra

A GUIFactory osztályhierarchiája.



2.10 ábra

Elvont termékosztályok és konkrét alosztályok.

Ennél persze vannak kifinomultabb módszerek is arra, hogy futásidőben kiválasszuk a megfelelő gyárat. Készíthetünk például egy bejegyzés-adatbázist, amely karakterláncokat rendel az egyes gyárobjektumokhoz. Így anélkül jegyezhetünk be új gyár alosztályokat, hogy a meglevő kódot módosítanunk kellene, amit az előző megközelítés megkövetel – ráadásul nem kell minden rendszerfüggő gyárat belefordítanunk az alkalmazásba. Ez igen fontos, hiszen a MotifFactory-t egy, a Motifot nem támogató rendszeren esetleg nem is kapcsolhatjuk a programhoz.

A lényeg azonban az, hogy ha egyszer beállítottuk az alkalmazást a megfelelő gyár objektum használatára, azzal meghatároztuk a megjelenítési stílusát. Ha meggondoljuk magunkat, újratölthetjük a `guiFactory`-t egy más megjelenítést támogató gyárral és újraépíthetjük a felületet. Nem számít, mikor és hogyan döntünk e kérdésben, tudni fogjuk, hogy a döntés után a program képes lesz beállítani saját magát, anélkül, hogy bármit is módosítanunk kellene.

Az Elvont gyár tervezési minta

A gyárak és termékek az Elvont gyár minta kulcsfontosságú elemei. Ez a minta arra nyújt megoldást, hogyan hozhatjuk létre rokonságban álló termékobjektumok családját anélkül, hogy az osztályokat közvetlenül példányosítanunk kellene. A módszer használata akkor a legcélszerűbb, ha a termékobjektumok száma és típusa állandó, az egyes termékcsaládok között pedig különbségek vannak. A családok közül úgy választunk ki egyet, hogy egy konkrét gyárat példányosítunk, és következetesen azzal hozzuk létre a termékeket. Ha az

adott gyárat egy másik egy példányával váltjuk fel, egész termékcsaládokat cserélhetünk ki. Az Elvont gyárat az különbözteti meg a többi, egyetlen típusú termékobjektumot használó alkotó létrehozási mintától, hogy a hangsúlyt a termékek *családjára* helyezi.

2.6 Több ablakkezelő rendszer támogatása

A megjelenítési stílus csak egyike a hordozhatóság szempontjainak. A másik az ablakkezelő rendszer nyújtotta környezet, amelyben a Lexi fut. Az ablakkezelő rendszer adja azt az illúziót, hogy egy háttérképpel ellátott asztalon egymásra pakolt ablakokat látunk, ez határozza meg, hogy az ablakok mekkora és milyen képernyőterületet foglalhatnak el, és ez irányítja hozzájuk a billentyű- és egérüzeneteket. Számos elterjedt és egymással nagyrészt összeegyeztethetetlen ablakkezelő rendszer létezik (Macintosh, Presentation Manager, Windows, X stb.), mi pedig – ugyanabból az okból kifolyólag, amiért a különböző megjelenítési stílusokat is támogattuk –azt szeretnénk, ha a Lexi a lehető legtöbbjükön futna.

Használhatunk elvont gyárat?

Első pillantásra úgy tűnik, itt egy újabb lehetőség az Elvont gyár minta használatára. Csakhogy az ablakkezelő rendszer hordozhatóságára vonatkozó megszorítások jelentősen különböznek azoktól a követelményektől, amelyek a megjelenítési stílus függetlenségét biztosítják.

Az Elvont gyár alkalmazásánál feltételeztük, hogy minden megjelenítési szabványhoz meghatározunk egy konkrét vezérlőosztályt, ami azt jelentette, hogy egy elvont termékosztályból (amilyen pl. a ScrollBar) adott megjelenítési stílusnak megfelelő konkrét termékeket (MotifScrollBar, MacScrollBar stb.) származtathattunk. Ebben az esetben azonban azzal a feltevéssel kell élnünk, hogy különböző gyártóktól számos osztályhierarchiával rendelkezünk, amelyek egy-egy megjelenítési szabványt támogatnak, és nagy valószínűséggel egymással egyáltalán nem összeegyeztethetők. Így aztán nem lesz egy közös elvont termékosztályunk a különböző vezérlőkhöz (ScrollBar, Button, Menu stb.), e létfontosságú osztály nélkül pedig az Elvont gyár nem működik. Először meg kell oldanunk, hogy a különböző vezérlőhierarchiák képesek legyenek igazodni elvont termékfelületek egy közös halmazához, csak ezután vezethetjük be megfelelően a `Create...` műveleteket elvont gyárunk felületében.

A vezérlők esetében a fenti problémát úgy oldottuk meg, hogy megalkottuk saját elvont és konkrét termékosztályainkat. Most, amikor a Lexit megpróbáljuk képessé tenni arra, hogy több különböző ablakkezelő rendszeren is működőképes legyen, hasonló problémával kerülünk szembe, mégpedig azzal, hogy e rendszerek össze nem egyeztethető programozási felületekkel rendelkeznek. Ez egy kicsit keményebb dió, mint az előző, hiszen nem engedhetjük meg magunknak, hogy saját, nem szabványos ablakrendszert fejlesszünk ki.

Szerencsére van megoldás. Akárcsak a megjelenítési szabványok, az ablakkezelő rendszerek felületei sem különböznek gyökeresen egymástól, hiszen lényegében ugyanazokat

a feladatokat hajtják végre. Tehát az ablakkezelő rendszer fogalmainak egységes halmazára lesz szükségünk, így a különböző megvalósításokat egyetlen közös felületbe tuszkolhatjuk.

A megvalósítási függőségek egységbe zárása

A 2.2 részben a képjelek, illetve képjelszerkezetek képernyőn való megjelenítésére bevezettük a Window (Ablak) osztályt. Nem határoztuk meg az ablakkezelő rendszert, amellyel működik, hiszen ilyen kimondottan nem is volt. A Window osztály zárja egységbe azokat a dolgokat, amelyért az ablakok egy ablakrendszerben általában felelnek:

- Műveleteket biztosítanak az alapvető geometriai alakzatok rajzolásához.
- Lekicsinyíthetik és felnagyíthatják magukat.

Felelősségi kör	Műveletek

Ablakkezelés	`virtual void Redraw()`
	`virtual void Raise()`
	`virtual void Lower()`
	`virtual void Iconify()`
	`virtual void Deiconify()`
	`...`

Grafika	`virtual void DrawLine(...)`
	`virtual void DrawRect(...)`
	`virtual void DrawPolygon(...)`
	`virtual void DrawText(...)`
	`...`

2.3 táblázat

A Window osztály felülete.

- Átméretezhetik magukat.
- Tartalmukat igény szerint újrarajzolhatják, például amikor kis méretről visszaállítják őket eredeti méretükre, amikor egymással fedésbe kerülnek, vagy amikor addig elfedett területük a képernyő előterébe kerül.

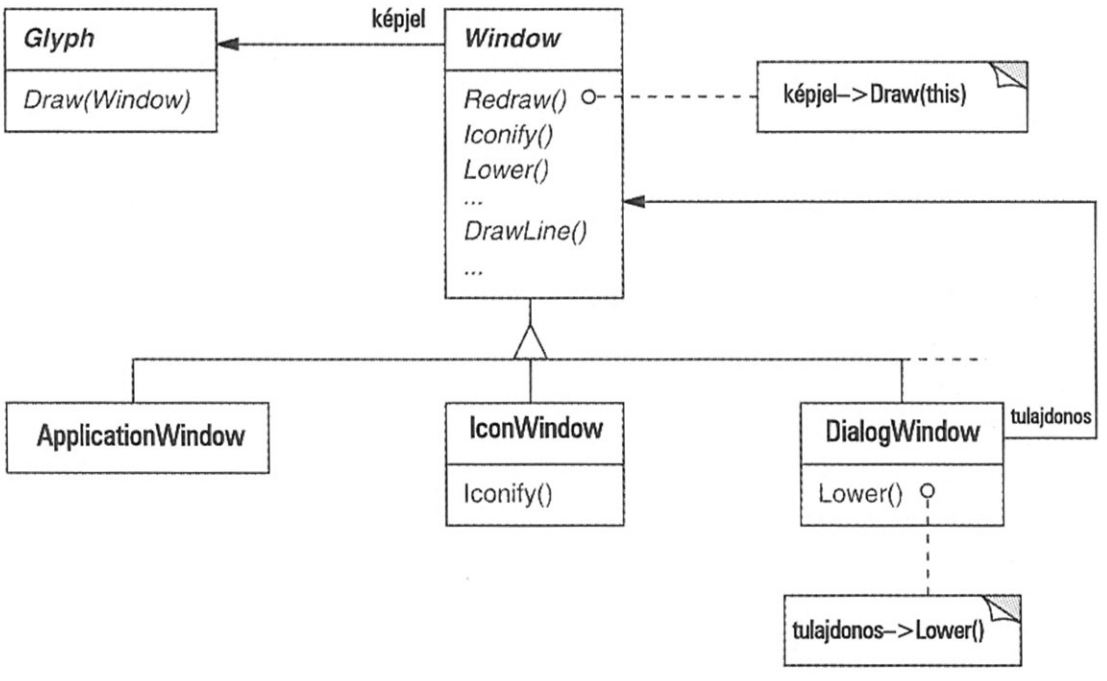
A Window osztálynak le kell fednie a különböző ablakrendszerek szolgáltatásainak teljes körét. Lássunk két szélsőséges megközelítést:

1. *A szolgáltatások metszete.* A Window osztály felülete csak azokat a szolgáltatásokat tartalmazza, amelyek *minden* ablakrendszerben megtalálhatók. Ezzel a megközelítéssel az a gond, hogy Window felületünk csak annyit fog tudni, amennyit a legkevesebb szolgáltatást nyújtó rendszer, így nem aknázhatjuk majd ki a fejlettebb képességeket, még ha a legtöbb (de nem az összes) ablakkezelő rendszer támogatja is azokat.
2. *A szolgáltatások uniója.* Létrehozunk egy felületet, amelyben helyet kap *minden* létező rendszer minden képessége. A gond itt az, hogy az eredményként előálló felü-

let hatalmassá és következetlenné válhat, ráadásul minden alkalommal módosítanunk kell (és vele a Lexit is, amely rá támaszkodik), ha akár csak egyetlen fejlesztőcég is átalakítja ablakkezelő rendszerének felületét.

Egyik szélsőséges megoldás sem kivitelezhető, így az arany középutat igyekszünk megtalálni. Window osztályunk kezelhető felülettel fog rendelkezni, és a legtöbb népszerű szolgáltatást támogatja majd. Miután a Lexi közvetlenül használja ezt az osztályt, a Window-nak támogatnia kell azokat az elemeket is, amelyeket a Lexi ismer, vagyis a képjeleket. Ez azt jelenti, hogy a Window felületének alapvető grafikai műveleteket is tartalmaznia kell, amelyek segítségével a képjelek kirajzolhatják magukat az ablakban. A 2.3 táblázat egy lehetséges művelethalmazt mutat, amit a Window osztály felülete tartalmazhat.

A Window elvont osztály. Konkrét alosztályai biztosítják a különféle ablakok támogatását: az alkalmazások ablakai, az ikonok, a figyelmeztető üzenetek mind ablakok, de viselkedésük némileg különböző. Ezeknek megfelelően olyan alosztályokat határozhatunk meg, mint az AlkalmazásAblak (ApplicationWindow), az IkonAblak (IconWindow) vagy a PárbeszédAblak (DialogWindow). Az előállított osztályhierarchia a Lexihez hasonló alkalmazásoknak egységes, elvont ablakműködést biztosít, amely nem függ egyetlen fejlesztőcég ablakkezelő rendszerétől sem.



Most, hogy meghatároztunk egy ablakfelületet a Lexi számára, amellyel működhet, hol kezeljük ténylegesen a rendszerfüggő ablakokat? Ha nem készítünk saját ablakkezelő rendszert, elvont ablakainkat előbb-utóbb a célrendszer által biztosított fogalmak segítségével kell felépítenünk. Vagyis mi lesz a megvalósítással?

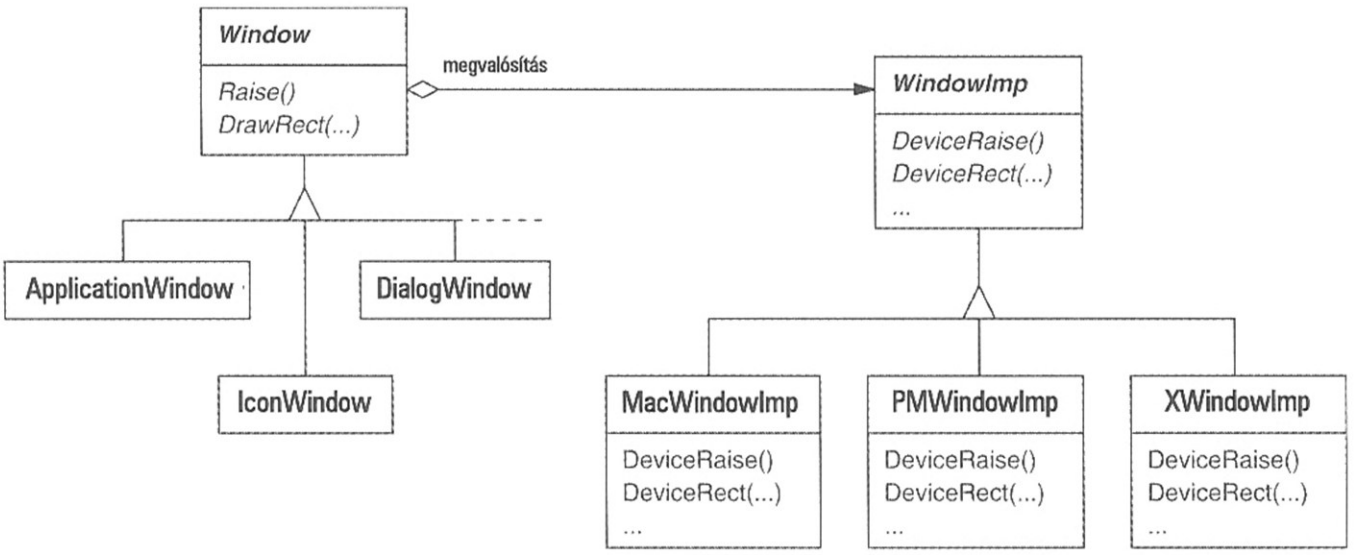
Az egyik lehetőség, hogy minden rendszerre külön változatot készítünk a Window osztályból és alosztályaiból, és akkor választjuk ki a megfelelőt, amikor egy adott felületre lefordít-

juk a programot. Ez azonban komoly fejfájást okozhat, hiszen számos Window nevű, de más-más rendszerre megvalósított osztályt kell számon tartanunk. Azt is megtehetjük, hogy a Window hierarchiában levő minden osztályból megvalósításfüggő alosztályokat hozunk létre – de ez az osztályok számának hasonló robbanásához vezet, mint amiről a díszítés kapcsán már szót ejtettünk. Emellett mindkét módszernek van még egy hátulütője: egyik sem nyújt kellő rugalmasságot, ugyanis a program lefordítása után már nincs lehetőségünk módosítani az ablakrendszert, ráadásul emiatt több futtatható állományt kell fenntartanunk.

Egyik lehetőség sem túl vonzó, de mi mást tehetünk? Nos, ugyanazt, amit a formázás és a díszítés esetében: a *változó tényezőt egységbe kell zárnunk.* Ez pedig ebben az esetben az ablakkezelő rendszer megvalósítása. Ha az ablakrendszer szolgáltatásait egy objektumba zárjuk, az objektum felületéhez igazodva készíthetjük el a Window osztályt és annak alosztályait. Ha pedig a felület minden kívánt ablakrendszert képes kiszolgálni, az említett osztályok egyikét sem kell módosítanunk a rendszer támogatásához. Az ablakobjektumokat egyszerűen úgy igazíthatjuk a kívánt rendszerhez, hogy átadjuk a megfelelő ablakrendszer-betokozó objektumot. Így az ablakok futásidőben is módosíthatók.

Window és WindowImp

Létrehozunk tehát egy külön **WindowImp** (AblakMegvalósítás) osztályhierarchiát, amelyben elrejtjük a különböző ablakkezelő rendszerek megvalósításait. A WindowImp egy elvont osztály lesz azon objektumok számára, amelyek a rendszerfüggő kódokat tartalmazzák. Ahhoz, hogy a Lexi egy adott ablakrendszeren működhessen, az egyes ablakobjektumokat a WindowImp megfelelő alosztályának egy példányával állítjuk be. Az alábbi diagram a Window és WindowImp hierarchiák közötti kapcsolatot mutatja:



A megvalósításokat a WindowImp osztályokba rejtve elkerülhetjük a Window osztályok rendszerfüggő kódokkal való „beszennyezését", és a Window hierarchiát kis méreten és stabilan tarthatjuk, miközben a megvalósítási hierarchia bővítésével az új ablakkezelők támogatása könnyen megoldható marad.

A WindowImp alosztályai

A WindowImp alosztályai a kérelmeket ablakrendszer-függő műveletekké alakítják át. Ha visszaemlékszünk a 2.2 részben bemutatott példára, ott a téglalaprajzoló `Rectangle::Draw` műveletet a Window példányon a `DrawRect` művelet alapján határoztuk meg:

```
void Rectangle::Draw (Window* w) {
   w->DrawRect (_x0, _y0, _x1, _y1);
}
```

A `DrawRect` alapértelmezett megvalósítása a WindowImp által bevezetett elvont téglalaprajzoló műveletet használja:

```
void Window::DrawRect (
   Coord x0, Coord y0, Coord x1, Coord y1
) {
   _imp->DeviceRect (x0, y0, x1, y1);
}
```

A fenti kódban az `_imp` a Window tagváltozója, és az annak beállítására használt WindowImp-et tartalmazza. Az ablak-megvalósítást tehát azon WindowImp alosztálypéldány határozza meg, amelyre az `_imp` mutat. Egy XWindowImp (vagyis az X Window rendszer számára készített WindowImp alosztály) esetében a `DeviceRect` megvalósítása például az alábbi alakot öltheti:

```
void XWindowImp::DeviceRect (
   Coord x0, Coord y0, Coord x1, Coord y1
) {
int x = round(min(x0, x1));
int y = round(min(y0, y1));
int w = round(min(x0 - x1));
int h = round(min(y0 - y1));
XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

A `DeviceRect` meghatározása azért így fest, mert az `XDrawRectangle` (a téglalaprajzolásra használt X felület) a téglalapokat bal alsó sarkukkal, szélességükkel és magasságukkal határozza meg; a `DeviceRect`-nek a kapott adatokból ezeket kell kiszámolnia. Először megállapítja, melyik a bal alsó sarok – elvégre az `(x0,y0)` bármelyik lehet a négy sarok közül –, majd kiszámítja a szélességet és magasságot.

A PMWindowImp (a WindowImp alosztálya a Presentation Manager számára) másképp határozná meg a `DeviceRect`-et:

```
void PMWindowImp::DeviceRect (
   Coord x0, Coord y0, Coord x1, Coord y1
    ) {
```

```
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left; point[0].y = top;
    point[1].x = right;point[1].y = top;
    point[2].x = right;point[2].y = bottom;
    point[3].x = left; point[3].y = bottom;

    if (
          (GpiBeginPath(_hps, 1L) == false) ||
          (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
          (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
          (GpiEndPath(_hps) == false)
    ) {
          //hibajelentés

    } else {
          GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Miért különbözik ez ennyire az X változattól? Nos, az az oka, hogy az X-szel ellentétben a PM nem rendelkezik kifejezett téglalaprajzoló művelettel, csak egy általános felülettel, amit a többszakaszos alakzatok (úgynevezett **görbék**, path) vektorainak meghatározására, illetve az általuk körbezárt terület határolására és kitöltésére használ.

A PM `DeviceRect`-megvalósítása tehát meglehetősen különbözik az X-étől, de ez nem számít. A WindowImp az ablakrendszer-felületek változatosságát lehet, hogy nagy, de kellően stabil felület mögé rejti, így a Window alosztályok írói az elvont ábrázolással foglalkozhatnak és nem kell az ablakkezelő rendszer részleteivel törődniük, ráadásul az új rendszerek támogatásának beépítése is lehetővé válik a Window osztályok módosítása nélkül.

Az ablakok beállítása az ablak-megvalósítások segítségével

Egy kulcsfontosságú kérdés, amelyet még nem érintettünk, hogyan állítunk be egy ablakot a megfelelő WindowImp alosztállyal? Másképp fogalmazva: mikor kerül sor az `_imp` kezdeti beállítására, és hogyan állapítjuk meg, hogy éppen melyik ablakkezelőt (és következésképpen melyik WindowImp alosztályt) kell használjuk? Az adott ablaknak valamilyen WindowImp-re mindenképpen szüksége lesz, mielőtt bármit csinálhatna.

Több lehetőségünk is van, de mi arra összpontosítunk, amelyik az Elvont gyár mintát alkalmazza. Létrehozhatunk egy elvont gyár osztályt (WindowSystemFactory, AblakRendszerGyár), amely felületet biztosít a különféle rendszerfüggő megvalósítási objektumok létrehozásához:

```
class WindowSystemFactory {
public:
   virtual WindowImp* CreateWindowImp() = 0;
   virtual ColorImp* CreateColorImp() = 0;
   virtual FontImp* CreateFontImp() = 0;

   // Create... művelet minden ablak-erőforráshoz
};
```

Ezután minden ablakkezelő rendszerhez meghatározhatunk egy konkrét gyárat:

```
class PMWindowSystemFactory : public WindowSystemFactory {
   virtual WindowImp* CreateWindowImp()
          { return new PMWindowImp; }
   //...
};
class XWindowSystemFactory : public WindowSystemFactory {
   virtual WindowImp* CreateWindowImp()
          { return new XWindowImp; }
   //...
};
```

A Window alaposztály konstruktora a `WindowSystemFactory` felület segítségével adhat kezdőértéket (a használatban levő ablakkezelőnek megfelelő WindowImp-et) az `_imp` tagnak:

```
Window::Window () {
   _imp = windowSystemFactory->CreateWindowImp();
}
```

A `windowSystemFactory` változó egy WindowSystemFactory alosztály jól ismert példánya, amely hasonló a megjelenítési stílust meghatározó `guiFactory` változóhoz. A `windowSystemFactory`-nek ugyanazzal a módszerrel adhatunk kezdőértéket.

A Híd tervezési minta

A WindowImp osztály az ablakrendszer szolgáltatásainak közös felületét határozza meg, de felépítését más megszorítások kötik, mint amik a Window felületére vonatkoznak. Az alkalmazásprogramozók nem nyúlnak közvetlenül a WindowImp felületéhez, csak a Window objektumokat kezelik. Így a WindowImp felületének nem kell igazodnia az alkalmazásprogramozó nézőpontjához, ami a Window osztályhierarchia és felület megtervezésénél szempont volt. A WindowImp felülete közelebbről tükrözheti az ablakkezelő rendszer nyújtotta szolgáltatásokat, és a metszet, illetve unió megközelítés felé tetszőlegesen tolódhat, attól függően, melyik illeszkedik jobban a célrendszerekhez.

A lényeg az, hogy megértsük, a Window felület az alkalmazásprogramozó, míg a WindowImp az ablakkezelő rendszer igényeit elégíti ki. Az, hogy a szolgáltatásokat két hierarchiába rendezzük, lehetővé teszi, hogy a két felületet egymástól függetlenül valósíthassuk meg, a Lexi több rendszeren való működését pedig a két hierarchia objektumainak együttműködése biztosítja.

A Window és a WindowImp közötti kapcsolat a Híd tervezési mintára ad példát. A Híd mögött megbúvó szándék az, hogy lehetővé tegyük önálló, egymástól függetlenül fejleszthető osztályhierarchiák együttműködését. Tervezési céljaink ahhoz vezettek, hogy létrehoztunk két ilyen osztályhierarchiát, amelyek közül az egyik az ablakok logikai fogalmát, a másik pedig különböző megvalósításaikat támogatta. A Híd minta alkalmazásával az elvont logikai ablakokat úgy fejleszthetjük tovább, hogy nem kell az ablakrendszer-függő kódhoz hozzányúlnunk, és ez fordítva is igaz.

2.7 Felhasználói műveletek

A Lexi szolgáltatásainak egy része a dokumentum vizuális megjelenítésén keresztül érhető el: szöveget írunk be és törlünk, arrébb visszük a beszúrási pontot, az egérrel szövegrészleteket jelölünk ki. Más szolgáltatásokat közvetve, a Lexi menüin, gombjain és gyorsbillentyűin keresztül használunk. Az ezek mögötti műveletek közé a következők tartoznak:

- Új dokumentum létrehozása
- Létező dokumentum megnyitása, mentés és kinyomtatása
- Kijelölt szöveg kivágása a dokumentumból, illetve beillesztése a dokumentum más részére
- Kijelölt szöveg stílusának és betűtípusának módosítása
- A szövegformázás – igazítás, sorkizárt elrendezés – módosítása
- Kilépés a programból
- Egyéb műveletek

E műveleteket a Lexi különböző felületeken teszi elérhetővé a felhasználó számára, de egyetlen művelet sem kötődik egyetlen felülethez. Célunk az, hogy a műveletek többféle módon is végrehajthatók legyenek, például egy másik oldalra átugorhassunk egy gomb, de akár egy menüpont segítségével is. Az is elképzelhető, hogy a jövőben változtatni szeretnénk a felhasználói felületen.

A műveleteket emellett több különböző osztályban valósítjuk meg. Fejlesztőként anélkül szeretnénk elérni őket, hogy a felhasználói felületi és megvalósító osztályok között túl sok függőséget alakítanánk ki, mert a túl szorosan csatolt megvalósítás kevésbé átlátható és karbantartható, illetve nehezebben bővíthető.

Tovább bonyolítja a dolgokat, hogy azt szeretnénk, ha a Lexi támogatná néhány, *de nem minden* művelet visszavonását és megismétlését[8]. Egészen pontosan az olyan dokumentum-módosító műveletek visszavonására szeretnénk lehetőséget, mint a törlés, amivel a felhasználó szándékán kívül rengeteg adatot törölhet véletlenül, de nem kívánunk visszavonni olyan műveleteket, mint egy rajz mentése vagy a programból való kilépés. E műveleteknek függetleneknek kell lenniük a visszavonási folyamattól. Emellett korlátozni sem kívánjuk a visszavonások számát.

[8] Vagyis egy éppen visszavont művelet újbóli végrehajtását.

Nyilvánvaló, hogy a felhasználói műveletek támogatása az egész alkalmazást át meg átszövi. A kihívást az jelenti, hogy olyan megoldással kell előrukkolnunk, ami az igények kielégítése mellett egyszerű és könnyen bővíthető.

Kérelem egységbe zárása

Tervezői szempontból a lenyitható menü is csak egy képjel, ami további képjeleket tartalmaz. Az különbözteti meg más, gyermekekkel rendelkező képjelektől, hogy a benne levők kattintásra valamilyen műveletet hajtanak végre.

Tegyük fel, hogy ezek a műveletvégző képjelek a Glyph **MenuItem** (MenüElem) alosztályának példányai, és működésüket egy ügyféltől[9] érkező kérelem váltja ki. A kérelem teljesítése járhat egyetlen objektumon végzett művelettel, több objektumon végzett több művelettel, vagy lehet valahol a kettő között.

Megtehetnénk, hogy minden felhasználói művelethez létrehozzuk a MenuItem egy alosztályát, majd ezekbe „bedrótozzuk" a kérelem végrehajtását biztosító kódot, de ez nem igazán jó megközelítés; semmivel sincs jobban szükségünk minden kérelemhez külön alosztályra, mint külön osztályokra egy lenyíló menü minden eleméhez. Ráadásul így a kérelmet egy adott felhasználói felülethez kötnénk, ami megnehezítené, hogy más módon is teljesíthessük a kérést.

Példaként tegyük fel, hogy a dokumentum utolsó oldalára szeretnénk ugrani, és erre kétféle módszert szeretnénk. Az egyik, hogy egy menüben megjelenő MenuItem-re kattintunk, a másik pedig, hogy a Lexi ablakának alján levő oldalikonra (ami rövidebb dokumentumoknál jóval kényelmesebb lehet). Ha a kérelmet öröklés útján a MenuItem-hez kapcsoljuk, akkor ugyanezt kell tennünk az oldalikon esetében is, sőt minden olyan vezérlő esetében, amellyel esetleg ugyanezt a műveletet kívánjuk végrehajtani. Ezzel az osztályok száma ugrásszerűen megnőhet, akár a vezérlőtípusok és kérelemtípusok számának szorzatáig.

Amivel nem rendelkezünk, az egy megoldás, amellyel a menüelemeknek paramétereket adhatnánk át, amelyekben megkapják a végrehajtandó kérelmet. Ezzel elkerülhetnénk az osztályok számának növelését, és nagyobb futásidejű rugalmasságot érhetnénk el. A MenuItem például kaphatna paraméterként egy függvényt, amit meghívhat, de ez három okból nem nyújtana tökéletes megoldást:

1. Nem oldhatnánk meg a visszavonás–ismétlés problémáját.
2. Állapotot függvénnyel összekapcsolni nehéz. Például egy függvénynek, ami megváltoztatja a betűtípust, tudnia kell, *melyik* betűtípus van érvényben.
3. A függvények nehezen bővíthetők, és részleteikben nehezen újrahasznosíthatók.

A fentiek azt sugallják, hogy a menüelemeknek nem függvényt, hanem *objektumot* kellene paraméterként átadnunk. Ezután már használhatunk öröklést a kérelem megvalósításának bővítéséhez, illetve újrahasznosításához, lesz helyünk, ahol tárolhatjuk az állapotot, és

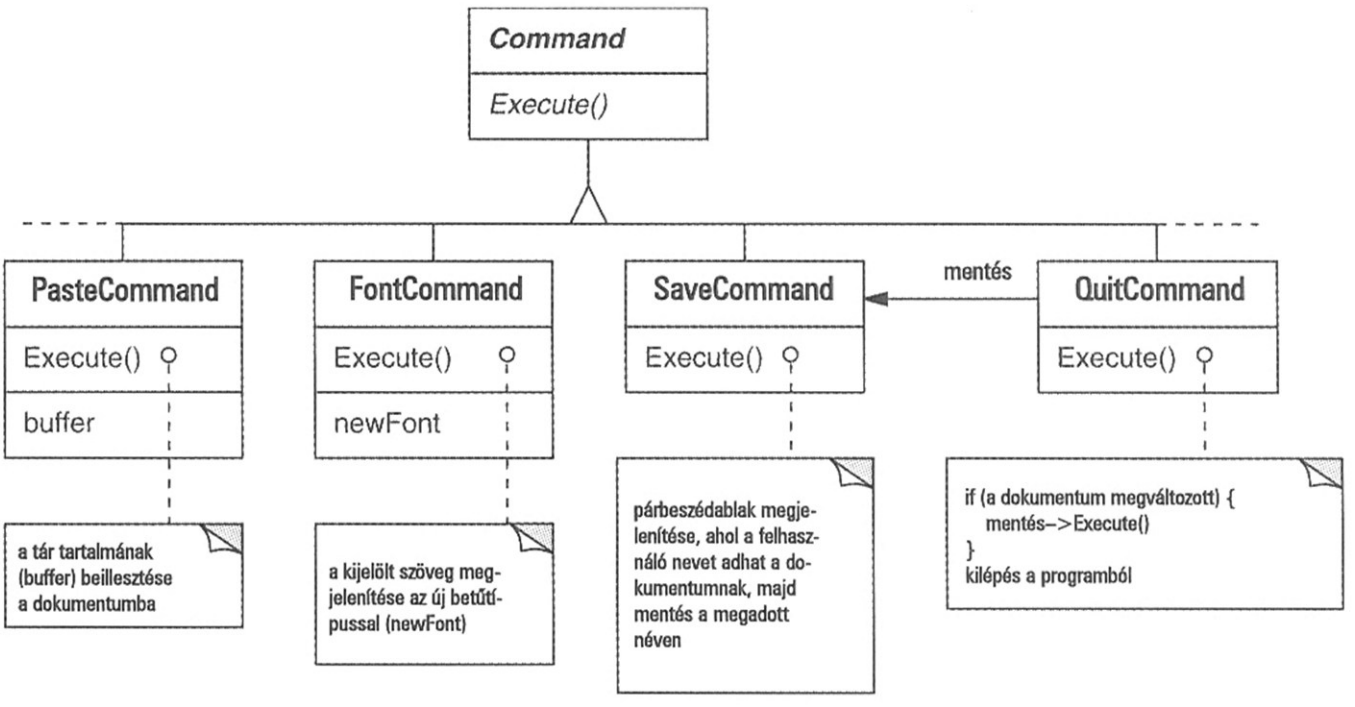
[9] Elvben az ügyfél a Lexi felhasználója, valójában azonban egy másik objektum (például egy eseménytovábbító), amely a felhasználói bemeneteket kezeli.

megvalósíthatjuk a visszavonást, illetve ismétlést biztosító műveleteket. Íme egy újabb példa a változó elem egységbe zárására, ami ebben az esetben egy kérelem. A kérelmeket egy-egy **parancs** objektumba zárjuk.

A Command osztály és alosztályai

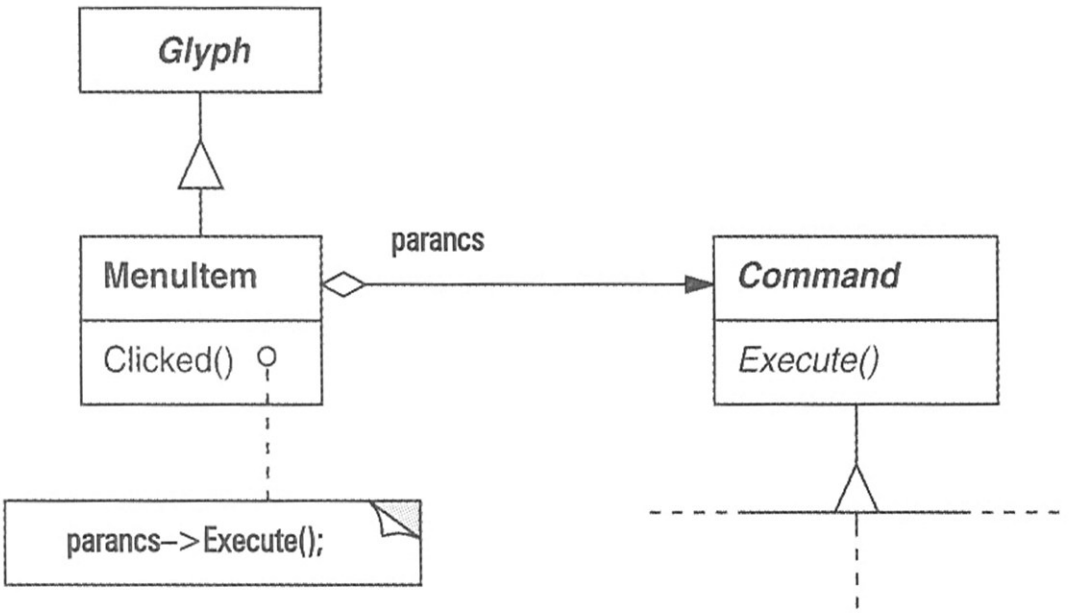
Először is, létrehozzuk a **Command** (Parancs) elvont osztályt, ami a kérelmek kiadásához biztosítja a felületet. Az alapfelület egyetlen elvont műveletből áll, amelyet Execute-nak (Végrehajt) nevezünk. A Command alosztályai a különböző kérelmek végrehajtásához különféle módokon valósítják meg az Execute műveletet. Egyes alosztályok munkájuk egy részét vagy egészét más objektumra ruházhatják át, mások képesek önmaguk kielégíteni a kérelmet (lásd a 2.11 ábrát). A kérelmező ugyanakkor nem lát különbséget a Command (Parancs) objektumok között, mindet egyformán kezeli.

A MenuItem tárolhatja a Command objektumot, ami a kérelmet egységbe zárja (2.12 ábra). Minden menüelem objektumnak a megfelelő Command alosztály egy példányát adjuk át, illetve meghatározzuk a szöveget, ami a menüelem helyén megjelenik. Amikor a felhasználó kiválaszt egy menüpontot, a MenuItem egyszerűen meghívja az Execute műveletet az általa tárolt Command objektumra, így teljesíti a kérelmet. A gombok és más vezérlők ugyanúgy használhatják a parancsokat, mint a menüelemek.



2.11 ábra

A Command osztályhierarchia részlete.



2.12 ábra
A Menultem és a Command kapcsolata.

Visszavonási lehetőség

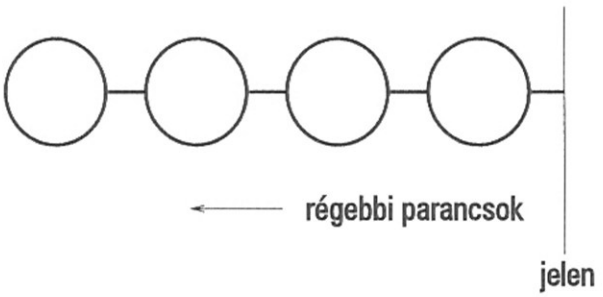
A visszavonás, illetve újbóli végrehajtás lehetősége igen fontos az interaktív alkalmazásokban. E parancsok végrehajtását a Command felülethez adott Unexecute (Ismét vagy Mégse) művelettel biztosítjuk. Az Unexecute megfordítja a megelőző Execute művelet hatását, mégpedig annak az információnak a felhasználásával, amelyet az Execute tárol. A FontCommand (BetűtípusParancs) esetében például az Execute a betűtípusváltás által érintett szövegrészt, illetve az eredeti betűtípust raktározza el, így az adott szövegrészletet az Unexecute visszaállíthatja korábbi állapotába.

A visszavonás lehetőségéről néha futás közben kell dönteni. A kijelölt szöveg betűtípusának megváltoztatására irányuló kérelem például nem eredményez semmit, ha a szöveg már eredetileg is az új betűtípussal íródott. Tegyük fel, hogy a felhasználó kijelöl egy részt, és kiad egy ilyen felesleges parancsot. Mi legyen a hatása az ezután kiadott Visszavonás parancsnak? Járjon egy értelmetlen változtatás egy ugyanolyan értelmetlen művelettel? Természetesen nem. Ha a felhasználó néhányszor megismétli a felesleges betűtípusváltási parancsot, nem szabad, hogy ugyanannyi visszavonási műveletre legyen szükség ahhoz, hogy visszajussunk az utolsó értelmes műveletig. Ha egy parancs végrehajtása semmilyen változást nem eredményez, egyáltalán nincs szükség visszavonási műveletre.

Így aztán ahhoz, hogy meghatározhassuk, hogy egy művelet visszavonható-e, a Command felülethez hozzáadjuk az elvont Reversible (Megfordítható) műveletet, amely egy logikai (Bool-féle) értéket ad vissza. Az alosztályok e művelet felülírásával adhatnak vissza igaz vagy hamis (true–false) értéket futásidőben.

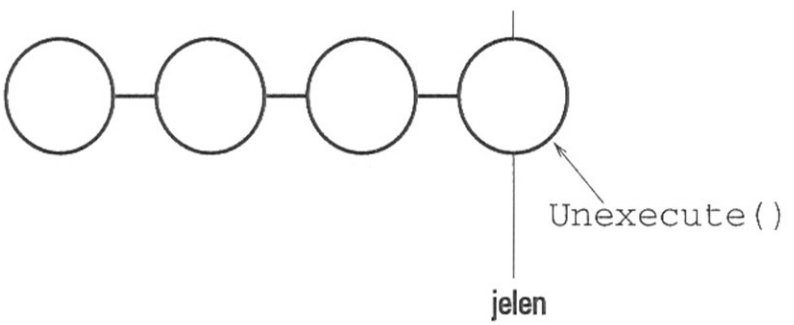
Parancselőzmények

A visszavonás–újbóli végrehajtás tetszőleges szintjének támogatásához az utolsó lépés a **parancselőzmények** listájának meghatározása, vagyis azon műveleteknek, amelyeket végrehajtottunk vagy visszavontunk. Elméletben a parancselőzmények listája valahogy így fest:

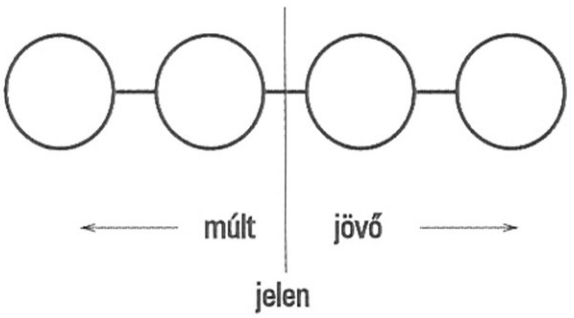


Mindegyik kör egy-egy Parancs (Command) objektumot jelképez. Ebben az esetben a felhasználó négy parancsot adott ki: a bal szélsőt először, a másodikat másodszor, és így tovább, míg a jobb szélső, vagyis az utoljára kiadott parancshoz nem érünk. A „jelen" felirattal ellátott vonal jelöli a legutolsó végrehajtott (vagy visszavont) parancsot.

Az utolsó parancs visszavonásához egyszerűen meghívjuk az Unexecute műveletet az utolsó parancsra:

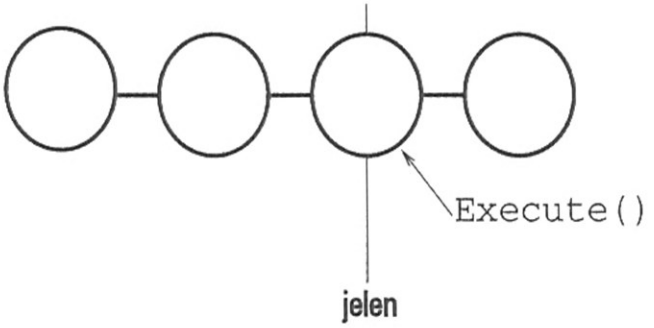


A visszavonás után a „jelen" vonalát eggyel balra toljuk. Ha a felhasználó úgy dönt, hogy még egy műveletet visszavon, a balra következő parancsra kerül sor, és az alábbi helyzet áll elő:

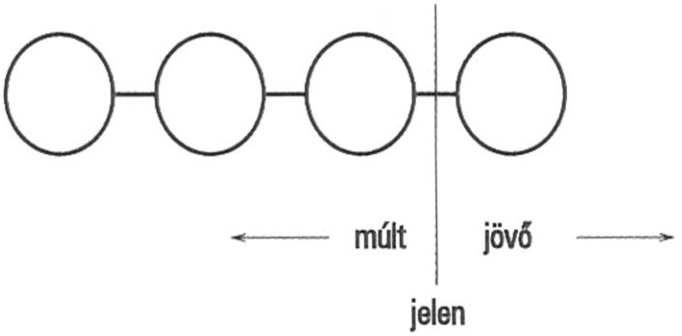


Látható, hogy ezen eljárás egyszerű ismétlésével többszintű visszavonásra nyílik lehetőség; a szintek számát csupán a parancselőzmények listájának hossza korlátozza.

Egy éppen visszavont parancs újbóli végrehajtásához ugyanezt az eljárást követjük, csak megfordítva. A jelen vonalától jobbra levő parancsok azok, amelyek visszavonása visszavonható. Az utoljára visszavont parancs újbóli végrehajtása úgy történik, hogy a jelen vonalától jobbra levő első parancsra meghívjuk az Execute műveletet:



Ezután a jelen vonalát jobbra visszük, hogy a legközelebbi Ismét művelet a következő visszavont művelet hatását álíthassa vissza:



Természetesen ha a következő művelet nem egy újabb Ismét, hanem Visszavonás, a jelen vonalától balra levő első parancsot fogjuk visszavonni. Így a felhasználó tetszőlegesen mozoghat a parancsok között, ha meggondolná magát vagy ki szeretne javítani egy hibát.

A Parancs tervezési minta

A Lexi parancsai a Parancs (Command) tervezési minta alkalmazását tükrözik, ami azt írja le, hogy zárhatunk egységbe egy kérelmet. A minta a kérelmek kiadásához egységes felületet követel meg, amely lehetővé teszi, hogy az ügyfelek különböző kérelmeket fogadhassanak, hiszen a felület elszigeteli őket a kérelem megvalósításától. A kérelem végrehajtását a parancs végezheti teljes egészében maga, de részben vagy egészben át is ruházhatja a megvalósítást más objektumokra. Ez a Lexihez hasonló alkalmazások számára tökéletes, mert így a program különböző részein elszórt szolgáltatásokhoz központi elérést biztosíthatunk. A minta a Command alapfelületre építve megoldást nyújt a visszavonás és újbóli végrehajtás problémájára is.

2.8 Helyesírás-ellenőrzés és elválasztás

Az utolsó bemutatandó tervezési probléma a szöveg elemzését igényli, mégpedig a helytelenül írt szavak megkeresését és a szebb formázás érdekében elválasztási pontok beszúrását.

Az elvárások itt hasonlók, mint a 2.3 részben bemutatott formázási probléma megoldásánál. Ahogy a sortörési módszerekből is több lehetett, a helyesírás ellenőrzése és az elválasztási pontok meghatározása is többféleképpen történhet, vagyis több algoritmus támogatására lesz szükség, így választhatunk majd, mi a fontosabb: a tárigény, a sebesség vagy a „minőség". Emellett azt is el kell érnünk, hogy később könnyen adhassunk a programhoz új algoritmusokat.

Az említett szolgáltatások merev bekódolása a dokumentumszerkezetbe kerülendő, még inkább, mint a formázás esetében, mert a helyesírás-ellenőrzés és elválasztás csupán kettő azon lehetséges elemző műveletek közül, melyek támogatását be szeretnénk építeni a Lexibe. Természetes, hogy idővel további ilyen irányú képességekkel szeretnénk bővíteni a programot, például keresési lehetőségekkel, a szavak megszámlálásával, számológéppel a táblázatos értékek beszúrásához, nyelvtani ellenőrzéssel és így tovább. A Glyph osztályt és alosztályait azonban nem szeretnénk minden alkalommal módosítani, amikor új szolgáltatást adunk az alkalmazáshoz.

A kirakósjátéknak tulajdonképpen két darabja van: (1) az elemzendő információ elérése, amely a dokumentumszerkezet képjeleiben szétszórva helyezkedik el, illetve (2) az elemzés végrehajtása. E két feladattal külön foglalkozunk.

Az elszórt információ elérése

Számos elemzéstípusnál a szöveg karakterről karakterre való átvizsgálására van szükség, az elemzendő szöveg pedig képjel objektumok hierarchikus szerkezetében szétszórva helyezkedik el. Egy ilyen felépítésű szöveg vizsgálata olyan elérési módszert igényel, amely „ismeri" azokat az adatszerkezeteket, amelyekben az objektumok tárolódnak. Egyes képjelek gyermekeiket láncolt listában tárolhatják, mások tömböket alkalmazhatnak, megint mások pedig még különlegesebb adatszerkezeteket. Elérési módszerünknek mindegyik lehetőséget támogatnia kell.

Tovább bonyolítja a helyzetet, hogy a különböző elemző műveletek más-más módon érik el az adatokat. A *legtöbb* elemző művelet elejétől végéig járja be a szöveget, de néhány ennek ellenkezőjét teszi – fordított keresést alkalmaz, például hátulról előre halad a szövegben. Az algebrai kifejezések kiértékelése például sokszor úgynevezett balról jobbra haladó bejárást (inorder bejárás, bal részfa–gyökér–jobb részfa sorrend) igényel.

Tehát az elérési módszernek több adatszerkezettel kell működnie, és a bejárás különböző típusait – előre haladó, visszafelé haladó, balról jobbra haladó – is támogatnunk kell.

Az elérés és bejárás egységbe zárása

Pillanatnyilag képjelfelületünk egész sorszámokkal teszi lehetővé az ügyfeleknek, hogy a gyermekekre hivatkozzanak. Bár ez ésszerűnek tűnhet a gyermekeiket egy tömbben tároló képjelosztályoknál, azoknál nem túl hatékony, amelyek láncolt listát használnak erre. Az elvont képjel-ábrázolás egyik fontos feladata, hogy elrejtse, milyen adatszerkezeteket használ az adott osztály a gyermekek tárolására, hiszen így anélkül változtathatjuk meg az adatszerkezetet, hogy ez más osztályokat érintene. Ennélfogva csak a képjel tudja, milyen adatszerkezetet is használ. Nyilvánvaló, hogy felületének nem szabad egyik szerkezetet sem előnyben részesítenie a többivel szemben, tehát a jelenlegi helyzet, amikor is jobban illeszkedik a tömbökhöz, mint például a láncolt listákhoz, nem előnyös.

E problémát és a bejárások különböző fajtáinak támogatását egyszerre oldhatjuk meg. Közvetlenül a képjelosztályokban elhelyezhetünk több elérési és bejárási módszert, és módot adhatunk az azok közötti választásra, mondjuk egy felsoroló típusú állandót paraméterként átadva. Az osztályok aztán ezt a paramétert továbbadhatják egymásnak, hogy biztosítsák, mindannyian ugyanazt a bejárási módszert használják. Emellett tovább kell adniuk a bejárás közben megszerzett valamennyi információt is. A fenti megközelítést támogatandó a következő elvont műveleteket adhatjuk a Glyph felületéhez:

```
void First (Traversal kind)
void Next ()
bool IsDone ()
Glyph* GetCurrent ()
void Insert(Glyph*)
```

A bejárást a `First` (Első), `Next` (Következő) és `IsDone` (Kész) műveletek vezérlik. A `First` indítja el, miután a felsoroló állandó típusú `Traversal` (Bejárás) paraméterben megkapja a bejárás módját (`kind`). A paraméter értéke `CHILDREN`, `PREORDER`, `POSTORDER` és `INORDER` lehet. Az első esetében csak a képjel közvetlen gyermekeit járjuk be, a másodiknál a teljes szerkezetet, előrefelé haladva. A `Next` a következő képjelre ugrik, az `IsDone` pedig jelentést ad, hogy a bejárás véget ért-e vagy sem. A `Child` (Gyermek) műveletet a `GetCurrent` (ElérAktuális) váltja fel, amely a bejárás során éppen érintett képjelet éri el. Az `Insert` (Beszúr) a korábbi, hasonló nevű műveletet cseréli le és beszúrja az adott képjelet az aktuális helyre. Az elemzés során a következő C++ kódot használhatnánk egy `g` gyökerű képjelszerkezet előre haladó bejárásához:

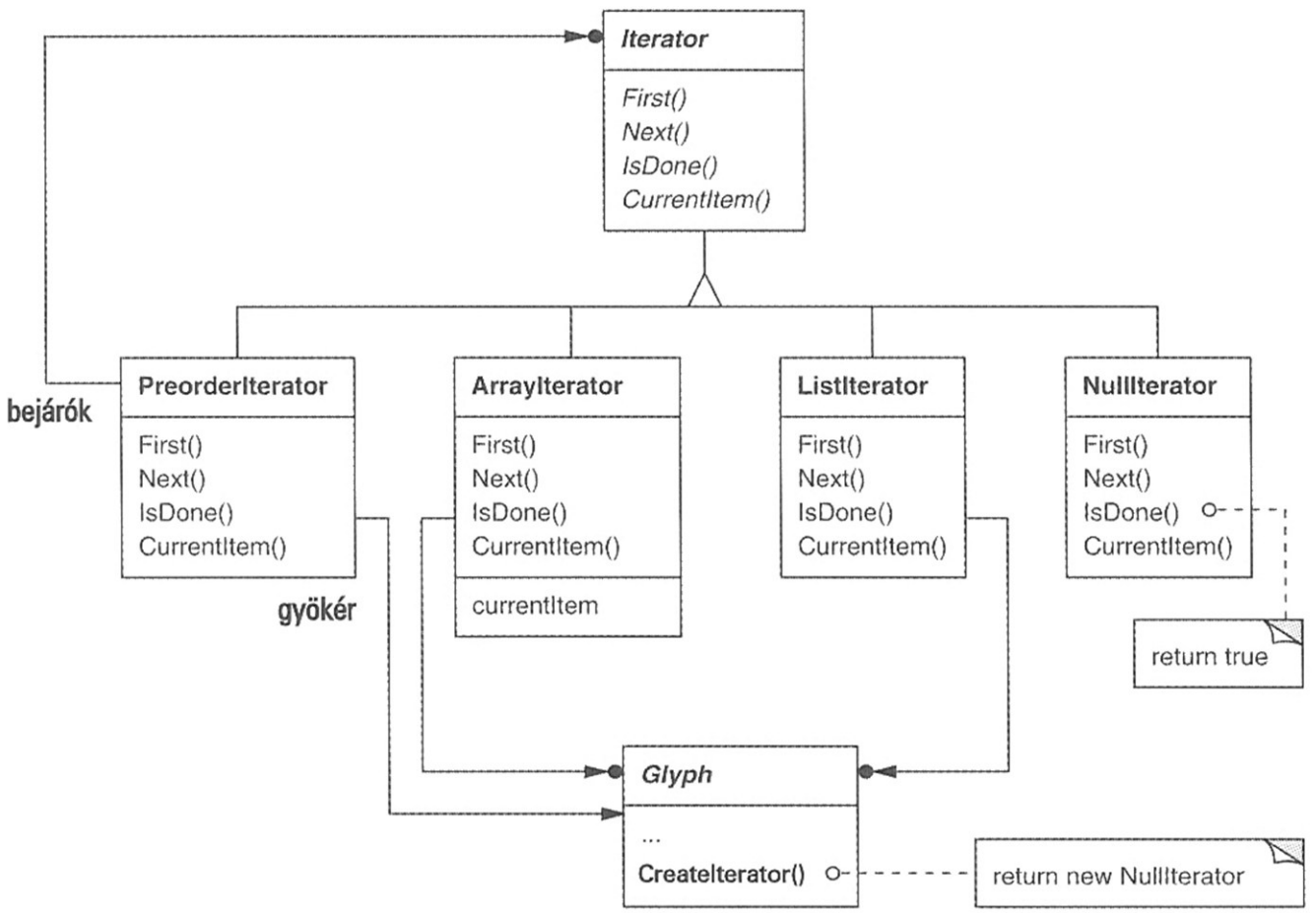
```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
   Glyph* current = g->GetCurrent();

   // valamilyen elemzés
}
```

Megfigyelhetjük, hogy az egész számokkal való sorszámozást kigyomláltuk a képjel felületéből, így az nem tolódik el többé egyik gyűjteménytípus támogatása felé sem. Attól is megkíméltük az ügyfeleket, hogy maguknak kelljen megvalósítaniuk az általános bejárástípusokat.

A megközelítéssel azonban továbbra is vannak gondok. Először is, nem képes támogatni új bejárási módokat anélkül, hogy új értékekkel egészítenénk ki a felsoroló típust vagy új műveleteket írnánk. Tegyük fel, hogy az előre haladó bejárás egy olyan változatát szeretnénk, ami automatikusan átugorja a nem szöveget jelölő képjeleket. Ekkor a `Traversal` felsorolást módosítanunk kellene, hogy tartalmazzon egy `TEXTUAL_PREORDER` (SZÖVEGES_ELŐRE) vagy hasonló értéket.

A meglevő deklarációk megváltoztatását el szeretnénk kerülni. Ha a bejárási módszert teljes egészében a Glyph osztályhierarchiába helyezzük, módosításához vagy bővítéséhez valószínűleg számos osztályt meg kell változtatnunk, az is nehézzé válik, hogy más típusú objektumszerkezetek bejárásához újrahasznosítsuk, ráadásul csak egy folyamatban levő bejárás lehetséges.

Ismét csak jobb megoldás a változó tényező egységbe zárása, ez pedig ebben az esetben az elérési és bejárási módszer. Ezért bevezetjük a Bejáró (Iterator) objektumok osztályát, amelynek egyetlen feladata a különböző bejárási módszerek meghatározása. Öröklés segítségével egységesítjük a különböző adatszerkezetek elérését, és támogatjuk az új bejárási módszereket is. Így nem kell megváltoztatnunk a képjelfelületeket vagy a képjelek megvalósításait terhelnünk ezzel a feladattal.



2.13 ábra

Az Iterator osztály és alosztályai.

Az Iterator osztály és alosztályai

Az elérés és bejárás közös felületét az **Iterator** (Bejáró) nevű elvont osztály határozza meg. A konkrét alosztályok közül az **ArrayIterator** (TömbBejáró) és a **ListIterator** (ListaBejáró) e felület megvalósításával a tömbök és listák elérését biztosítják, míg a **PreorderIterator**, a **PostorderIterator** és hasonlók az egyes szerkezetek bejárását. Mindegyik Iterator alosztály egy hivatkozást tartalmaz arra a szerkezetre, amit bejár; példányaik létrehozásukkor ezt a hivatkozást kapják kezdőértékül. A 2.13 ábra az Iterator osztályt és néhány alosztályát mutatja. Megfigyelhető rajta, hogy a bejárók támogatásához a Glyph osztályhoz adtunk egy CreateIterator (LétrehozBejáró) nevű elvont műveletet.

A bejárás vezérléséhez az Iterator felület a First, Next és IsDone műveleteket biztosítja. A ListIterator osztály First-megvalósítása a lista első elemére mutat, a Next a listában következő elemre lép, az IsDone pedig azt adja vissza, hogy a listamutató az utolsó elemen túlra mutat-e. A CurrentItem (AktuálisElem) a bejáró hivatkozásának követésével a hivatkozott képjelet adja vissza. Az ArrayIterator osztály feladata hasonló, de képjelek tömbjére vonatkozik.

A képjelszerkezet gyermekeit most már anélkül érhetjük el, hogy ismernénk a szerkezet ábrázolását:

```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();

for (i->First(); !i->IsDone(); i->Next()) {
   Glyph* child = i->CurrentItem();

// csinálunk valamit az aktuális gyermekkel
}
```

A CreateIterator alapértelmezés szerint egy NullIterator (NullBejáró) példányt ad vissza. A NullIterator egy csökevényes bejáró a gyermekkel nem rendelkező képjelek, vagyis a levél képjelek számára. A NullIterator IsDone művelete mindig igazat ad vissza.

A gyermekkel rendelkező képjel-alosztályok felülbírálják a CreateIteratort, így egy másik Iterator alosztály egy példányát adják vissza. Az, hogy *melyiket*, attól függ, milyen szerkezet tárolja a gyermekeket. Ha a Glyph Row alosztálya gyermekeit például a `_children` nevű listában tárolja, CreateIterator művelete az alábbihoz hasonlóan fog festeni:

```
Iterator<Glyph*>* Row::CreateIterator () {
   return new ListIterator<Glyph*>(_children);
}
```

Az előre haladó és balról jobbra haladó (inorder) bejárások képjelfüggő bejárókkal dolgoznak. E bejárókat a bejárandó szerkezet gyökerében levő képjel adja meg. A bejárók meghívják a CreateIteratort a szerkezetben levő képjelekre és az eredményként kapott bejárókat egy verem segítségével követik nyomon.

A PreorderIterator osztály például megszerzi a bejárót a gyökérképjeltől, beállítja, hogy az első elemére mutasson, majd a verem tetejére helyezi:

```
void PreorderIterator::First () {
   Iterator<Glyph*>* i = _root->CreateIterator();

    if (i) {
          i->First();
          _iterators.RemoveAll();
          _iterators.Push(i);
    }
}
```

A CurrentItem egyszerűen meghívja a CurrentItem-et a verem tetején levő bejáróra:

```
Glyph* PreorderIterator::CurrentItem () const {
   return
          iterators.Size() > 0 ?
          iterators.Top()->CurrentItem() : 0;
}
```

A `Next` művelet veszi a verem tetején levő bejárót és első elemét egy bejáró létrehozására utasítja, hogy olyan messzire nyúlhasson a képjelszerkezetben, amennyire csak lehet (végül is előre haladó bejárásról van szó). A `Next` az új bejárót a bejárandó szerkezet első elemére állítja, majd a verem tetejére helyezi. Ezután a `Next` ellenőrzi a legutóbbi bejárót: ha `IsDone` művelete igazat ad vissza, az aktuális részfa (vagy levél) bejárását befejezettnek tekinthetjük. Ebben az esetben a `Next` leveszi a verem tetején levő bejárót és addig ismétli a folyamatot, amíg csak teljesen be nem járt szerkezeteket talál. Ha elfogytak, a bejárást befejeztük.

```
void PreorderIterator::Next () {
   Iterator<Glyph*>* i =
          _iterators.Top()->CurrentItem()->CreateIterator();

   i->First();
   _iterators.Push(i);

    while (
          _iterators.Size() > 0 && _iterators.Top()->IsDone()
    ) {
          delete _iterators.Pop();
          _iterators.Top()->Next();
   }
}
```

Figyeljük meg, hogy az Iterator osztályhierarchia hogyan teszi lehetővé, hogy új fajta bejárásokat adjunk meg, anélkül, hogy a képjelosztályokat módosítanunk kellene – egyszerűen új Iterator alosztályokat hozunk létre, majd megadjuk az új bejárási módot, mint

a `PreorderIterator`-nél. A képjel-alosztályok ugyanazt a felületet használják, hogy az ügyfelek a gyermekeket a tárolásukra használt adatszerkezetek felfedése nélkül érhessék el. Miután a bejárók a bejárás állapotát maguk is tárolják, egyszerre több bejárást végezhetünk, akár ugyanazon a szerkezeten. Emellett – bár a példában képjelszerkezeteket jártunk be – a `PreorderIterator`-hoz hasonló osztályoknak paraméterként átadhatjuk az objektumok típusát, így a kódot újrahasznosítva más szerkezeteket is bejárhatunk. A C++-ban erre sablonokat alkalmaznánk.

A Bejáró tervezési minta

A Bejáró (Iterator) minta az objektumszerkezetek elérését és bejárását támogatja. Nem csak összetett szerkezetek esetében alkalmazható, de gyűjteményekre is. Elvonatkoztatja a bejárási algoritmust és elrejti a bejárandó objektumok belső szerkezetét az ügyfelek elől. Vagyis, a Bejáró minta ismét csak arra példa, hogyan érhetünk el nagyobb rugalmasságot és újrahasznosíthatóságot a változó elem egységbe zárásával. A bejárás kérdésköre persze igen bonyolult, de a minta szerencsére az itt bemutatottnál sokkal több mindenre használható.

Bejárás vagy bejáró műveletek?

Most, hogy rendelkezünk egy módszerrel a képjelszerkezet bejárására, ellenőrizhetjük a helyesírást és az elválasztást. Mindkét elemzés információk gyűjtését igényli a bejárás közben.

Először is el kell döntenünk, hová helyezzük az elemzésért felelős kódot. Tehetnénk például az Iterator osztályokba, így az elemzést a bejárás szerves részévé tennénk. Ennél azonban nagyobb rugalmasságot és újrahasznosíthatóságot érhetünk el, ha a bejárást és a bejárás közben végzett műveleteket elválasztjuk egymástól, már csak azért is, mert különböző elemző műveletek igényelhetik ugyanazt a fajta bejárást; ennélfogva ugyanazt a bejáróhalmazt különféle elemzésekhez használhatjuk. Az előre haladó bejárás például számos elemzés alapja: ilyen a helyesírás-ellenőrzés, az elválasztás, az előre haladó keresés vagy a szavak megszámlálása.

Tehát az elemzés és a bejárás legyen egymástól független. De akkor kire ruházzuk az elemzés felelősségét? Tudjuk, hogy többféle elemzést is végezni szeretnénk, melyek mindegyike a bejárás más-más pontján hajt végre műveleteket. Az elemzéstől függően egyes képjelek fontosabbak lehetnek másoknál. Ha a helyesírást ellenőrizzük vagy szavakat választunk el, a karakter képjeleket szeretnénk figyelembe venni, és nem a grafikusakat, amilyenek a sorok vagy a bitképek. Ha szín alapján válogatunk, nyilván a látható képjelekre, és nem a láthatatlanokra lesz szükségünk. A különböző elemzések során tehát más-más képjeleket elemzünk.

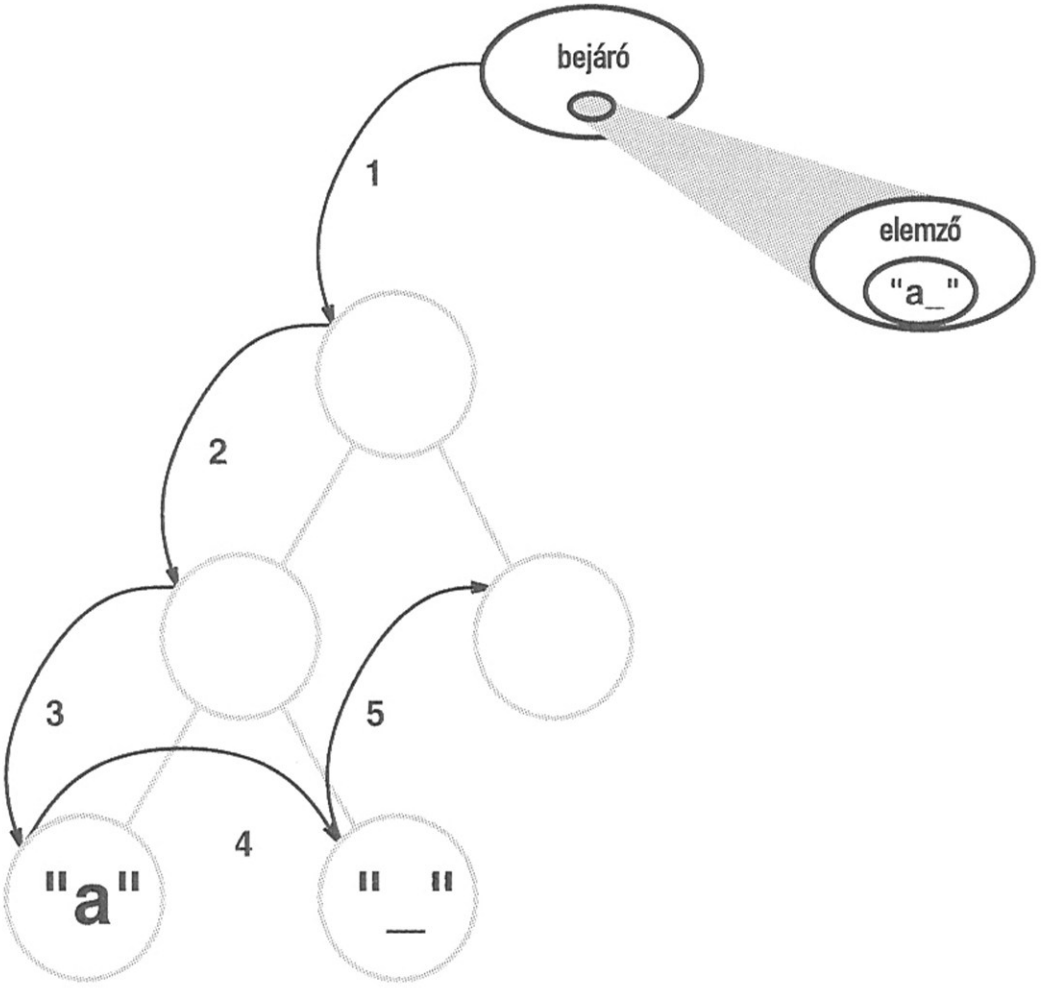
Egy adott elemzésnek nyilvánvalóan különbséget kell tudnia tenni a különböző képjeltípusok között. Így kézenfekvőnek látszik, hogy az elemző képességgel magukat a képjelosztályokat bővítsük. Mondjuk minden elemzéstípushoz egy vagy több elvont műveletet határozunk meg a Glyph osztályban, az alosztályok pedig ezeket az elemzésben játszott szerepüktől függően valósítják meg.

Ezzel a megközelítéssel azonban az a gond, hogy minden új elemzéstípus esetében módosítanunk kell az összes képjelosztályt. Egyes esetekben ezen enyhíthetünk: például ha az elemzésben csak kevés osztály vesz részt, vagy ha az elemzést a legtöbb osztály azonos módon végzi, az elvont művelet egy alapértelmezett megvalósítását megadhatjuk a Glyph osztályban. Ez lefedné az alapeseteket, és szükség esetén csak a Glyph osztályt kellene módosítani, illetve azokat az alosztályokat, amelyek eltérnek az alapértelmezett viselkedéstől.

De még ha az alapértelmezett megvalósítás csökkenti is a szükséges változtatások számát, marad egy gond: a Glyph felülete minden hozzáadott elemzési képességgel nő, ezek pedig idővel elfedik a Glyph alapfelületét. Nehéz lesz meghatározni, mi is a képjel eredeti célja: a megjeleníthető, alakkal rendelkező objektumok meghatározása és rendszerezése. A felület elveszik a „zajban".

Az elemzés egységbe zárása

Úgy tűnik tehát, az elemzést mégis csak külön objektumba kell zárnunk, ahogy eddig is tettük. Így egy adott elemzés kódja saját osztályba kerül, és ennek az osztálynak egy példányát használjuk majd a megfelelő bejáróval, ami a példányt a szerkezet egyes képjeleihez „szállítja", ahol az elemző objektum a bejárás pontjain elvégzi a munkáját. A bejárás előrehaladtával az elemző információt gyűjt (ebben az esetben karaktereket):



Az alapvető kérdés ezzel a megközelítéssel kapcsolatban az, hogyan különbözteti meg az elemző a különféle képjeleket anélkül, hogy típusellenőrzést vagy -átalakítást végezne. Az alábbihoz hasonló (ál)kód megjelenését a `SpellingChecker` (Helyesírás-ellenőrző) osztályban el szeretnénk kerülni:

```
void SpellingChecker::Check (Glyph* glyph) {
   Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
          // a karakter elemzése

    } else if (r = dynamic_cast<Row*>(glyph)) {
    // előkészület r gyermekeinek elemzésére

    } else if (i = dynamic_cast<Image*>(glyph)) {
    // semmit nem csinálunk
    }
}
```

Ez meglehetősen csúnya kód, ami olyan kerülendő műveleteken alapul, mint a típusbiztos átalakítás (dynamic cast), ráadásul nehezen bővíthető és a Glyph osztályhierarchia módosítása esetén a fenti függvény törzsét is módosítanunk kell. Pontosan az a fajta kód, aminek elkerülésére az objektumközpontú nyelveket megalkották.

Tehát a nyers erőfitogtatás kerülendő – de hogyan? Nézzük, mi történik, ha a Glyph osztályhoz a következő elvont műveletet adjuk:

```
void CheckMe (SpellingChecker&)
```

A `CheckMe`-t (EllenőrizEngem) a Glyph valamennyi alosztályában a következőképpen határozzuk meg:

```
void GlyphAlosztaly::CheckMe (SpellingChecker& checker) {
   checker.CheckGlyphAlosztaly(this);
}
```

A fenti kódban a `GlyphAlosztaly` helyére a megfelelő alosztály neve írandó. Ez azt jelenti, hogy amikor a `CheckMe` hívására sor kerül, az adott alosztály ismert – végül is az egyik műveletében vagyunk. A SpellingChecker osztályfelület cserébe minden Glyph alosztályhoz[10] tartalmaz egy `CheckGlyphAlosztaly` vagy hasonló műveletet.

[10] Ha függvény-túlterhelést alkalmazunk, ezen tagfüggvények mindegyikének ugyanazt a nevet adhatjuk, hiszen a paramétereik úgyis megkülönböztetik őket. Itt azért kaptak különböző nevet, hogy különbségeiket hangsúlyozzuk.

```
class SpellingChecker {
public:
    SpellingChecker();

    virtual void CheckCharacter(Character*);
    virtual void CheckRow(Row*);
    virtual void CheckImage(Image*);

    // ...és így tovább

    List<char*>& GetMisspellings();
protected:
    virtual bool IsMisspelled(const char*);

private:
   char _currentWord[MAX_WORD_SIZE];
   List<char*> _misspellings;
};
```

A `SpellingChecker` ellenőrző művelete a `Character` képjelek esetében valahogy így festhet:

```
void SpellingChecker::CheckCharacter (Character* c) {
   const char ch = c->GetCharCode();

   if (isalpha(ch)) {
      // betűkarakter hozzáfűzése a _currentWord-höz

   } else {
      // a karakter nem betű

      if (IsMisspelled(_currentWord)) {
         // a _currentWord hozzáadása a _misspellings-hez
         _misspellings.Append(strdup(_currentWord));
      }

      _currentWord[0] = '\0';
        // a _currentWord visszaállítása a következő szó vizsgálatához
   }
}
```

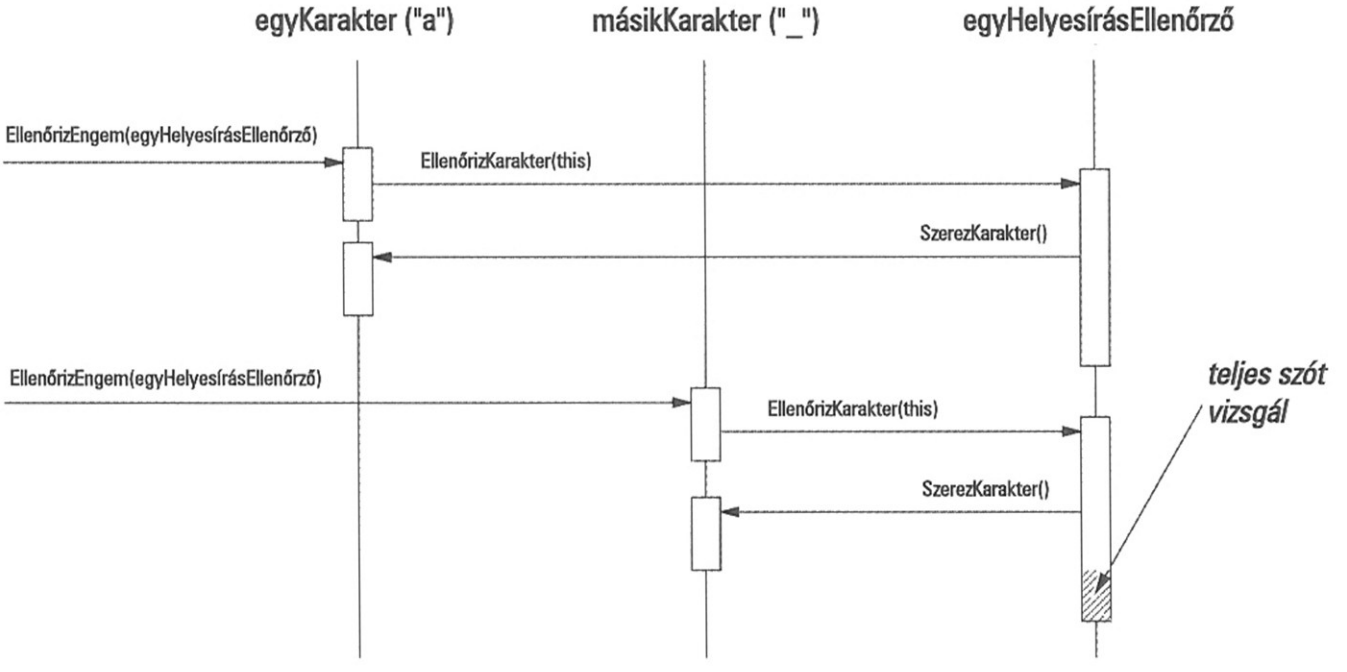
Észrevehetjük, hogy a `GetCharCode` (LekérKarakterKód) művelet csak a `Character` osztállyal működik, így a helyesírás-ellenőrző a különböző alosztályokhoz kapcsolódó műveleteket anélkül végezheti el, hogy típusellenőrzést vagy -átalakítást kellene végeznie, vagyis az objektumokat egyedileg kezelheti.

A `CheckCharacter` (EllenőrizKarakter) a betűkaraktereket a `_currentWord` (_aktuálisSzó) átmeneti tárban gyűjti össze. Ha nem betű karakterrel, például egy aláhúzásjellel találkozik, az `IsMisspelled` (HibásanÍrt) művelet segítségével ellenőrzi a tárban levő szó helyesírását[11]. Ha a szót helytelenül írták, a `CheckCharacter` hozzáadja azt a rosszul írt szavak listájához, majd kiüríti a `_currentWord` tárat, hogy helyet csináljon a következő szónak. A bejárás végeztével a helytelenül írt szavakat a `GetMisspellings` (SzerezHelytelenek) művelettel kérdezhetjük le.

Most már bejárhatjuk a dokumentumot: meghívjuk a `CheckMe` műveletet minden képjelre, argumentumként pedig a helyesírás-ellenőrzőt adjuk át neki. Ez minden képjelet azonosít a SpellingChecker számára, és az ellenőrzőt előre lépteti:

```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);

    for (i.First(); !i.IsDone(); i.Next()){
          g = i.CurrentItem();
          g->CheckMe(spellingChecker);
    }
```

Az alábbi együttműködési diagram azt mutatja be, hogyan dolgoznak együtt a Karakter (`Character`) képjelek és a HelyesírásEllenőrző (`SpellingChecker`) objektum:



[11] Az IsMisspelled valósítja meg a helyesírás-ellenőrző algoritmust, amit itt nem részletezünk, hiszen függetlenítettük a Lexi felépítésétől. A SpellingChecker-ből alosztályokat létrehozva, vagy a Stratégia mintát (a 2.3 részben, a formázással kapcsolatban bemutatott módon) alkalmazva különböző algoritmusokat támogathatunk.

A módszer megfelel a helyesírási hibák megtalálására, de hogyan segíthet abban, hogy többféle elemzést támogassunk? Jelenleg úgy tűnik, a Glyph-et és alosztályait minden esetben ki kell egészítenünk egy `CheckMe(SpellingChecker&)`-hez hasonló művelettel, ha új elemzési módszert építünk a programba. Ez így is van, már ha ragaszkodunk ahhoz, hogy minden elemzéshez *önálló* osztály tartozzon. Azt viszont semmi nem indokolja, hogy ezen osztályoknak ne legyen *közös* felülete. Ha készítünk egy ilyet, kihasználhatjuk a többalakúság előnyeit, vagyis a `CheckMe(SpellingChecker&)`-hez hasonló, az elemzési módtól függő műveleteket egy olyan, elemzésfüggetlen művelettel válthatjuk fel, ami általánosabb paramétert vár.

A Visitor osztály és alosztályai

A **látogató** (visitor) kifejezéssel általánosságban olyan objektumok osztályára utalunk a könyvben, amelyek bejárás közben más objektumokat „látogatnak meg" és ott valamilyen műveletet végeznek[12]. Ebben az esetben egy olyan Látogató (`Visitor`) osztályt készítünk, ami a képjelek „meglátogatásának" elvont felületét határozza meg:

```
class Visitor {
public:
   virtual void VisitCharacter(Character*) { }
   virtual void VisitRow(Row*) { }
   virtual void VisitImage(Image*) { }

   // és így tovább...
};
```

A `Visitor` konkrét alosztályai különböző elemzéseket végeznek. A helyesírás-ellenőrzés elvégzésére például létrehozhatunk egy `SpellingCheckingVisitor` (HelyesírásEllenőrzőLátogató) alosztályt, az elválasztásra pedig egy `HyphenationVisitor` (ElválasztásLátogató) nevűt. A `SpellingCheckingVisitor` osztályt pontosan ugyanúgy valósítanánk meg, mint feljebb a `SpellingChecker`-t, azzal az eltéréssel, hogy a műveletek neve az általánosabb `Visitor` felületre utalnának, például a `CheckCharacter` a `VisitCharacter` (LátogatKarakter) nevet kapná.

Mivel a `CheckMe` azon látogatók számára, akik nem ellenőriznek semmit, nem megfelelő, a helyénvalóbb `Accept` (Elfogad) nevet adjuk neki. Argumentumának is meg kell változnia (`Visitor&`), hogy tükrözze, bármilyen látogatót elfogad. Így egy új elemzési módszer hozzáadásához csak létre kell hoznunk a `Visitor` egy új alosztályát – egyetlen képjelosztályhoz sem kell hozzányúlnunk. A jövőbeni kiegészítések támogatásához ezt az egyetlen műveletet kell hozzáadnunk a Glyph-hez és alosztályaihoz.

Már láttuk, hogyan működik a helyesírás-ellenőrzés. A `HyphenationVisitor`-ban a szöveggyűjtéshez hasonló megközelítést alkalmazunk, de amikor annak `VisitCharacter`

[12] A „meglátogat" kifejezés az „elemez" általánosabb változata, amit a hamarosan bemutatandó tervezési minta szóhasználatában alkalmazunk.

művelete megtalált egy teljes szót, kissé másképp működik. Nem ellenőrzi, hogy helyesen írták-e a szót, hanem egy elválasztási algoritmussal megállapítja benne a lehetséges elválasztási helyeket, ha vannak ilyenek. Ezután minden elválasztási ponton beszúr egy **elválasztó** képjelet az összetételbe. Az elválasztó képjelek a `Glyph` osztály `Discretionary` (Elválasztó) alosztályának példányai.

Az elválasztó képjelek kétféleképpen jelenhetnek meg, attól függően, hogy a sor utolsó karakteréről van-e szó. Ha igen, az elválasztó egy kötőjel formáját ölti, ha nem, rejtett elem lesz belőle. Az elválasztó szülője (egy Row, vagyis Sor objektum) ellenőrzésével állapítja meg, hogy az utolsó gyermek-e a sorban, és ezt az ellenőrzést mindig elvégzi, amikor meghívják, hogy rajzolja ki magát vagy számítsa ki a helyét. A formázási módszer az elválasztókat ugyanúgy kezeli, mint az üreshelyeket (térközöket), vagyis lehetséges sorzáró karakterekként. A következő diagram azt mutatja, hogyan jelenhet meg egy beágyazott elválasztó.

A Látogató tervezési minta

Amit fentebb vázoltunk, az egy példa a Látogató (Visitor) tervezési minta alkalmazására. E mintában a korábban bemutatott Látogató (Visitor) osztály és alosztályai játszanak kulcsszerepet. A minta lényege, hogy segítségével a képjelszerkezetek tetszőleges számú elemzését támogathatjuk, anélkül, hogy magukat a képjelosztályokat módosítanunk kellene. A látogatók másik hasznos szolgáltatása, hogy nem csak a fentebb bemutatott, összetétel típusú szerkezetekre alkalmazhatók, hanem *bármilyen* objektumszerkezetre, vagyis halmazokra, listákra, sőt, irányított körmentes gráfokra is. Ráadásul a látogató által meglátogatható osztályok nem feltétlenül kell, hogy egy közös szülőosztályon keresztül kapcsolatban álljanak egymással, vagyis a látogatók osztályhierarchiákon átívelve is használhatók.

A Látogató minta alkalmazása előtt egy fontos kérdést kell feltennnük magunknak: mely osztályhierarchiák változnak majd a leggyakrabban? A minta ugyanis akkor hajtja a legtöbb hasznot, ha a stabil osztályszerkezettel rendelkező objektumokon sokféle műveletet szeret-

nénk végrehajtani. Egy új látogató hozzáadása nem igényli, hogy hozzányúljunk ehhez az osztályszerkezethez, ami különösen akkor jön jól, ha a hierarchia nagy méretű. Amikor azonban a szerkezethez új alosztályokat adunk, minden látogató felületet frissítenünk kell, hogy tartalmazzák az új osztály `Visit...` (`Látogat...`) műveletét. Példánkban ez azt jelenti, hogy ha a `Glyph`-ből `Izé` néven egy új osztályt hozunk létre, a `Visitor`-t és annak alosztályait is módosítanunk kell, hogy tartalmazzák a `LátogatIzé` műveletet. Jelenlegi tervezési céljaink azonban inkább azt valószínűsítik, hogy a Lexit inkább új elemző műveletekkel, semmint új képjelekkel bővítjük majd, így a Látogató minta megfelel az igényeinknek.

2.9 Összefoglalás

A Lexi tervezése során nyolc tervezési mintát alkalmaztunk:

- az Összetétel mintát a dokumentum fizikai szerkezetének ábrázolására,
- a Stratégia mintát a különböző formázási algoritmusok támogatására,
- a Díszítő mintát a felhasználói felület finomítására,
- az Elvont gyár mintát többféle megjelenítési szabvány támogatására,
- a Híd mintát a különféle ablakkezelő rendszereken való futáshoz,
- a Parancs mintát a visszavonható felhasználói műveletekhez,
- a Bejáró mintát az objektumszerkezetek eléréséhez és bejárásához,
- valamint a Látogató mintát, ami tetszőleges számú elemzési képesség beépítését tette lehetővé anélkül, hogy a dokumentumszerkezet megvalósítását bonyolította volna.

Egyik tervezési minta alkalmazhatósága sem szorítkozik a Lexihez hasonló szövegszerkesztő programokra. A legtöbb alkalmazásban lehetőség nyílik akár többnek a használatára is, persze nem feltétlenül ugyanezeket a feladatokat megoldandó. Egy pénzügyi elemző programban jó szolgálatot tehet az Összetétel minta, a befektetési portfoliók összeállításánál; egy fordítóprogramban a Stratégia mintát alkalmazhatjuk, hogy különböző célrendszerekre más-más regiszterfoglalási módszert biztosítsunk; a grafikus felhasználói felületű alkalmazásokban pedig valószínűleg szükségünk lesz legalább a Díszítő és a Parancs mintára, mint ahogy az itt bemutatott szövegszerkesztő programban is.

Bár a Lexi kapcsán számos tervezési kérdés felmerült, ezeken kívül még sok létezik, amelyeket nem érintettünk. A könyvben azonban nem csak az említett nyolc tervezési mintát tárgyaljuk, és mindegyik ismertetésénél érdemes elgondolkodni azon, hogyan hasznosíthatnánk a Lexi esetében – vagy még inkább saját programjainkban.

A tervezési minták katalógusa

3

Létrehozási minták

A létrehozási minták a példányosítási folyamat elvont ábrázolásai. Segítenek függetleníteni a rendszert a benne lévő objektumok létrehozási módjától, összeállításától és megjelenítésétől. Az osztálylétrehozási minták öröklés útján módosítják azt az osztályt, amelyből példány készül, míg az objektum-létrehozási minták átruházzák a példányosítást egy másik objektumra.

A létrehozási minták akkor válnak fontossá, amikor a rendszer a fejlődés során elkezd jobban függni az objektumok összeállításától, mint az osztályörökléstől. Amikor ez megtörténik, a lényeg áthelyeződik a rögzített viselkedéshalmaz állandó jellegű bekódolásáról az alapvető viselkedésmódok olyan kisebb halmazának meghatározására, amely beépíthető akárhány összetettebb halmazba. Ezáltal adott viselkedésű objektumok létrehozásához többre lesz szükség puszta osztálypéldányosításnál.

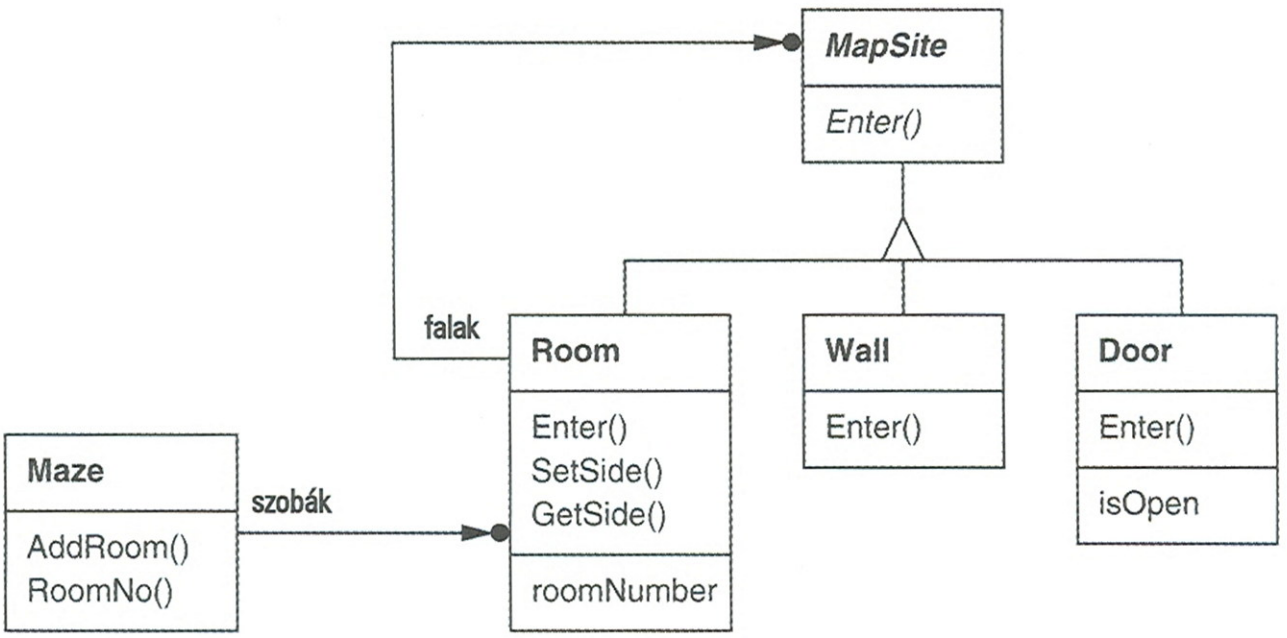
Ezen mintáknak két visszatérő ismérve van. Az első, hogy mindegyikbe be van ágyazva annak ismerete, hogy a rendszer valójában pontosan mely osztályokat használja, a másik, hogy elrejtik, hogy ezen osztályok példányait hogyan hozza létre és rakja össze a rendszer. Nagyjából minden, amit a rendszer ezekről az objektumokról tud, az, hogy milyen felületet határoznak meg hozzájuk az elvont osztályok. Ennek következtében a létrehozási minták nagy rugalmasságot tesznek lehetővé annak terén, hogy *mit* hozunk létre, *ki* hozza azt létre, *hogyan* és *mikor*. Így kialakítható egy olyan rendszer, amely rendkívül változatos felépítésű és szerepű objektumokat „termel". A kialakítás lehet statikus (azaz fordítási időben megadott) vagy dinamikus (futásidőben megadott).

A létrehozási osztályok néha versenyeznek egymással. Vannak például olyan esetek, amikor a Prototípus vagy az Elvont gyár gazdaságosan használható, máskor viszont kiegészítenek más osztályokat: az Építő más mintákat felhasználva valósíthatja meg, mely összetevők legyenek elkészítve. A Prototípus minta saját megvalósításához az Egyke mintát használhatja fel.

Mivel a létrehozási minták rendkívül közeli kapcsolatban állnak egymással, mind az ötöt együtt tanulmányozzuk, hogy megtudjuk, miben hasonlítanak, és miben különböznek. Megvalósításuk szemléltetésére egy közös példát adunk – egy számítógépes játékhoz hozunk létre egy labirintust. A labirintus és a játék mindegyik mintánál más lesz egy kicsit. Néha csak az lesz a feladat, hogy megtaláljuk a kiutat a labirintusból, ekkor a játékos valószínűleg csak a labirintus helyi képét fogja látni. Néha a labirintuson belül feladatokat is meg kell oldani és veszélyeket kell leküzdeni, és ezeknél a játékoknál a labirintus egy részének térképe is megtekinthető lesz.

Számos olyan részletet figyelmen kívül fogunk hagyni, hogy mi lehet a labirintuson belül, és hogy a labirintusjátékot egy vagy több játékos játszhatja-e. Ehelyett arra összpontosítunk, hogyan kell létrehozni a labirintust, amelyet „szobák" halmazaként alakítunk ki. Mindegyik szoba tudja, hogy kik veszik körül, kik a szomszédai. A lehetséges szomszédok: egy másik szoba, egy fal, vagy egy ajtó egy másik szobába.

A `Room` (Szoba), a `Door` (Ajtó) és a `Wall` (Fal) osztály határozza meg a labirintus alkotóelemeit minden példában. Az osztályoknak csak azon részeit határozzuk meg, amelyek fontosak a labirintus létrehozása szempontjából, és figyelmen kívül hagyjuk a játékosokat, a labirintusban való mozgáshoz és az ott látottak megjelenítéséhez szükséges műveleteket és más olyan, egyébként fontos szolgáltatásokat, amelyek magának a labirintusnak az elkészítéséhez nem létfontosságúak. Az alábbi ábra az osztályok közötti kapcsolatokat mutatja:



Mindegyik szobának négy oldala van. A C++ nyelven történő megvalósításkor a `Direction` felsorolás segítségével adjuk meg a szoba északi, déli, keleti és nyugati oldalát:

```
enum Direction {North, South, East, West};
```

A Smalltalk megvalósításban az irányok jelölésére a megfelelő jeleket kell használni.

A `MapSite` (HelyTérkép) osztály a labirintus összes alkotóelemének közös elvont osztálya. A példa egyszerűsítéséhez a `MapSite` osztály csak egy műveletet (`Enter`) határoz meg, amelynek jelentése attól függ, milyen adatot viszünk be. Ha szobát, akkor megváltozik a hely. Ha megpróbálunk belépni egy ajtón, két dolog történhet: ha az ajtó nyitva van, belépünk a következő szobába, ha pedig zárva, beverjük az orrunkat.

```
class MapSite {
public:
    virtual void Enter() = 0;
};
```

Az `Enter` egyszerű alapot ad a játék kifinomultabb műveleteihez. Ha például valamelyik szobában vagyunk, és a „Keletre!" parancsot adjuk ki, a játék egyszerűen meg tudja határozni, hogy melyik `MapSite` osztály esik közvetlenül keletre, és meghívja rá az `Enter` műveletet. Az alosztályfüggő `Enter` művelet kitalálja, hogy helyet változtattunk-e, vagy az orrunkat vertük be. Az igazi játékban az `Enter` argumentumként megkaphatná a játékos objektumot is, amely a labirintusban mozog.

A Room a `MapSite` osztály konkrét alosztálya, ez határozza meg a labirintus alkotóelemei közti főbb kapcsolatokat. A MapSite osztály többi objektumára való hivatkozásokat tartalmaz, és tárolja a szobaszámot. A labirintusban a szobákat a szobaszámmal lehet azonosítani.

```
class Room : public MapSite {
public:
    Room(int roomNo);

    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);

    virtual void Enter();

private:
    MapSite* _sides[4];      // falak
    int _roomNumber;
};
```

Az alábbi osztályok a szoba egyes oldalain található ajtókat, illetve falakat jelképezik.

```
class Wall : public MapSite {
public:
    Wall();

    virtual void Enter();
};
```

```
class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);

    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

Több dologról is tudnunk kell, nem csak a labirintus alkotóelemeiről, ezért meg kell határoznunk egy `Maze` (Labirintus) nevű osztályt is, amely a szobák összességét jelképezi. A `Maze` osztály képes egy adott szobát is megtalálni a szobaszám alapján, a `RoomNo` (SzobaSzám) művelet segítségével.

```
class Maze {
public:
    Maze();

    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    // ...
};
```

A `RoomNo` lineáris keresés, hasítótábla (kivonattábla), de akár egyszerű tömb használatával is tud keresni, de ilyen részletekkel most nem foglalkozunk. Ehelyett arra összpontosítunk, hogyan adhatjuk meg egy labirintusobjektum alkotóelemeit.

Egy másik általunk meghatározandó osztály a `MazeGame` (Labirintusjáték), amely magát a labirintust hozza létre. A labirintus létrehozásának egyik leglogikusabb módja, ha olyan műveletek sorával készítjük el, amelyek alkotóelemeket adnak a labirintushoz, majd összekapcsolják azokat. A következő tagfüggvény például egy két szobából és egy köztük lévő ajtóból álló labirintust hoz létre:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
```

```
    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Ez a függvény meglehetősen összetett, főleg ha azt is figyelembe vesszük, hogy semmi mást nem tesz, csak létrehoz egy kétszobás labirintust. Nyilvánvalóan van mód az egyszerűsítésére. A `Room` konstruktor például előkészítheti a szoba oldalait falakkal, de ezzel csak máshová kerül át a kód. Az igazi gond ezzel a tagfüggvénnyel nem a mérete, hanem a *rugalmatlansága*, mivel mereven kódolja be a labirintus elrendezését. Ha módosítani akarjuk a labirintus kialakítását, ezt a tagfüggvényt módosítanunk kell, akár felülbírálva azt (ami az egész újbóli megvalósítását jelenti), akár egyes részeit megváltoztatva (ami viszont növeli a hibákra való hajlamot, és nem teszi lehetővé az újrahasznosítást).

A létrehozási minták azt mutatják meg, hogyan lehet ezt az elrendezést *rugalmasabbá* – de nem feltétlenül kisebbé – tenni; nevezetesen megkönnyítik a labirintus elemeit meghatározó osztályok módosítását.

Tételezzük fel, hogy egy már létező labirintuselrendezést szeretnénk felhasználni egy új játékhoz, amely (többek között) elvarázsolt labirintusokat is tartalmaz. Az elvarázsolt labirintus játékban új elemek is vannak, például a `DoorNeedingSpell` (AjtóVarázsigével) egy olyan ajtó, amelyet csak varázsigével lehet bezárni, majd utána kinyitni, és az `EnchantedRoom` (ElvarázsoltSzoba) egy olyan szoba, amelyben szokatlan dolgok, például bűvös kulcsok vagy varázsigék vannak. Hogyan lehet egyszerűen módosítani a `CreateMaze` (LétrehozLabirintus) műveletet úgy, hogy olyan labirintust hozzon létre, amelyben megtalálhatók ezek az új objektumosztályok?

Ebben az esetben a módosítások legfőbb gátja a példányosítandó osztályok merev kódolása. A létrehozási minták különféle módszereket biztosítanak a kifejezett hivatkozások példányosító kódból konkrét osztályokba történő áthelyezésére:

- Ha a `CreateMaze` konstruktorok helyett virtuális függvényeket meghívva hozza létre a szükséges szobákat, falakat és ajtókat, akkor a `MazeGame` alosztályait létrehozva és ezeket a virtuális függvényeket felülírva módosíthatók a példányosított osztályok. Ezt a megközelítést mutatja be a Gyártófüggvény minta (Factory Method).
- Ha a `CreateMaze` paraméterként kapott valamilyen objektumot a szobák, falak és ajtók létrehozásához, akkor a szobák, falak és ajtók osztályait úgy lehet megváltoz-

tatni, ha egy másik paramétert adunk át a `CreateMaze` osztálynak. Erre jó példa az Elvont gyár (Abstract Factory) minta.

- Ha a `CreateMaze` olyan objektumot kapott, amely egy teljes új labirintust képes létrehozni, műveleteket használva a szobák, ajtók és falak hozzáadására a felépítendő labirintushoz, öröklés útján módosíthatjuk a labirintus elemeit vagy a labirintus felépítésének módját. Erre jó példa a Építő (Builder) minta.
- Ha a `CreateMaze` osztály paraméterei különféle prototípus jellegű szoba-, ajtó- és falobjektumok, amelyeket aztán lemásol, és úgy ad a labirintushoz, a labirintus felépítése ezen prototípus jellegű objektumok más objektumokkal való lecserélésével módosítható. Erre jó példa a Prototípus (Prototype) minta.

A fennmaradó létrehozási minta, az Egyke (Singleton) azt biztosíthatja, hogy játékonként csak egyetlen labirintus legyen, és a játék minden objektumának legyen hozzá olvasási engedélye anélkül, hogy globális változókhoz vagy függvényekhez kellene folyamodni, ezenkívül megkönnyíti a labirintus bővítését vagy lecserélését anélkül, hogy hozzá kellene nyúlni a már meglévő kódhoz.

Elvont gyár

Objektum-létrehozási minta

Cél

Kapcsolódó vagy egymástól függő objektumok családjának létrehozására szolgáló felületet biztosítani a konkrét osztályok megadása nélkül.

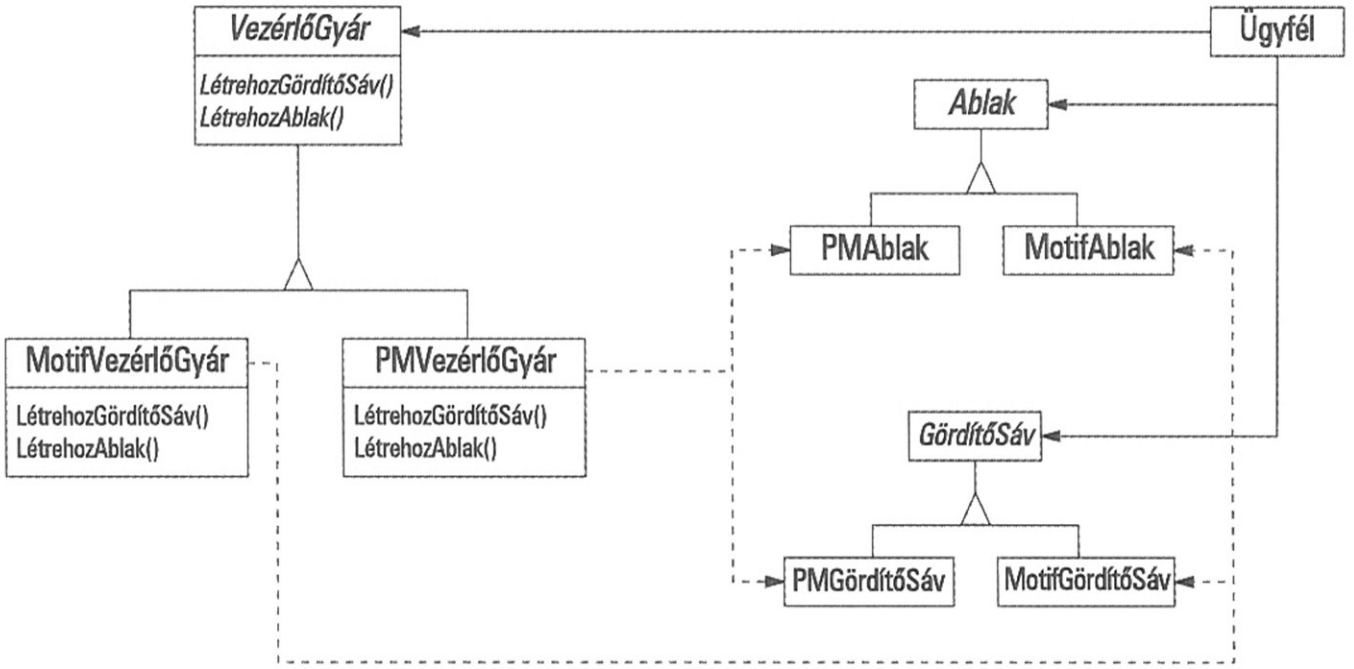
Egyéb nevek

Abstract Factory, Kit (Készlet)

Feladat

Vegyünk egy olyan felhasználói felületi elemkészletet, amely többféle megjelenítési szabványt támogat – amilyen például a Motif vagy a Presentation Manager. Ezekben máshogy néznek ki és viselkednek a felhasználói felület grafikus elemei (vezérlők, widget), például a gördítősávok, az ablakok és a gombok. Ha a különféle megjelenítési szabványok között hordozható alkalmazásokat szeretnénk készíteni, az alkalmazásokba nem lehet „bedrótozni" a felhasználói felületi elemeket, mert a kifejezetten egy adott szabványhoz tartozó elemosztályok példányosítása az alkalmazásban megnehezíti a megjelenítés későbbi megváltoztatását.

Ezt a problémát úgy oldhatjuk meg, ha olyan elvont VezérlőGyár (WidgetFactory) osztályt készítünk, amely egy felületet határoz meg minden alapvető elemfajta létrehozásához. Emellett minden elemfajtához is tartozik egy elvont osztály, az egyes megjelenítési szabványokhoz tartozó elemeket pedig konkrét alosztályok valósítják meg. A VezérlőGyár felületének van egy olyan művelete, amely minden elvont elemosztályhoz egy új vezérlőobjektumot ad vissza. Az ügyfelek ezeket a műveleteket meghívva kérhetnek elempéldányokat, de nem tudják, hogy pontosan mely konkrét osztályokat használják, így függetlenek maradnak az éppen használt megjelenítési szabványtól.



A VezérlőGyár osztály minden megjelenítési szabványhoz konkrét alosztállyal rendelkezik. Az alosztályok azokat a műveleteket valósítják meg, amelyek az adott megjelenítési szabványhoz szükséges felületelemek létrehozásához kellenek. Például a MotifVezérlőGyár (MotifWidgetFactory) alosztály LétrehozGördítőSáv (CreateScrollBar) művelete egy Motif gördítősávot példányosít és ad vissza, a megfelelő PMVezérlőGyár (PMWidgetFactory) alosztályon végzett művelet pedig egy Presentation Manager gördítősávot. Az ügyfelek kizárólag a VezérlőGyár felületen keresztül hoznak létre felületelemeket, és nem tudnak semmit azokról az osztályokról, amelyek a konkrét megjelenítés felületelemeit valósítják meg. Más szavakkal: az ügyfeleknek nincs más teendőjük, mint rábízni magukat egy olyan felületre, amelyet az elvont osztályok és nem a megfelelő konkrét osztályok hoznak létre.

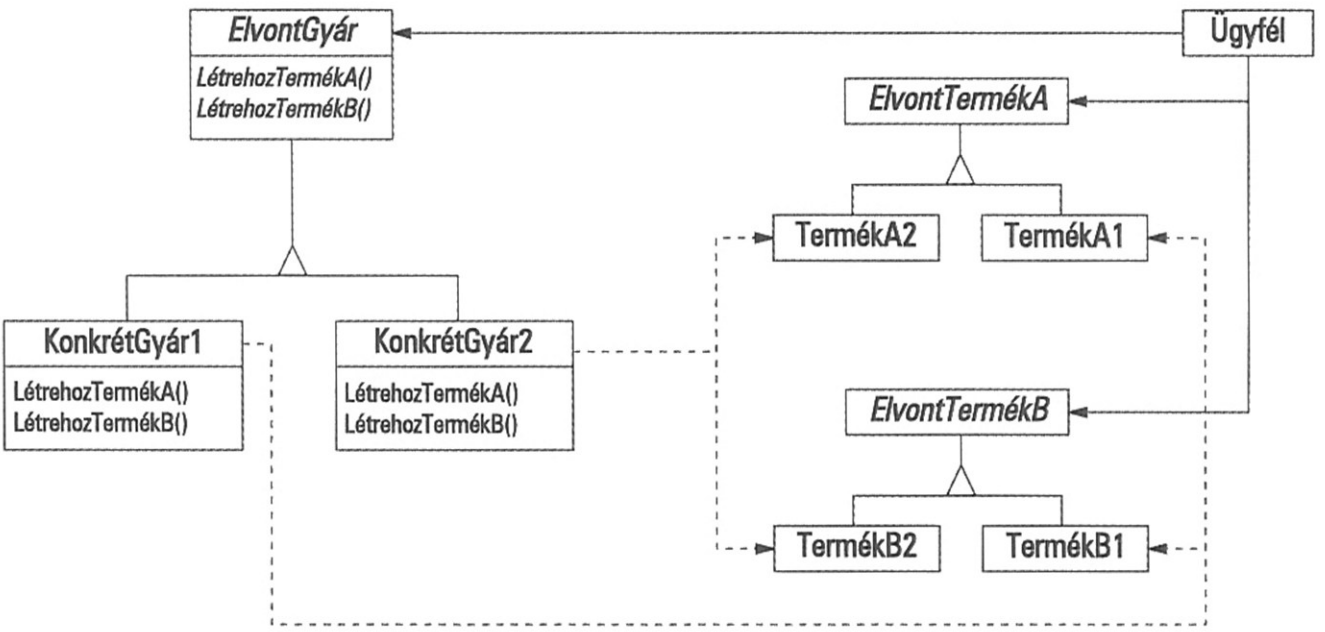
A VezérlőGyár függőségeket is kényszerít a konkrét vezérlőosztályokra. A Motif gördítősávok Motif gombbal és Motif szövegszerkesztővel használhatók, és ezt a megszorítást a MotifVezérlőGyár használatának következményeként automatikusan kikényszeríti a program.

Alkalmazhatóság

Az Elvont gyár minta az alábbi esetekben használható:

- A rendszernek függetlennek kell lennie attól, hogy „termékeit" hogyan hozza létre, állítja össze és jeleníti meg.
- A rendszert úgy kell beállítani, hogy a „termékcsaládok" egyikével használható legyen.
- Az egy családhoz tartozó termékobjektumokat együttes használatra tervezték, és ezt a megszorítást ki kell kényszerítenünk.
- Termékek osztálykönyvtárát szeretnénk létrehozni, de csak a felületeiket szeretnénk felfedni, megvalósításukat nem.

Szerkezet



Résztvevők

- ElvontGyár (VezérlőGyár)
  - Elvont termékobjektumokat létrehozó műveletek számára vezet be felületet.
- KonkrétGyár (MotifVezérlőGyár, PMVezérlőGyár)
  - Megvalósítja a konkrét termékobjektumokat létrehozó műveleteket.
- ElvontTermék (Ablak, GördítőSáv)
  - Egy adott típusú termékobjektumhoz hoz létre felületet.
- KonkrétTermék (MotifAblak, MotifGördítőSáv)
  - A megfelelő konkrét gyár (factory) által létrehozandó termékobjektumot határozza meg.
  - Megvalósítja az ElvontTermék felületet.

- Ügyfél
  - Csak az ElvontGyár vagy az ElvontTermék osztály által meghatározott felületeket használja.

Együttműködés

- Normál esetben a program futásidőben egy példányt hoz létre a KonkrétGyár osztályból. Ez a konkrét gyár olyan termékobjektumokat hoz létre, amelyeknek egy adott megvalósításuk van. Ha más termékobjektumokat szeretnénk létrehozni, az ügyfeleknek más konkrét gyárakat kell használniuk.
- Az ElvontGyár a KonkrétGyár alosztályra bízza a termékobjektumok létrehozását.

Következmények

Az Elvont gyár minta előnyei és hátrányai a következők:

1. *Elszigeteli a konkrét osztályokat.* Az Elvont gyár minta segíti az alkalmazások által létrehozott objektumosztályok kezelését. Mivel a gyár tartalmazza a termékobjektumok létrehozásának felelősségét és folyamatát, elszigeteli az ügyfeleket a megvalósítási osztályoktól. Az ügyfelek saját elvont felületükön át kezelik a példányokat. A termékosztálynevek el vannak szigetelve a konkrét gyár megvalósításában, és nem jelennek meg az ügyfél kódjában.
2. *Megkönnyíti a termékcsaládok cseréjét.* A konkrét gyár osztálya egy alkalmazásban csak egyszer jelenik meg – akkor, amikor példányosítják. Ez megkönnyíti az alkalmazás által használt konkrét gyár megváltoztatását. A különféle termékegyüttesek használatához nem kell mást tenni, mint megváltoztatni a konkrét gyárat. Mivel az elvont gyárak teljes termékcsaládokat állítanak elő, egyszerre megváltozik a teljes termékcsalád. A felhasználói felületes példában pusztán a megfelelő gyárobjektumok közötti átkapcsolással és a felületet újra létrehozva átválthatunk a Motif felületelemekről a Presentation Manager elemekre.
3. *Biztosítja a termékek közti egységességet.* Amikor egy családon belül együttműködésre tervezett termékobjektumok vannak, fontos, hogy az alkalmazások egyidőben csak egy termékcsaládba tartozó objektumokat használjanak. Az ElvontGyár leegyszerűsíti ennek kikényszerítését.
4. *Az új termékfajták támogatása nehézségekbe ütközik.* Az elvont gyárak kibővítése újfajta termékek létrehozására nem könnyű. Ennek az az oka, hogy az ElvontGyár felület rögzíti a létrehozható termékeket. Az újfajta termékek támogatásához ki kell bővíteni a gyár felületét, amihez meg kell változtatni az ElvontGyár osztályt és annak minden alosztályát. A probléma egyik megoldását a Megvalósítás rész ismerteti.

Megvalósítás

Az alábbiakban az Elvont gyár minta megvalósításának néhány módját mutatjuk be.

1. *Gyárak mint egykék.* Egy alkalmazásnak jellemzően csak egy KonkrétGyár példányra van szüksége termékcsaládonként, így azokat általában a legjobb Egyke formájában megvalósítani.
2. *A termékek létrehozása.* Az ElvontGyár csak a *felületet* adja meg, amelyre a termékek előállításához szükség van. A termékek konkrét létrehozását a KonkrétTermék alosztályok végzik. Ennek legáltalánosabb módszere az, hogy minden termékhez egy külön gyártófüggvényt határozunk meg (lásd a Gyártófüggvény mintát). A konkrét gyárak termékeiket a gyártófüggvény felülírásával állítják elő. Miközben ez a megvalósítás egyszerű, új konkrét gyár alosztály kell minden termékcsaládhoz, akkor is, ha a termékcsaládok csak kismértékben térnek el egymástól.
   Ha sok termékcsaládra lehet szükség, a konkrét gyár megvalósítható a Prototípus minta segítségével is. A konkrét gyár a családba tartozó minden termékhez külön prototípus-példányt használva készíthető elő, és az új termékek a prototípus klónozásával jönnek létre. A Prototípus alapú megközelítést használva nincs szükség minden új termékcsaládhoz új konkrét gyár osztályra.
   Az alábbi példa egy Prototípus alapú gyár Smalltalk nyelven történő megvalósítását szemlélteti. A konkrét gyár egy `partCatalog` nevű katalógusban tárolja a klónozandó prototípusokat. A `make:` metódus beolvassa és klónozza a prototípust:

```
make: partName
    ^ (partCatalog at: partName) copy
```

   Új alkatrészeket a konkrét gyár metódusával vehetünk fel a katalógusba.

```
addPart: partTemplate named: partName
    partCatalog at: partName put: partTemplate
```

   A prototípusokat egy szimbólummal azonosítva lehet a gyárhoz adni:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

   A Prototípus alapú megközelítés olyan nyelveken lehetséges, amelyek az osztályokat első osztályú objektumokként kezelik (ilyen például a Smalltalk és az Objective C). Ezeken a nyelveken az osztályt úgy kell elképzelni, mint egy olyan csökevényes gyárat, ami csak egyfajta terméket állít elő. Az *osztályokat* olyan konkrét gyárak belsejében lehet tárolni, amelyek a különböző konkrét termékeket változókban állítják elő, nagyjából úgy, mint a prototípusok. Az osztályok egy konkrét gyár nevében hoznak létre új példányokat. Új gyárat egy konkrét gyár egy példányának *termékosztályok* útján történő előkészítésével lehet meghatározni, nem pedig alosztályok létrehozásával. Ez a megközelítés kihasználja az egyes nyelvek szolgáltatásainak előnyeit, míg a tisztán Prototípus alapú megközelítés nyelvfüggetlen.
   Mint a fentebb említett Prototípus alapú gyárban a Smalltalk nyelvben, az osztály alapú változatban is egyetlen példányváltozó van (`partCatalog`), amely egy olyan katalógus vagy „szótár", amelynek kulcsa az alkatrész neve. A `partCatalog` nem klónozandó prototípusokat tárol, hanem termékosztályokat. A `make:` metódus ebben az esetben így néz ki:

```
make: partName
    ^ (partCatalog at: partName) new
```

3. *Bővíthető gyárak meghatározása.* Az ElvontGyár általában minden általa előállítható terméktípushoz más-más műveletet határoz meg. A terméktípusokat a művelet aláírása azonosítja. Új termékfajta hozzáadásához módosítani kell az ElvontGyár felületet és az összes vele függőségi viszonyban lévő osztályt.
   Rugalmasabb, bár kevésbé biztonságos megoldás, ha az objektumokat létrehozó műveleteket látjuk el paraméterekkel. Ez a paraméter adja meg a létrehozandó objektum típusát. Ez lehet osztályazonosító, egész szám, karakterlánc vagy bármi más, ami alkalmas a termékfajta azonosítására. Ezt a megközelítést használva az ElvontGyár osztálynak csak egyetlen „Make" műveletre van szüksége, amely el van látva egy olyan paraméterrel, amely jelzi a létrehozandó objektum típusát. Ezt az eljárást használjuk a fentebb említett Prototípus és osztály alapú elvont gyárakban.
   Ezt a változatot a dinamikus jellegű nyelvekben – amilyen például a Smalltalk – könnyebb használni, mint a statikusokban, amilyen például a C++. A C++ nyelvben a megoldás csak akkor használható, ha minden objektum ugyanazt az elvont alaposztályt használja, vagy ha a termékobjektumok az őket igénylő ügyfél útján biztonságosan a megfelelő típusra korlátozhatók. A Gyártófüggvény minta megvalósítását tárgyaló részben látni fogjuk, hogyan lehet ilyen paraméteres műveleteket megvalósítani a C++ nyelvben.
   De akkor is marad még egy gond, ha nem kell semmit sem korlátoznunk: az ügyfélnek minden terméket *ugyanazzal* az elvont felülettel ad vissza a program, amelyet a visszatérési típus meghatároz, így az ügyfél nem lesz képes megkülönböztetni vagy biztonsággal felismerni a különféle termékosztályokat. Amennyiben az ügyfélnek alosztályra jellemző műveletet kell végrehajtania, az elvont felületen keresztüli elérés lehetetlenné válik. Bár az ügyfél lefelé irányuló típusátalakítást (ez a C++ nyelvben a `dynamic_cast`) tud végezni, ez nem mindig ésszerű és nem is mindig biztonságos, mivel az átalakítás meghiúsulhat. Rendszerint ez az ára a nagy rugalmasságú és bővíthető felületeknek.

Példakód

Az Elvont gyár minta segítségével most létrehozzuk azt a labirintust, amelyről a fejezet elején volt szó.

A `MazeFactory` (LabirintusGyár) osztály a labirintus elemeit tudja létrehozni. Szobákat, falakat és szobák közti ajtókat épít. Használhatja olyan program, amely képes fájlból kiolvasni a labirintus „alaprajzát", és felépíteni abból a kívánt labirintust, vagy olyan, amely véletlenszerűen építi fel a labirintust. A labirintuskészítő programok argumentumként veszik fel a `MazeFactory` osztályt, hogy a programozók megadhassák a kialakítandó szobák, falak és ajtók osztályait.

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Jusson eszünkbe, hogy a korábban bemutatott `CreateMaze` (LétrehozLabirintus) tagfüggvény egy kis, két szobából és a köztük lévő ajtóból álló labirintust épít fel. A `CreateMaze` mereven bekódolja az osztályneveket, megnehezítve a különböző elemekből kialakított labirintusok létrehozását.

Íme a `CreateMaze` egy olyan változata, amely kijavítja ezt a hiányosságot, mégpedig úgy, hogy egy `MazeFactory` osztályt kap paraméterként:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

Az elvarázsolt labirintust építő `EnchantedMazeFactory` (ElvarázsoltLabirintusGyár) osztályt a `MazeFactory` alosztályaként hozhatjuk létre. Az `EnchantedMazeFactory` különböző tagfüggvényeket bírál felül, és a `Room`, `Wall` stb. osztályok különböző alosztályait adja vissza.

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n)  const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2)  const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

Most tételezzük fel azt, hogy olyan labirintusjátékot szeretnénk készíteni, amelyben az egyik szobában egy bomba van. Ha a bomba felrobban, az lerombolja (legalább) a falakat. Készíthetünk olyan `Room` (Szoba) alosztályt, amely nyilvántartja, hogy van-e bomba az adott szobában, és ha igen, felrobbant-e már. Szükségünk lesz olyan `Wall` (Fal) alosztályra is, amely azt tartja nyilván, hogy megsérült-e a fal, és ha igen, hogyan. Legyen ezeknek az alosztályoknak a neve `RoomWithABomb` (SzobaBombával), illetve `BombedWall` (LeromboltFal).

Az utolsó osztály, amelyet meghatározunk, a `BombedMazeFactory` (LeromboltLabirintusGyár). Ez a `MazeFactory` osztály alosztálya, és a `BombedWall` osztály lerombolt falait és a `RoomWithABomb` osztály bombát tartalmazó szobáit alakítja ki. A `BombedMazeFactory` alosztálynak csak két függvényt kell felülírnia:

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

Ha olyan egyszerű labirintust szeretnénk készíteni, amelyben bomba van, egyszerűen csak a `BombedMazeFactory` alosztállyal hívjuk meg a `CreateMaze` műveletet.

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

A `CreateMaze` megkaphatja az `EnchantedMazeFactory` egy példányát is, így tudunk például elvarázsolt labirintust létrehozni.

Jegyezzük meg, hogy a `MazeFactory` nem más, mint gyártófüggvények puszta gyűjteménye – ez az Elvont gyár minta legáltalánosabb megvalósítási módja –, valamint azt is, hogy a `MazeFactory` nem elvont osztály, így az ElvontGyár *és* a KonkrétGyár szerepét is betölti, ami egy másik gyakori megvalósítása az Elvont gyár mintának egyszerű alkalmazásokban. Mivel a `MazeFactory` olyan konkrét osztály, amely tisztán gyártófüggvényekből áll, új `MazeFactory` osztályt könnyű létrehozni úgy, hogy készítünk egy alosztályt, és felülírjuk a módosítandó műveleteket.

A `CreateMaze` a `SetSide` (BeállítOldal) művelettel hozta létre a szobák oldalait. Ha a `BombedMazeFactory` alosztállyal hozza létre a szobákat, akkor a labirintus `BombedWall` oldalú `RoomWithABomb` objektumokból áll össze. Ha a `RoomWithABomb` alosztálynak a `BombedWall` alosztály egy tagját kell elérnie, akkor a falaira egy hivatkozást kell átalakítania a `Wall*` alosztályból a `BombedWall*` alosztályba. Ez a lefelé irányuló átalakítás egészen addig biztonságos, amíg az argumentum tényleg egy `BombedWall` alosztály, ami akkor igaz, ha a falakat kizárólag `BombedMazeFactory` alosztályokkal építjük fel.

A dinamikus típusokkal dolgozó nyelvekben, amilyen például a Smalltalk, természetesen nincs szükség lefelé irányuló átalakításra, de futásidejű hibákat okozhat, ha `Wall` osztályba ütközünk ott, ahol a `Wall` osztály egy *alosztályára* számítunk. Ha a falakat az Elvont gyár minta segítségével építjük fel, az segít megelőzni az ilyen futásidejű hibákat, mivel biztosítja, hogy csak bizonyos típusú falakat lehessen létrehozni.

Vizsgáljuk meg a `MazeFactory` egy Smalltalk nyelvű változatát, egy olyat, amelyben egyetlen `make` művelet van, amely az objektum típusát kapja paraméterül. Sőt mi több, a konkrét gyár tárolja a saját maga által létrehozott termékosztályokat.

Először írjuk meg a `CreateMaze` megfelelőjét Smalltalk nyelven:

```
createMaze: aFactory
    | room1 room2 aDoor |
    room1 := (aFactory make: #room) number: 1.
    room2 := (aFactory make: #room) number: 2.
    aDoor := (aFactory make: #door) from: room1 to: room2.
    room1 atSide: #north put: (aFactory make: #wall).
    room1 atSide: #east put: aDoor.
    room1 atSide: #south put: (aFactory make: #wall).
    room1 atSide: #west put: (aFactory make: #wall).
    room2 atSide: #north put: (aFactory make: #wall).
    room2 atSide: #east put: (aFactory make: #wall).
    room2 atSide: #south put: (aFactory make: #wall).
    room2 atSide: #west put: aDoor.
    ^ Maze new addRoom: room1; addRoom: room2; yourself
```

Amint a Megvalósítás részben már volt róla szó, a MazeFactory osztálynak csak a partCatalog változó egy példányára van szüksége, hogy egy olyan szótárat biztosítson, amelynek a kulcsa az összetevő osztálya. Jusson eszünkbe az is, hogyan valósítottuk meg a make: metódust:

```
make: partName
    ^ (partCatalog at: partName) new
```

Most létrehozhatunk egy MazeFactory osztályt, és használhatjuk a createMaze megvalósítására. A gyárat a MazeGame osztály createMazeFactory metódusának segítségével hozzuk létre.

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

A BombedMazeFactory vagy az EnchantedMazeFactory alosztályt a kulcsokhoz különböző osztályokat társítva hozhatjuk létre. Egy EnchantedMazeFactory alosztály például így hozható létre:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

Ismert felhasználások

Az InterViews a „Kit" utótagot használja [Lin92] az ElvontGyár osztályok megjelölésére; a WidgetKit és DialogKit elvont gyárak például a megjelenítési szabványhoz kapcsolódó felhasználói felületi objektumok (vezérlők, illetve párbeszédablakok) létrehozására valók. Az InterViews tartalmaz egy olyan LayoutKit osztályt is, amely különféle összetett objektumokat állít elő, attól függően, hogy milyen elrendezésre van szükség. Például egy alapvetően vízszintes elrendezésben másfajta összetett objektumokra lehet szükség a különféle dokumentum-tájolásokhoz (álló vagy fekvő).

Az ET++ [WGM88] az Elvont gyár mintát használja a különböző ablakkezelő rendszerek (például X Window és SunView) közötti hordozhatóság elérésére. A WindowSystem elvont alaposztály az ablakrendszer erőforrásait (például MakeWindow, MakeFont, MakeColor) jelképező objektumok létrehozására szolgáló felületet határozza meg. A konkrét alosztály-

ok egy-egy ablakrendszer felületét valósítják meg. Futásidőben az ET++ abból a konkrét WindowSystem alosztályból hoz létre egy példányt, amely a konkrét rendszererőforrás-objektumokat állítja elő.

Kapcsolódó minták

Az ElvontGyár osztályokat gyakran gyártófüggvényekkel valósítják meg (Gyártófüggvény minta), de megvalósíthatók a Prototípus minta segítségével is.

A konkrét gyár gyakran Egyke.

Építő

Objektum-létrehozási minta

Egyéb nevek

Builder

Cél

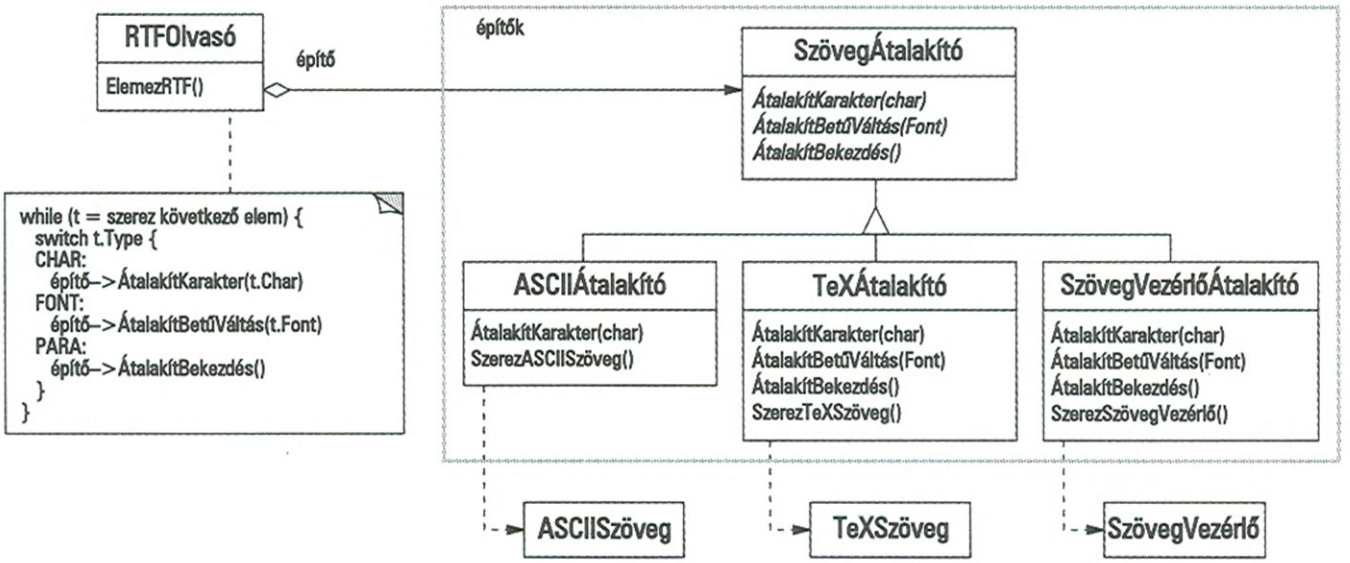
Az összetett objektumok felépítésének függetlenítése az ábrázolásuktól, így ugyanazzal az építési folyamattal különböző ábrázolásokat hozhatunk létre.

Feladat

Az RTF (Rich Text Format) dokumentumcsere-formátumot olvasó alkalmazásnak képesnek kell lennie az RTF sok egyéb szövegformátumba való átalakítására is. Az olvasóprogram átalakíthatja az RTF dokumentumot sima ASCII szöveggé vagy interaktív módon szerkeszthető szövegkezelő felületelemmé. A gond azonban az, hogy a lehetséges átalakítások száma korlátlan, így az olvasóprogram módosítása nélkül is egyszerűen kell, hogy új átalakítást adhassunk meg.

Egy megoldás lehet, ha az RTFOlvasó (RTFReader) osztályt az RTF formátumot másfajta szövegmegjelenítési formátumra átalakító SzövegÁtalakító (TextConverter) objektummal állítjuk be. Amikor az RTFOlvasó elemzi az RTF dokumentumot, a SzövegÁtalakító objektumot használja az átalakítás végrehajtására. Amikor az RTFOlvasó felismer egy RTF elemet („tokent") (sima szöveget vagy RTF-vezérlőszót), kiad egy kérést a SzövegÁtalakító objektumnak, hogy alakítsa át azt. A SzövegÁtalakító objektumok felelnek az adatátalakítás végrehajtásáért és az elem kért formátumban való megjelenítéséért is.

A SzövegÁtalakító alosztályai különböző átalakításokra és formátumokra „szakosodhatnak". Az ASCIIÁtalakító (ASCIIConverter) alosztály például figyelmen kívül hagyhat minden kérelmet, kivéve a sima szöveges átalakításra vonatkozóakat. A TeXÁtalakító (TeXConverter) alosztályok azokat a műveleteket valósíthatják meg, amely a TeX formátumban való, a szöveg minden stílusinformációját tartalmazó szövegmegjelenítésre vonatkozó kérelmekkel kapcsolatosak. A SzövegVezérlőÁtalakító (TextWidgetConverter) alosztályok olyan összetett felhasználói felületi objektumokat hozhatnak létre, amelyek lehetővé teszik, hogy a felhasználó láthassa és szerkeszthesse a szöveget.



Mindegyik átalakítóosztály-típus tartalmazza, ami az összetett objektumok létrehozásához és összeállításához szükséges, és egy elvont felület mögé rejti azt. Az átalakító elkülönül az olvasóprogramtól, amely az RTF dokumentum elemzéséért felelős.

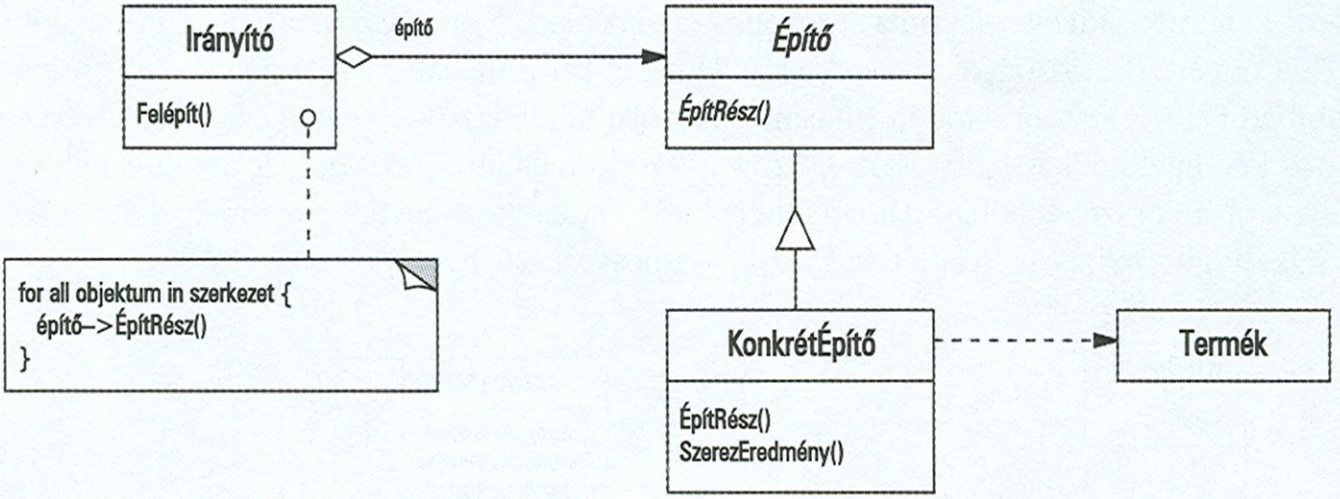
Az Építő minta mindeme kapcsolatokat tartalmazza. A mintában az átalakító osztályok neve **építő** (builder), míg az olvasóprogramé **irányító** (director). A fenti példára alkalmazva: az Építő minta elkülöníti a szöveges formátumot értelmező algoritmust (azaz az RTF formátumot elemző részt) az átalakított formátum létrehozásának és megjelenítésének módjától. Ez lehetővé teszi, hogy az RTFOlvasó elemző algoritmusát újra felhasználjuk RTF dokumentumok alapján készített másfajta szövegmegjelenítési formátumok létrehozására, és ehhez nem kell mást tenni, csak másik SzövegÁtalakító alosztályt megadni az RTFOlvasó osztálynak.

Alkalmazhatóság

Az Építő mintát a következő esetekben használjuk:

- Az összetett objektumok létrehozási algoritmusának függetlennek kell lennie az objektumot alkotó részegységektől és azok összeállítási módjától.
- Az építés folyamatának lehetővé kell tennie, hogy a létrehozott objektum többféleképpen jelenhessen meg.

Szerkezet



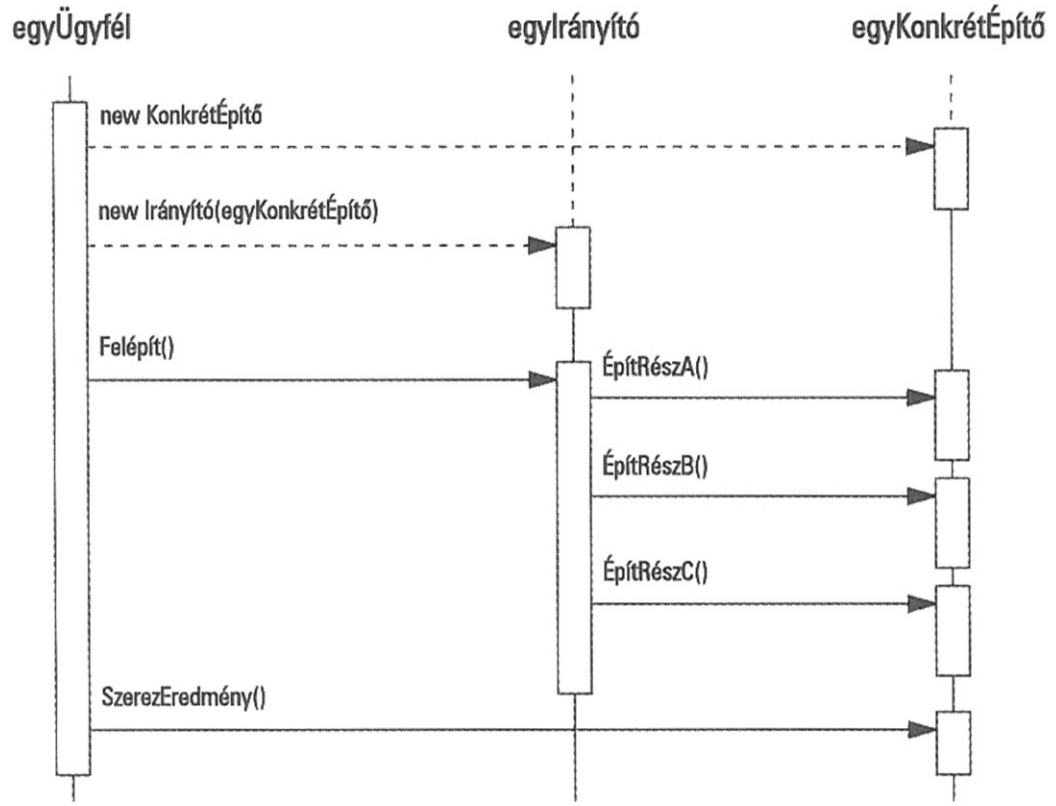
Résztvevők

- **Építő** (SzövegÁtalakító)
  - Megadja a Termék objektumok alkotórészeinek létrehozására szolgáló elvont felületet.
- **KonkrétÉpítő** (ASCIIÁtalakító, TeXÁtalakító, SzövegVezérlőÁtalakító)
  - Az Építő felületét megvalósítva megalkotja és összeállítja a termék részeit.
  - Meghatározza és nyilvántartja az általa létrehozott megjelenítési módokat.
  - A termék beolvasására szolgáló felületet (például SzerezASCIISzöveg, SzerezSzövegVezérlő) biztosít.
- **Irányító** (RTFOlvasó)
  - Az Építő felület segítségével megalkot egy objektumot.
- **Termék** (ASCIISzöveg, TeXSzöveg, SzövegVezérlő)
  - A felépítendő összetett objektumot jelképezi. A KonkrétÉpítő felépíti a termék belső ábrázolását, és meghatározza a termék összeállítási folyamatát.
  - Tartalmazza az elemeket meghatározó osztályokat, köztük az elemek összeállítására szolgáló felületeket is.

Együttműködés

- Az ügyfél létrehozza az Irányító objektumot, és beállítja a megfelelő Építő objektummal.
- Az Irányító értesíti az építőt, ha a termékhez hozzá kell adni valamilyen alkotórészt.
- Az Építő kezeli az irányítótól érkező kérelmeket, és alkotórészeket ad a termékhez.
- Az ügyfél elkéri a terméket az építőtől.

A következő együttműködési diagram azt szemlélteti, hogyan működik együtt az Építő és az Irányító az ügyféllel.



Következmények

Az Építő minta legfőbb előnyei a következők:

1. *Lehetővé teszi a termék belső ábrázolásának megváltoztatását.* Az Építő objektum egy elvont felületet biztosít az irányító számára a termék összeállításához. A felület lehetővé teszi, hogy az építő elrejtse a termék belső szerkezetét, valamint a termék összeállításának módját is. Mivel a termék összeállítása egy elvont felületen át történik, a termék belső ábrázolásának megváltoztatásához elegendő egy új építőt meghatározni.
2. *Elszigeteli az összeállítási és a megjelenítési kódot.* Az Építő minta úgy javítja a modulrendszerű kialakítást, hogy egységbe zárja az összetett objektum összeállításának és megjelenítésének módját. Az ügyfélnek nem kell tudnia semmit a termék belső szerkezetét meghatározó osztályokról, így ilyen osztályok nem jelennek meg az Építő felületében.
   Mindegyik KonkrétÉpítő tartalmazza az összes olyan kódot, amely egy adott termékfajta létrehozásához és összeállításához szükséges. A kódot csak egyszer kell megírni, aztán a különböző Irányítók újra felhasználják azt a Termék különféle változatainak ugyanabból az elemhalmazból való felépítéséhez. A korábban említett RTF-es példában például meghatározhatunk egy olvasóprogramot egy RTF-től eltérő formátumhoz, mondjuk egy SGMLOlvasó (SGMLReader) alosztályt, majd ugyanazokat a SzövegÁtalakító alosztályokat felhasználva előállíthatjuk az SGML dokumentumok ASCIISzöveg, TeXSzöveg és SzövegVezérlő alosztályok szerint formázott változatát is.

3. *A létrehozási folyamat finomabb vezérlését teszi lehetővé.* Azoktól a létrehozási mintáktól eltérően, amelyek egy lépésben alakítják ki a termékeket, az Építő minta az irányító felügyeletével lépésről lépésre alkotja meg azokat. Az irányító csak akkor veszi át a terméket az építőtől, amikor az már készen van. Emiatt az Építő felület jobban tükrözi a termék megalkotásának folyamatát, mint a többi létrehozási minta, és így finomabban lehet vezérelni az építési folyamatot és ennek következtében a végeredményként kapott termék belső szerkezetét.

Megvalósítás

Jellemzően van egy elvont Építő osztály, amely meghatározza minden olyan összetevő működését, amelynek létrehozására az irányító megkéri. A műveletek alapértelmezés szerint nem tesznek semmit. A KonkrétÉpítő osztályok azon elemek esetében felülbírálják a műveleteket, amelyeknek a létrehozásában érdekeltek.

A megvalósítással kapcsolatban az alábbiakat kell figyelembe venni:

1. *Összeállítási és építési felület.* Az Építők a termékeket lépésről lépésre alkotják meg, ezért az Építő osztály felületének eléggé általánosnak kell lennie ahhoz, hogy mindenféle konkrét építő számára lehetővé tegye a termékek létrehozását.
   A tervezés kulcsproblémája az építési és összeállítási modell. Azok a modellek általában kielégítőek, ahol a létrehozási kérelmek eredményét egyszerűen hozzácsatolják a termékhez. A RTF-es példában az építő átalakítja a következő elemet, és hozzáfűzi az addig már átalakított szöveghez.
   Néha viszont hozzá kell férni a korábban felépített termék elemeihez. A Példakód részben lévő labirintusos példában a MazeBuilder (LabirintusÉpítő) felület lehetőséget ad arra, hogy egy ajtót vegyünk fel két, már meglévő szoba közé. Az alulról felfelé építkező faszerkezetek – például az elemzőfák – egy másik példáját adják a probléma megoldására. Ebben az esetben az építő gyermekcsomópontokat ad vissza az irányítónak, amely aztán átadja azokat a szülőcsomópontokat kialakító építőnek.
2. *Miért nem elvont osztályokat használunk a termékekhez?* Általános esetben a konkrét építők által előállított termékek oly mértékben eltérnek megjelenésükben, hogy csak keveset nyerhetünk azzal, ha a különféle termékekhez ugyanazt az általános szülőosztályt rendeljük. Az RTF-es példában nem túl valószínű, hogy az ASCIISzöveg és a SzövegVezérlő objektum felülete közös lesz, és az sem, hogy szükségük legyen rá. Mivel az ügyfél általában a megfelelő konkrét építővel állítja be az irányítót, az ügyfél tudja, hogy az Építő mely konkrét alosztálya van használatban, és ennek megfelelően tudja kezelni a termékeket.
3. *Üres függvények alapértelmezettként az Építőben.* A C++ nyelvben az építő függvényeket szándékosan nem tisztán virtuális tagfüggvényekként vezetjük be. Ehelyett üres függvényekként határozzuk meg őket, lehetővé téve, hogy az ügyfelek csak azokat a műveleteket bírálják felül, amelyekben érdekeltek.

Példakód

A `CreateMaze` tagfüggvény egy olyan változatát fogjuk létrehozni, amely a `MazeBuilder` osztály egyik építőjét kapja argumentumként.

A `MazeBuilder` osztály a következő felületet határozza meg a labirintus létrehozásához:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Ez a felület három dolgot képes létrehozni: (1) a labirintust, (2) szobákat a megfelelő szobaszámmal és (3) ajtókat a számozott szobák között. A `GetMaze` (SzerezLabirintus) művelet a labirintust adja vissza az ügyfélnek. A `MazeBuilder` alosztályai felülbírálják ezt a műveletet, hogy a saját maguk által felépített labirintust adják vissza.

A `MazeBuilder` összes labirintusépítő művelete alapértelmezés szerint nem csinál semmit. Nem tisztán virtuálisként vezetjük be őket, hogy a származtatott osztályok csak azokat a függvényeket bírálhassák felül, amelyekben érdekeltek.

Ha adott a `MazeBuilder` felület, a `CreateMaze` tagfüggvényt úgy módosíthatjuk, hogy ezt az építőt vegye át paraméterként.

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Hasonlítsuk össze a `CreateMaze` ezen változatát az eredetivel. Figyeljük meg, hogy az építő hogyan rejti el a labirintus belső ábrázolását – azaz az ajtókat, szobákat és falakat meghatározó osztályokat –, és hogyan állítja össze ezekből az elemekből a végleges labirintust. Biztos van, aki már rájött, hogy vannak olyan osztályok, amelyek a szobák és ajtók megjelenítésére szolgálnak, de nyoma sincs a falak megjelenítésére valóknak. Ez megkönnyíti a labirintus megjelenítési módjának megváltoztatását, mivel a `MazeBuilder` egyik ügyfelét sem kell megváltoztatni.

A többi létrehozási mintához hasonlóan az Építő minta is egységbe zárja az objektumok létrehozásának módját, amely ebben az esetben a `MazeBuilder` által meghatározott felületen át történik. Ez annyit tesz, hogy a `MazeBuilder` felhasználható többféle labirintus felépítésére. Nézzük meg ezt most a `CreateComplexMaze` (LétrehozÖsszetettLabirintus) művelettel:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Figyeljük meg, hogy a `MazeBuilder` nem magát a labirintust hozza létre, a fő célja csak annyi, hogy egy felületet határozzon meg a labirintusok létrehozásához. Üres megvalósításokat határoz meg, főleg kényelmi szempontok miatt. A konkrét munkát a `MazeBuilder` alosztályai végzik.

A `StandardMazeBuilder` (SzabványLabirintusÉpítő) alosztály egy olyan megvalósítás, amely egyszerű labirintusokat készít. A saját maga által készített labirintus adatait a `_currentMaze` (_aktuálisLabirintus) változóban tárolja.

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

A `CommonWall` (KözösFal) egy olyan segédművelet, amely meghatározza a szobák közti közös falak irányát.

A `StandardMazeBuilder` konstruktor egyszerűen csak kezdőértéket ad a `_currentMaze` változónak.

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

A `BuildMaze` (ÉpítLabirintus) példányosítja az egyik `Maze` osztályt, amelyet a többi művelet állít össze, és végül visszaadja az ügyfélnek (a `GetMaze` művelettel).

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

A `BuildRoom` (ÉpítSzoba) művelet egy szobát hoz létre, és felépíti a határoló falait:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Ha ajtót szeretnénk két szoba közé, a `StandardMazeBuilder` megkeresi a két szobát a labirintusban, és a közös falukat:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

Az ügyfelek most a `CreateMaze` műveletet és a `StandardMazeBuilder` alosztályt együtt használva hozhatják létre a labirintust:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

Berakhattuk volna az összes `StandardMazeBuilder` műveletet a `Maze` osztályba, és hagyhattuk volna, hogy minden `Maze` osztály felépítse saját magát (azaz a labirintust). De ha kisebbre vesszük a `Maze` osztályt, könnyebb lesz megérteni és módosítani, és a `StandardMazeBuilder` alosztályt könnyű elkülöníteni a `Maze` osztálytól. Ennél is fontosabb, hogy a kettőt elkülönítve sokféle `MazeBuilder` építőt használhatunk, amelyek mindegyike különböző osztályokat használ a szobák, falak és ajtók létrehozására.

Egy egzotikusabb `MazeBuilder` építő a `CountingMazeBuilder` (SzámolóLabirintus-Építő). Ez az építő nem hoz létre semmilyen labirintust, csak összeszámolja a különféle létrehozható összetevőket.

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

A konstruktor kezdőértéket ad a számlálóknak, amelyek értékét a felülbírált `MazeBuilder` műveletek növelik.

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Egy ügyfél így alkalmazhatja például a `CountingMazeBuilder` alosztályt:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "A labirintusban "
     << rooms << " szoba és "
     << doors << " ajtó van." << endl;
```

Ismert felhasználások

Az RTF-átalakító alkalmazás az ET++ [WGM88] része. Szövegkialakító blokkja egy építőt használ az RTF formátumban tárolt szöveg feldolgozására.

Az Építő a Smalltalk-80 [Par90] esetében általános minta:

- A fordítói alrendszerben a Parser (Elemző) osztály egy olyan Irányító, amely egy ProgramNodeBuilder (ProgramCsomópontÉpítő) nevű objektumot kap argumentumként. A Parser objektumok értesítik saját ProgramNodeBuilder objektumukat, amikor felismernek egy nyelvtani szerkezetet. Miután az elemző végzett, elkéri az építőtől azt az elemzőfát, amelyet felépített, és visszaadja az ügyfélnek.
- A ClassBuilder (OsztályÉpítő) olyan építő, amelyet az osztályok (Class) használnak arra, hogy alosztályokat hozzanak létre saját részükre. Ebben az esetben a Class egyszerre Irányító és Termék.
- A ByteCodeStream (BájtKódFolyam) olyan építő, amely bájttömb formájában hoz létre egy lefordított metódust. A ByteCodeStream az Építő minta nem szabványos felhasználási módja, mert a létrehozott összetett objektum bájttömb formájában kódolt, nem normál Smalltalk objektumként. A ByteCodeStream felülete viszont olyan, mint az építőké általában, és a ByteCodeStream könnyen lecserélhető más olyan osztályokra, amelyek összetett objektumként jelenítik meg a programokat.

Az Adaptive Communications Environment szolgáltatásbeállító keretrendszere (Service Configurator) egy építő segítségével alkotja meg a hálózati szolgáltatási összetevőket, amelyek futásidőben kapcsolódnak egy kiszolgálóhoz [SS94]. Az összetevőket valamilyen beállítási nyelv írja le, amely egy LALR(1) elemzővel elemezhető. A jelentéselemző eljárások azon az építőn hajtanak végre műveleteket, amely a szolgáltatási összetevőhöz ad hozzá információkat. Ebben az esetben az elemző Irányító.

Kapcsolódó minták

Az Elvont gyár annyiban hasonlít az Építőhöz, hogy az is összetett objektumok megalkotására képes. A legfőbb eltérés közöttük, hogy az Építő minta az összetett objektumok lépésről lépésre történő létrehozását helyezi előtérbe, az Elvont gyár pedig a termékobjektumcsaládokra helyezi a hangsúlyt (legyenek azok egyszerűek vagy összetettek). Az Építő utolsó lépésként adja vissza a terméket, míg az Elvont gyár azonnal.

Az építő gyakran Összetételeket készít (lásd az Összetétel tervezési mintát).

Gyártófüggvény

Osztálylétrehozási minta

Cél

Felület meghatározása egy objektum létrehozásához, az alosztályokra bízva, melyik osztályt példányosítják. A gyártófüggvények megengedik az osztályoknak, hogy a példányosítást az alosztályokra ruházzák át.

Egyéb nevek

Factory Method, Gyár módszer, Gyártó metódus, Virtuális konstruktor

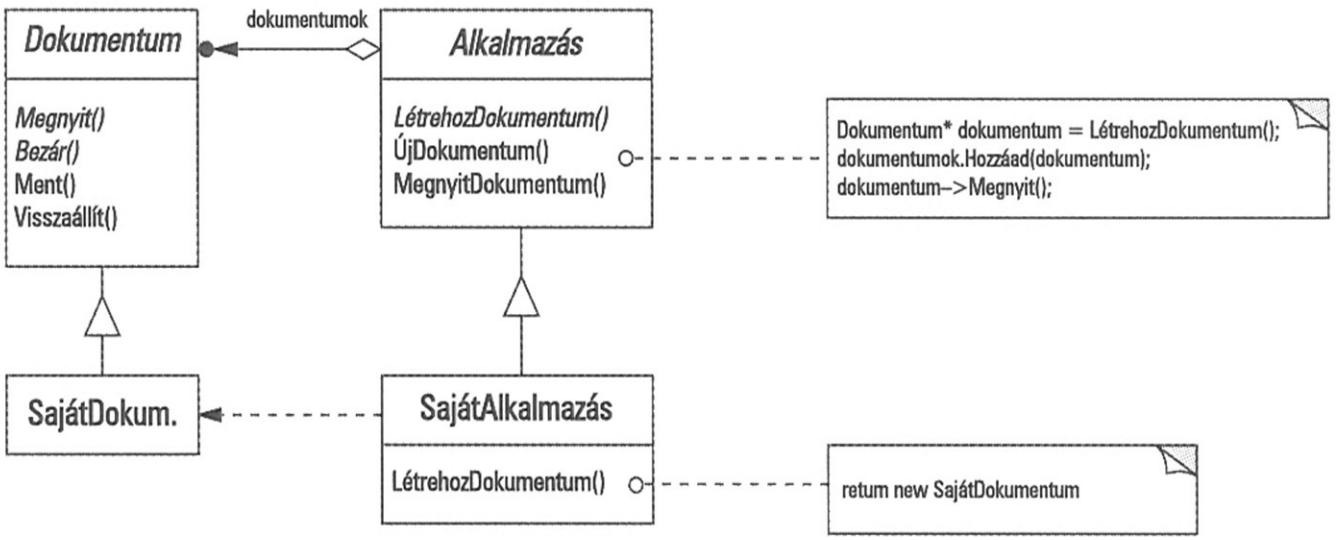
Feladat

A keretrendszerek elvont osztályokat használnak az objektumok közötti kapcsolatok meghatározására és fenntartására. A keretrendszer gyakran felelős ezeknek az objektumoknak a létrehozásáért is.

Képzeljünk el egy olyan alkalmazásokhoz való keretrendszert, amely több dokumentumot is meg tud jeleníteni a felhasználó számára. Ebben a keretrendszerben a két fő elem az Alkalmazás (Application) és a Dokumentum (Document) osztály. Mindkét osztály elvont, és az ügyfeleknek alosztályt kell belőlük készíteni ahhoz, hogy elő tudják állítani ezen osztályok alkalmazásfüggő megvalósításait. Egy rajzolóprogram létrehozásához például a RajzAlkalmazás (DrawingApplication) és a RajzDokumentum (DrawingDocument) osztályt határozzuk meg. Az Alkalmazás osztály felel a dokumentumok kezeléséért, és ez hozza őket létre, amikor arra szükség van, azaz amikor a felhasználó egy menüben például a Megnyitás vagy az Új elemet választja.

Mivel az, hogy melyik Dokumentum alosztályból kell példányt létrehozni, alkalmazásfüggő, az Alkalmazás osztály nem tudja előre megjósolni, melyik Dokumentum alosztályt kell példányosítani – csak azt tudja, hogy *mikor* kell új dokumentumot létrehozni, azt nem, hogy *milyen típusút*. Ez egy nagy problémát vet fel: a keretrendszernek példányokat kell előállítania az osztályokból, de csak az elvont osztályokat ismeri, amelyeket viszont nem tud példányosítani.

Erre a problémára a Gyártófüggvény minta ajánlja a megoldást. A gyártófüggvény zárja egységbe azt a tudást, hogy melyik Dokumentum alosztályt kell létrehozni, és kiemeli ezt a tudást a keretrendszerből.



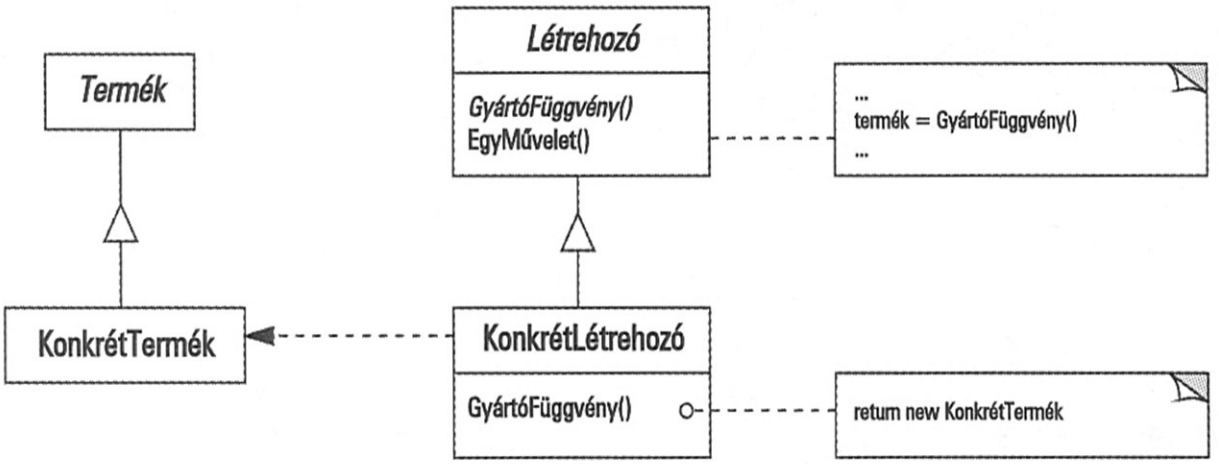
Az Alkalmazás alosztályok az Alkalmazás osztályon működő elvont LétrehozDokumentum (CreateDocument) művelet felülírásával adják vissza a megfelelő Dokumentum alosztályt. Az Alkalmazás alosztály példányosítása után már képes példányosítani az alkalmazásfüggő dokumentumokat, anélkül, hogy tudná, melyik osztályba tartoznak. A LétrehozDokumentum műveletet **gyártófüggvénynek** (gyártó metódusnak) is nevezik, mert ez felel az objektumok „legyártásáért".

Alkalmazhatóság

A Gyártófüggvény mintát az alábbi esetekben használjuk:

- Az osztály nem tudja előre megjósolni, milyen objektumosztályt kell létrehoznia.
- Az osztály azt szeretné elérni, hogy az alosztályai adják meg, milyen objektumot kell létrehozni.
- Az osztályok átruházzák a felelősséget a számos segítő alosztály egyikére, és mi szeretnénk tudni, melyik lett a képviselő.

Szerkezet



Résztvevők

- **Termék** (Dokumentum)
  - Meghatározza a gyártófüggvény által létrehozott objektumok felületét.
- **KonkrétTermék** (SajátDokumentum)
  - Megvalósítja a Termék felületet.
- **Létrehozó** (Alkalmazás)
  - Meghatározza azt a gyártófüggvényt, amely egy Termék (Product) típusú objektumot ad vissza. A Létrehozó (Creator) meghatározhatja a függvény valamilyen alapértelmezett megvalósítását is, amely egy alapértelmezett KonkrétTermék objektumot ad vissza.
  - Meghívhatja a Termék objektumot létrehozó gyártófüggvényt.
- **KonkrétLétrehozó** (SajátAlkalmazás)
  - Felülbírálja a gyártófüggvényt, hogy a KonkrétTermék alosztály egy példányát adja vissza.

Együttműködés

- A létrehozó a saját alosztályaira bízza a gyártófüggvény meghatározását, hogy az a megfelelő KonkrétTermék alosztály egy példányát adja vissza.

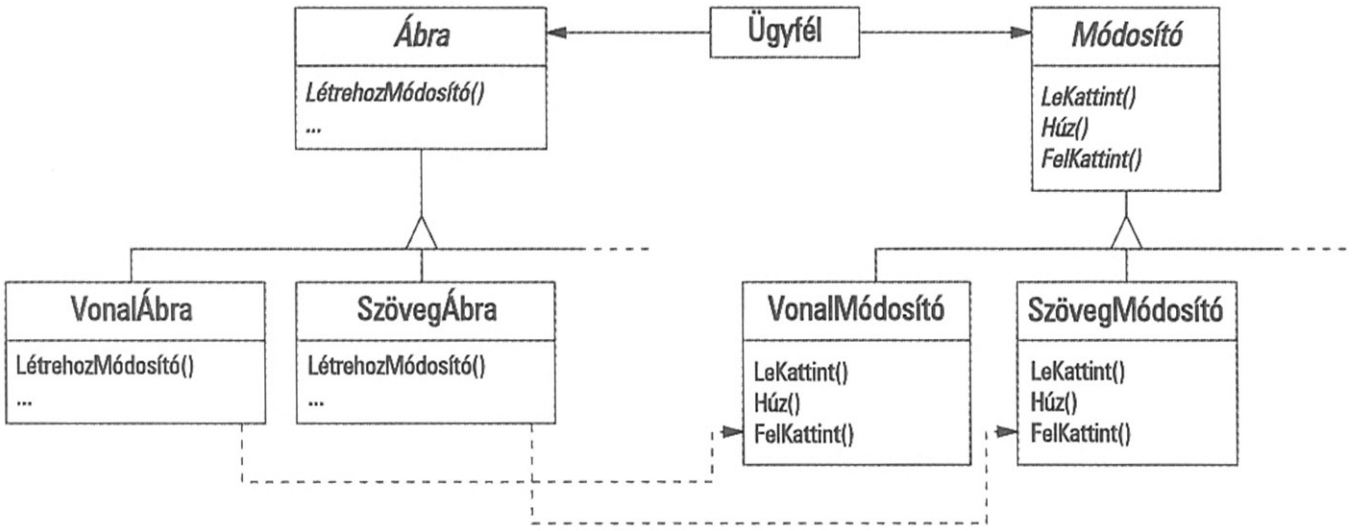
Következmények

A gyártófüggvények kiküszöbölik az alkalmazásfüggő osztályok kódhoz kötésének szükségességét. A kód csak a Termék felülettel foglalkozik, ezért képes együttműködni bármely felhasználó által meghatározott KonkrétTermék osztállyal.

A gyártófüggvényekben rejlő hátrány az, hogy az ügyfeleknek esetleg a Létrehozó osztályból is alosztályt kell létrehozniuk egy adott KonkrétTermék objektum elkészítéséhez. Az alosztály-származtatás abban az esetben megfelel, ha az ügyfélnek amúgy is alosztályt kellene létrehoznia a Létrehozó osztályból, de a többi esetben az ügyfélnek ilyenkor egy másik kibontakozási ponttal is foglalkoznia kell.

A Gyártófüggvény mintának többek között két előnye van:

1. *Horgokat biztosít az alosztályok számára.* Ha egy osztályon belül gyártófüggvénnyel hozunk létre objektumokat, az mindig rugalmasabb, mintha közvetlenül hoznánk létre azokat. A Gyártófüggvény minta egy olyan horoggal (hook) látja el az alosztályokat, amelynek segítségével létrehozható az objektum bővített változata.
   A dokumentumos példában a Dokumentum osztályban meghatározható egy LétrehozFájlPárbeszédablak (CreateFileDialog) nevű gyártófüggvény, amely egy alapértelmezett párbeszédablak-objektumot hoz létre a meglévő dokumentumok megnyitásához. A Dokumentum alosztály ezt a függvényt felülbírálva határozhat meg egy alkalmazásfüggő párbeszédablakot. Ebben az esetben a gyártófüggvény nem elvont, hanem egy ésszerű alapértelmezett megvalósítást tesz lehetővé.
2. *Összekapcsolja a párhuzamos osztályhierarchiákat.* Az eddig vizsgált példákban a gyártófüggvényt csak a Létrehozó osztályok hívták meg, de ennek nem kell okvetlenül így lennie. Az ügyfelek is hasznosnak találhatják a gyártófüggvényeket, főleg párhuzamos osztályhierarchiák esetében.
   Párhuzamos osztályhierarchiák akkor jönnek létre, amikor egy osztály átadja felelősségi körének egy részét egy másik, önálló osztálynak. Képzeljünk el egy olyan rajzot, ami interaktív módon módosítható, azaz nyújtható, áthelyezhető és elforgatható az egér segítségével. Ennek megvalósítása nem mindig könnyű, gyakran van hozzá szükség a módosítások egy adott időpontbeli állapotát tükröző adatok tárolására és frissítésére. Ezekre az állapotinformációkra csak a módosítások során van szükség, ezért nem kell azokat a rajzobjektumban tárolni. Ezenkívül a különböző rajzok módosításkor különbözőképpen viselkednek. Egy vonal nyújtása például okozhatja az egyik végpont áthelyezését, míg egy szöveges ábra nyújtásakor a sortávolság változhat.
   Ezek miatt a korlátozások miatt jobb külön Módosító (Manipulator) objektumot használni, amely megvalósítja az interaktivitást, és nyilvántartja a módosításhoz szükséges állapotinformációkat. A különféle ábrák különböző Módosító alosztályokat használhatnak az egyes műveletek kezelésére. Az így kapott Módosító osztályhierarchia (legalább részben) illeszkedik az Ábra (Figure) osztályhierarchiához:



Az Ábra osztály egy olyan LétrehozMódosító gyártófüggvényt biztosít, amely lehetővé teszi, hogy az ügyfelek létrehozzák a megfelelő Módosító alosztályt. Az Ábra alosztályok e függvényt felülbírálva adják vissza a Módosító alosztály egy olyan példányát, amely számukra megfelelő. A másik lehetőség, hogy az Ábra osztály valósítja meg a LétrehozMódosító műveletet, úgy, hogy az egy alapértelmezett Módosító példányt adjon vissza, és az Ábra alosztályok egyszerűen csak öröklik az alapértelmezést. Az ezt végrehajtó Ábra osztályoknak nincs szükségük semmilyen megfelelő Módosító alosztályra, emiatt a hierarchiák csak részlegesen „párhuzamosak".
Figyeljük meg, hogy a gyártófüggvény hogyan határozza meg a két osztályhierarchia közötti kapcsolatot, meghatározva azt, hogy mely osztályok tudása tartozik együvé.

Megvalósítás

A Gyártófüggvény minta megvalósításakor legyünk tekintettel a következőkre:

1. *Két főbb változat létezik.* A Gyártófüggvény mintának két főbb változata létezik: (1) az az eset, amikor a Létrehozó elvont osztály, és nem biztosítja az általa meghatározott gyártófüggvény semmilyen megvalósítását, és (2) az az eset, amikor a Létrehozó konkrét osztály, és biztosítja a gyártófüggvény egy alapértelmezett megvalósítását. Az is lehetséges, hogy egy olyan elvont osztályunk legyen, amely meghatároz valamilyen alapértelmezett megvalósítást, de ez nem túl gyakori.
   Az első esetben az alosztályok *kötelezőek* a megvalósítás meghatározásához, mert nincs semmilyen elfogadható alapértelmezés. Ez azt a problémát veti fel, hogy előre nem látható alosztályokat kell példányosítani. A második esetben a konkrét Létrehozó a gyártófüggvényt elsősorban a rugalmassága miatt veszi igénybe. A szabály a következő: „Az objektumokat külön művelettel hozzuk létre, hogy az alosztályok felül tudják bírálni a létrehozásuk módját." Ez a szabály biztosítja, hogy az alosztálytervezők szükség esetén módosíthassák azokat az objektumosztályokat, amelyeket a szülőosztályaik példányosítanak.

2. *Paraméterezett gyártófüggvények.* A minta egy másik változata lehetőséget ad arra, hogy a gyártófüggvény *többféle* terméket hozzon létre. A függvény ekkor egy olyan paramétert kap, amely azonosítja a létrehozandó objektumtípust. A függvény által készített minden objektum közösen a Termék felületet használja. A dokumentumos példában az alkalmazás többféle dokumentum használatát megengedheti; ehhez a LétrehozDokumentum műveletnek átadunk még egy paramétert, amely megadja, milyen típusú dokumentumot kell létrehozni.
   A Unidraw grafikus szerkesztő keretrendszer [VL90] ezt a megközelítést használja a lemezre mentett objektumok újraalkotására. Egy `Creator` nevű osztályt találunk benne, a `Create` gyártófüggvénnyel, amely egy osztályazonosítót kap argumentumként. Az osztályazonosító adja meg, melyik osztályt kell példányosítani. Amikor a Unidraw lemezre ment egy objektumot, először az osztályazonosítót írja ki, majd a példányváltozókat. Amikor újraalkotja az objektumot a lemezről, az osztályazonosítót olvassa be először.
   Az osztályazonosító beolvasása után a keretrendszer meghívja a `Create` gyártófüggvényt, és átadja neki paraméterként az azonosítót. A `Create` megkeresi a megfelelő osztályhoz tartozó konstruktort, és azt használja az objektumpéldány létrehozására. Végül a `Create` meghívja az objektum `Read` (Olvas) műveletét, amely beolvassa a lemezről az objektum többi adatát, és előkészíti az objektum példányváltozóit.
   A paraméterezett gyártófüggvény általános formája a következő: (A `MyProduct` és a `YourProduct` a `Product` osztály alosztályai.)

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE)  return new MyProduct;
    if (id == YOURS) return new YourProduct;
    //a többi termék esetében ismétlődik...

    return 0;
}
```

   A paraméterezett gyártófüggvény felülbírálása a Létrehozó (Creator) által előállított termékek egyszerű bővítését és módosítását teszi lehetővé. Az új termékfajtákhoz új azonosítókat lehet bevezetni, illetve más termékekhez társíthatjuk a már meglévő azonosítókat.
   A `MyCreator` (SajátLétrehozó) alosztály segítségével felcserélhető például a MyProduct (EnyémTermék) és a YourProduct (TiédTermék) alosztály, és támogatható egy új `TheirProduct` (ÖvékTermék) alosztály:

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
//Megjegyzés: a YOURS (tiéd) és a MINE (enyém) megcserélése
```

```
        if (id == THEIRS) return new TheirProduct;

        return Creator::Create(id);
        //ezt akkor hívja meg a program, ha a többi meghiúsul
    }
```

Figyeljük meg, hogy ez a művelet az utolsó lépésben meghívja a `Create`-et a szülőosztályra. Ennek oka, hogy a `MyCreator::Create` csak a `YOURS`, `MINE` és `THEIRS` osztályokat kezeli másképpen, mint a szülőosztály. Más osztályokban nincs érdekeltsége. Emiatt a `MyCreator` alosztály *kibővíti* a létrehozott termékfajtákat, és kevés kivétellel az összes termék létrehozásának felelősségét átruházza saját szülőjére.

3. *Nyelvfüggő változatok és problémák.* A különböző nyelvek további érdekes változatok használatát teszik lehetővé, és további hibalehetőségeket rejtenek.
   A Smalltalk programok gyakran használnak olyan függvényt (metódust), amely visszaadja a példányosítandó objektum osztályát. A létrehozó gyártófüggvény ezt az értéket egy termék létrehozásához használhatja, tárolását vagy akár a kiszámítását pedig egy KonkrétLétrehozó alosztály vállalhatja. Az eredmény: még későbbi kötés a példányosítandó KonkrétTermék alosztálytípushoz.
   A dokumentumos példa Smalltalk nyelven írt változatában meghatározható egy `documentClass` (dokumentumOsztály) metódus az `Application` osztályra. A `documentClass` a megfelelő `Document` osztályt adja vissza a dokumentumok példányosításához. A `documentClass` metódusnak a `MyApplication` alosztályban történő megvalósítása a `MyDocument` osztályt adja vissza. Így az `Application` osztályban a következők vannak:

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

   A `MyApplication` osztályban pedig ezek:

```
documentClass
    ^ MyDocument
```

   Ez az `Application` osztályhoz példányosítandó `MyDocument` osztályt adja vissza.
   Egy még rugalmasabb, a paraméterezett gyártófüggvényekhez hasonló megközelítés, ha a létrehozandó osztályt az `Application` osztály osztályváltozójaként tároljuk. Ily módon a termék különböző változatainak előállításához nem kell alosztályokat létrehozni az `Application` osztályhoz.
   A C++ nyelvben a gyártófüggvények mindig virtuális függvények, és gyakran tisztán virtuálisak. Csak arra vigyázzunk, hogy ne hívjunk meg gyártófüggvényt a Létrehozó konstruktorában – a KonkrétLétrehozó alosztályban ugyanis ekkor még nincs jelen.
   Ezt úgy kerülhetjük el, ha ügyelünk arra, hogy a termékeket kizárólag olyan elérési műveletekkel próbáljuk meg elérni, amelyek igény esetén hozzák létre a terméket. A konkrét termék konstruktorban való létrehozása helyett a konstruktor csak 0 kezdőértéket ad. Az elérési művelet visszaadja a terméket, de előbb ellenőrzi, hogy csak-

ugyan létezik-e, és ha nem, akkor létrehozza. Ezt a módszert néha **lusta előkészítésnek** (lazy initialization) is nevezik. A következő kód egy jellegzetes megvalósítást mutat:

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}
```

4. *Sablonok használata az alosztálykészítés elkerülése érdekében.* Amint már említettük, a gyártófüggvényekben rejlő másik lehetséges buktató az, hogy időnként csak azért kell alosztályokat előállítanunk, hogy létrehozhassuk a megfelelő termékobjektumokat. A C++ nyelvben ez úgy kerülhető meg, ha a Létrehozó (Creator) osztályból sablonalosztályt hozunk létre, amely a Termék (Product) osztályt kapja paraméterként:

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

Ezt a sablont használva az ügyfél csak a termékosztályt „szállítja", ezért nincs szükség arra, hogy alosztályokat hozzunk létre a létrehozó osztályhoz.

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

5. *Névadási szabályok.* Jó ötlet, ha olyan névadási szabályokat használunk, amelyek egyértelművé teszik, hogy gyártófüggvényekről van szó. Például a MacApp Macintosh alkalmazási keretrendszer [App89] azt az elvont műveletet, amely meghatározza a gyártófüggvényt, `Class* DoMakeClass()` formában vezeti be, ahol a `Class` a Termék osztály.

Példakód

A korábban már látott `CreateMaze` függvény egy labirintust hoz létre és ad vissza. Ezzel a függvénnyel az az egyik gond, hogy mereven kódolja a labirintus, a szobák, az ajtók és a falak osztályait. A gyártófüggvényeket azért vezetjük be, hogy az alosztályok választhassák ki ezeket az elemeket.

Először határozzuk meg a `MazeGame` labirintus, szoba, fal és ajtó objektumait létrehozó függvényeket:

```
class MazeGame {
public:
    Maze* CreateMaze();

// gyártófüggvények

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Mindegyik gyártófüggvény egy adott típusú labirintuselemet ad vissza. A `MazeGame` osztály olyan alapértelmezett megvalósításokat biztosít, amelyek a legegyszerűbb típusú labirintusokat, szobákat, falakat és ajtókat adják vissza.

Most újraírhatjuk a `CreateMaze` osztályt úgy, hogy ezeket a gyártófüggvényeket használja:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);
```

```
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

A különféle játékok alosztályokat hozhatnak létre a `MazeGame` osztályhoz a labirintus elemeinek megadása érdekében. Ezek az alosztályok aztán felülbírálhatják a gyártófüggvények egy részét vagy akár az összeset úgy, hogy azok a termék különféle változatait adják vissza. A `BombedMazeGame` alosztály például olyan új meghatározásokat adhat a `Room` és a `Wall` termékeknek, hogy azok a bombarobbanás utáni változatokat adják vissza:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

Az `EnchantedMazeGame` változat valahogy így határozható meg:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Ismert felhasználások

A gyártófüggvények meglehetősen elterjedtek az elemkészletekben és a keretrendszerekben. A korábban említett dokumentumos példa egy jellegzetes felhasználási mód a MacApp és az ET++ [WGM88] rendszerekben. A módosítós példa a Unidrawból származik.

A Smalltalk-80 Model/View/Controller (modell–nézet–vezérlő) keretrendszerének Class View (Osztálynézet) nézetében egy defaultController (alapértelmezettVezérlő) nevű osztályt találunk, amely egy vezérlőt hoz létre, és ez gyártófüggvénynek is tűnhet [Par90], de a View osztály alosztályai megadják saját alapértelmezett vezérlőjük osztályát azáltal, hogy meghatározzák a defaultControllerClass osztályt, amely azt az osztályt adja vissza, amelyből a defaultController példányokat készít. Így tehát valójában a defaultControllerClass az igazi gyártófüggvény, tehát az alosztályoknak azt kell felülbírálniuk.

A Smalltalk-80 nyelvben egy sokkal misztikusabb példa a Behavior (Viselkedés) osztály (az osztályokat jelképező összes objektumot tartalmazó főosztály) által meghatározott parserClass (elemzőOsztály) gyártófüggvény (gyártó metódus), amely lehetővé teszi, hogy az osztályok testreszabott elemzőprogramot használjanak a forráskódhoz. Egy ügyfél meghatározhat például egy SQLParser (SQLElemező) osztályt egy beágyazott SQL-utasításokat tartalmazó osztály forráskódjának elemzéséhez. A Behavior osztály a parserClass megvalósításával a szabványos Smalltalk Parser osztályt adja vissza. A beágyazott SQL-utasításokat tartalmazó osztály ezt a metódust bírálja felül (mint osztálymetódust), és az SQLParser osztályt adja vissza.

Az IONA Technologies [ION94] cég Orbix ORB rendszere a Gyártófüggvény minta segítségével hoz létre megfelelő típusú helyettest (lásd a Helyettes tervezési mintát), amikor egy objektum egy távoli objektumra mutató hivatkozást kér. A Gyártófüggvény minta megkönnyíti az alapértelmezett helyettes lecserélését például egy olyanra, amely felhasználó oldali gyorstárat használ.

Kapcsolódó minták

Az Elvont gyár mintát gyakran gyártófüggvényekkel valósítják meg. Az Elvont gyár minta Feladat részében említett példa szintén egy gyártófüggvényt szemléltet.

A gyártófüggvényeket általában sablonfüggvényekben hívják meg. A fentebb említett dokumentumos példában az ÚjDokumentum (NewDocument) is ilyen sablonfüggvény.

A prototípusok nem igénylik, hogy alosztályokat hozzunk létre a Létrehozó osztályból, de gyakran szükségük van egy, a Termék osztályon végrehajtott Előkészít (Initialize) műveletre. A Létrehozó osztály az Előkészít művelet segítségével készíti elő az objektumot. A gyártófüggvényeknek nincs szükségük ilyen műveletre.

Prototípus

Objektum-létrehozási minta

Cél

Prototípus példány használatával meghatározni, milyen típusú objektumokat kell létrehozni, az új objektumokat pedig ennek a prototípusnak a lemásolásával előállítani.

Feladat

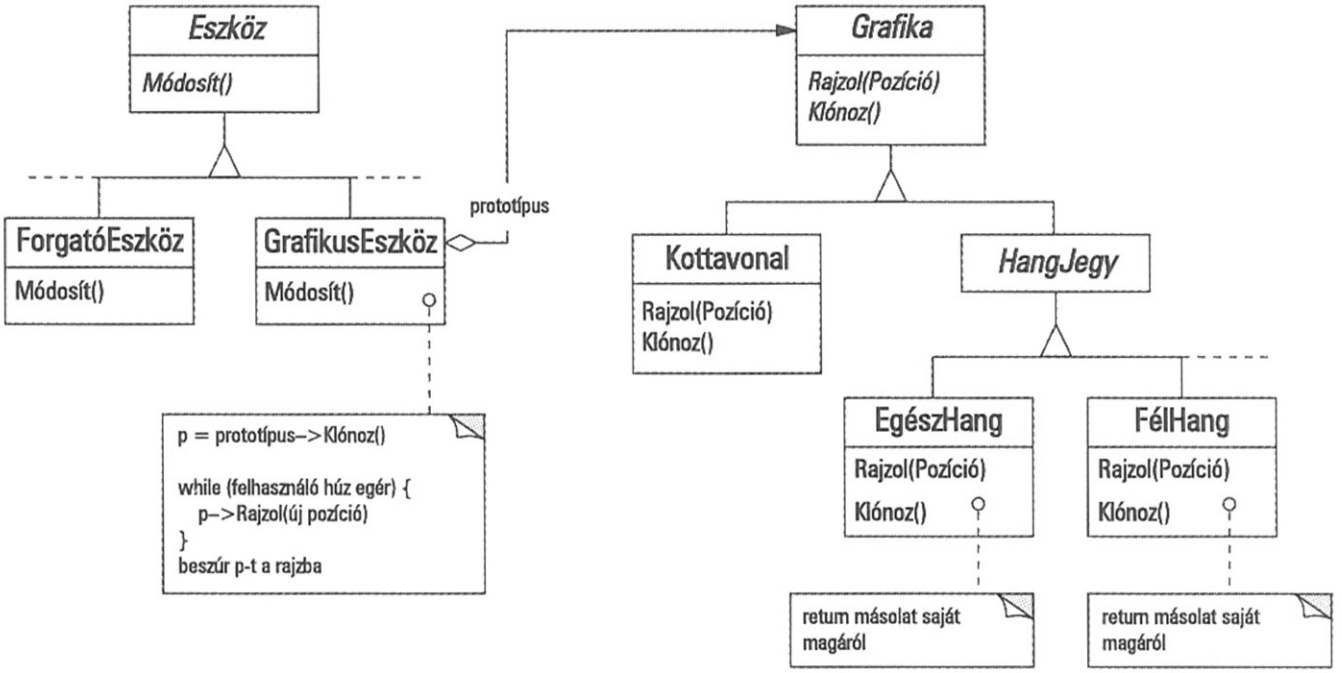
Kottaszerkesztő programot szeretnénk készíteni, átszabva egy képszerkesztőkhöz való általános keretrendszert, és hangjegyeket, szünetjeleket, illetve kottavonalakat jelképező új objektumokat adva ahhoz. A szerkesztő keretrendszerhez egy olyan eszközpaletta fog tartozni, amelynek segítségével ezeket a zenei objektumokat felvehetjük a kottába. A palettán lehetnek még a zenei objektumok kijelölésére, áthelyezésére és más módon történő módosítására szolgáló eszközök is. Ha a felhasználó negyed hangjegyeket akar felvenni a kottába, a „negyed hang eszközre" kattint, és azt használja. Ha a hangjegyet felfelé vagy lefelé szeretné mozgatni a kotta ötvonalas rendszerében, megváltoztatva a hangmagasságát, arra az „áthelyezés eszköz" használható.

Tételezzük fel, hogy a keretrendszer biztosít egy elvont Grafika (Graphics) osztályt a grafikus összetevőkhöz, amilyenek a hangjegyek és a kottavonalak, valamint egy elvont Eszköz (Tool) osztályt a palettán lévő eszközök meghatározásához. Ezenkívül készen biztosítja a GrafikusEszköz (GraphicTool) alosztályt a grafikus objektumok példányait előállító és azokat a dokumentumhoz adó eszközökhöz.

A GrafikusEszköz osztály azonban gondot jelent a keretrendszer-tervezőnek. A hangjegyek és kottavonalak osztályai csak erre az alkalmazásra jellemzőek, de a GrafikusEszköz osztály a keretrendszerhez tartozik. A GrafikusEszköz nem tudja, hogyan hozzon létre példányokat a zenei osztályokból, és hogyan adja azokat a kottához. Persze származtathatnánk alosztályokat a GrafikusEszköz osztályból a zenei objektumok különféle típusaihoz, de ekkor nagyon sok alosztály jönne létre, amelyek csak az általuk példányosított zenei objektumok típusában térnének el egymástól. Azt már tudjuk, hogy az objektum-összetétel az alosztálykészítés rugalmas alternatívája. A kérdés csak az, hogyan tudja ezt a keretrendszer a GrafikusEszköz osztály példányainak azon Grafika *osztállyal* való paraméterezésére használni, amelynek a létrehozására szolgálnak.

A megoldás: új Grafika osztály készíttetése a GrafikusEszköz osztállyal, lemásolva vagy „klónozva" egy Grafika alosztály egy példányát. Ezt a példányt hívjuk **prototípusnak**. A GrafikusEszköz osztály azt a prototípust kapja paraméterként, amelyet klónoznia kell, és a dokumentumhoz kell adnia. Ha minden Grafika alosztály támogatja a Klónoz (Clone) műveletet, akkor a GrafikusEszköz osztály a Grafika osztály bármely típusát képes klónozni.

Kottaszerkesztőnkben tehát minden zeneiobjektum-készítő eszköz a GrafikusEszköz osztály egy példánya, amelyet különböző prototípusokkal készítünk elő. Mindegyik GrafikusEszköz példány egy zenei objektumot hoz létre, mégpedig úgy, hogy klónozza saját prototípusát, majd felveszi a klónt a kottába.



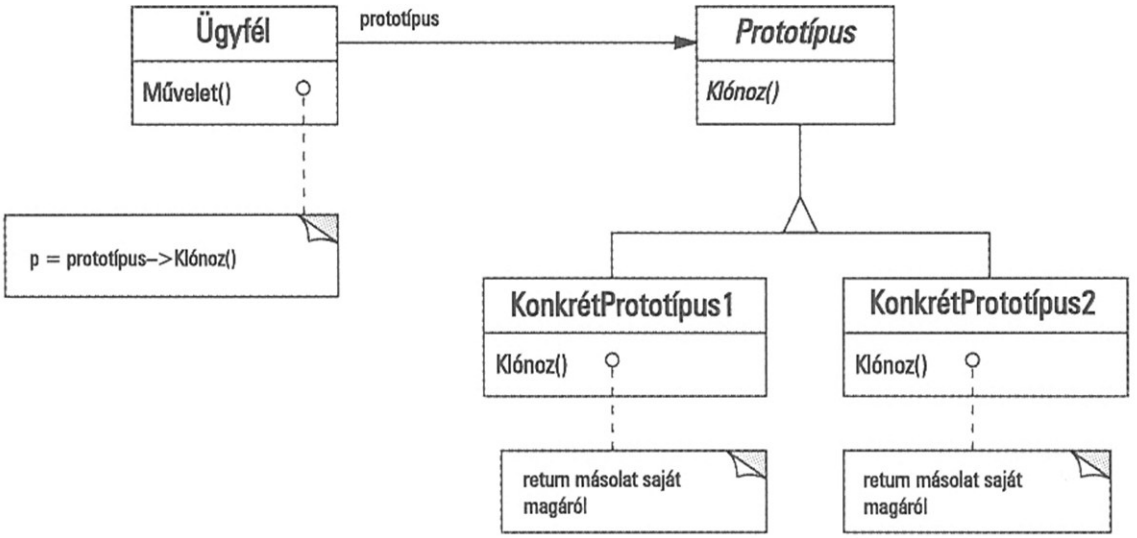
A Prototípus mintát az osztályok számának további csökkentésére használhatjuk. Külön osztályok vannak az egész és a fél hangjegyekhez, de erre valószínűleg nincs szükség, ehelyett lehetnek ezek ugyanannak az osztálynak a különféle bitképekkel és időtartamokkal előkészített példányai. Az egész hangjegyek létrehozására szolgáló eszköz így egy olyan GrafikusEszköz osztály lesz, amelynek prototípusa egy olyan HangJegy (MusicalNote) osztály, amely úgy van előkészítve, hogy egész hanggá váljon. Ez rendkívüli mértékben csökkentheti a rendszerben használt osztályok számát, és megkönnyíti új hangjegytípusok felvételét is a programba.

Alkalmazhatóság

A Prototípus mintát akkor használjuk, amikor a rendszernek függetlennek kell lennie a termékek létrehozásának, összeállításának és megjelenítésének módjától, *és*

- amikor a példányosítandó osztályokat futásidőben adják meg, például dinamikus betöltés útján, *vagy*
- ha el szeretnénk kerülni a termékekével párhuzamos osztályhierarchiájú gyárak építését, *vagy*
- amikor egy osztály példányainak csak néhány különböző állapotkombinációja jöhet létre. Kényelmesebb lehet a megfelelő számú prototípust beépíteni, majd klónozni, mint egyenként példányosítani az osztályokat, minden alkalommal a megfelelő állapottal.

Szerkezet



Résztvevők

- **Prototípus** (Grafika)
  - Felületet vezet be önmaga klónozásához.
- **KonkrétPrototípus** (Kottavonal, EgészHang, FélHang)
  - Megvalósít egy műveletet önmaga klónozása érdekében.
- **Ügyfél** (GrafikusEszköz)
  - Új objektumot hoz létre, megkérve egy prototípust, hogy klónozza önmagát.

Együttműködés

- Az ügyfél megkéri az egyik prototípust, hogy klónozza önmagát.

Következmények

A Prototípus minta alkalmazása hasonló következményekkel jár, mint az Elvont gyáré vagy az Építőé: elrejti a konkrét termékosztályt az ügyfél elől, csökkentve ezzel azon nevek számát, amelyeket az ügyfél ismer. Ezenkívül ezek a minták lehetővé teszik azt, hogy az ügyfél módosítás nélkül használja az alkalmazásra jellemző osztályokat.

A Prototípus minta további előnyei:

1. *Termékek hozzáadása és eltávolítása futásidőben.* A prototípusok lehetőséget adnak arra, hogy új konkrét termékosztályt építsünk be a rendszerbe, és ehhez nem kell mást tenni, csak bejegyeztetni egy prototípuspéldányt az ügyféllel. Ez némileg rugalmasabb, mint a többi létrehozási minta megoldása, mivel az ügyfél futásidőben építheti be és távolíthatja el a prototípusokat.

2. *Új objektumok megadása az értékek megváltoztatásával.* A nagymértékben dinamikus rendszerek lehetővé teszik, hogy objektum-összetételen keresztül – például értékeket adva az objektum változóinak – határozzunk meg új viselkedésmódokat, nem új osztályokat meghatározva. Ekkor tulajdonképpen a meglevő osztályok példányosításával, majd a példányokat ügyfélobjektum-prototípusokként bejegyezve határozunk meg új objektumtípusokat. Az ügyfél a prototípusra felelősséget átruházva állíthat elő új viselkedésmódot.
   Ez a tervezési mód lehetőséget ad arra, hogy a felhasználók programozás nélkül határozzanak meg új „osztályokat". Valójában a prototípusok klónozása hasonló az osztályok példányosításához. A Prototípus minta nagymértékben csökkentheti a rendszer által igényelt osztályok számát. A kottaszerkesztős példában a GrafikusEszköz osztály a zenei objektumok korlátlan változatosságát képes létrehozni.
3. *Új objektumok megadása a szerkezet megváltoztatásával.* Sok alkalmazás elemekből és elemrészekből építi fel az objektumokat. Az áramkörtervező programok például részáramkörökből állítják össze az áramköröket[1]. Az ilyen alkalmazások kényelmi szempontok miatt gyakran lehetővé teszik összetett, felhasználó által meghatározott szerkezetek példányosítását, mondjuk azért, hogy egy részáramkört többször is fel lehessen használni.
   A Prototípus minta ezt a megoldást is támogatja. A részáramkört egyszerűen csak hozzáadjuk prototípusként a különféle áramköri elemek választékához. Amennyiben az összetett áramköri objektumok a klónozás megvalósításakor mélymásolást végeznek (azaz az objektum minden alszerkezetét tartalmazó másolatot készítenek), a különféle felépítésű áramkörök prototípusokká válhatnak.
4. *Kevesebb alosztályra van szükség.* A Gyártófüggvény minta gyakran a termékosztály hierarchiájával párhuzamos Létrehozó osztályhierarchiát hoz létre. A Prototípus minta lehetőséget ad a prototípusok klónozására, ahelyett, hogy egy gyártófüggvényt kérne meg arra, hogy új objektumot hozzon létre, emiatt a Létrehozó osztályhierarchiára egyáltalán nincs is szükség. Ez elsősorban a C++ nyelvhez hasonló nyelveknél jelent előnyt, amelyek nem első osztályú objektumokként kezelik az osztályokat. Azon nyelvek esetében, amelyek viszont igen – ilyen például a Smalltalk és az Objective C –, kevesebb haszon származik ebből, mivel az osztályobjektumok mindig használhatók létrehozóként. Az osztályobjektumok ezekben a nyelvekben már önmagukban prototípusokként viselkednek.
5. *Az alkalmazás dinamikus beállítása osztályokkal.* Egyes futásidejű környezetek lehetővé teszik az osztályok dinamikus betöltését az alkalmazásokba. A Prototípus minta kulcsszerepet tölthet be ezen képességek kiaknázásában a C++ és az ahhoz hasonló nyelvekben.
   Azok az alkalmazások, amelyek egy dinamikusan betöltött osztály példányait szeretnék létrehozni, nem képesek statikusan hivatkozni annak konstruktorára, ehelyett a futásidejű környezet hoz létre automatikusan egy példányt minden osztályból, amikor az betöltődik, és bejegyzi a példányt egy prototípus-kezelőben (lásd a Meg-

[1] Az ilyen alkalmazások az Összetétel, illetve a Díszítő mintát követik.

valósítás részt). Ezután az alkalmazás elkérheti a prototípus-kezelőtől az újonnan betöltött osztályok példányait, olyan osztályokét, amelyeket eredetileg nem voltak a programba szerkesztve. Az ET++ alkalmazás-keretrendszer [WGM88] futásidejű rendszere ezt a sémát használja.

A Prototípus minta legfőbb felelőssége, hogy minden egyes Prototípus alosztály megvalósítsa a Klónoz műveletet, ami nem mindig egyszerű. Nehéz például megírni a műveletet olyankor, ha a kérdéses osztályok már léteznek, de megvalósítása akkor is nehéz lehet, ha olyan objektum van bennük, amely nem támogatja a másolást, vagy körkörös hivatkozásokat tartalmaznak.

Megvalósítás

A Prototípus minta különösen a statikus nyelveknél hasznos, amilyen például a C++, ahol az osztályok nem objektumok, és futásidőben kevés vagy semmilyen típusinformáció nem áll rendelkezésre. Az olyan nyelveknél, mint a Smalltalk vagy az Objective C, amelyek a prototípusokkal egyenértékű objektumokat (azaz osztályobjektumokat) biztosítanak az osztályok példányainak létrehozásához, kisebb a jelentősége. A mintát a prototípusokra épülő nyelvek – amilyen a Self [US87] –, amelyekben minden objektumlétrehozás egy prototípus klónozása útján valósul meg, beépítve tartalmazzák.

A prototípusok megvalósítása során tartsuk szem előtt a következőket:

1. *Prototípus-kezelő használata.* Amikor egy rendszerben nincs rögzítve a prototípusok száma (azaz dinamikusan lehet őket létrehozni és megsemmisíteni), a rendelkezésre álló prototípusokról nyilvántartást kell vezetni. Az ügyfelek maguk nem kezelik a prototípusokat, csak a nyilvántartóba mentik azokat, és beolvassák onnan. Az ügyfél elkéri a nyilvántartóból a prototípust, mielőtt klónozná. Ezt a nyilvántartót nevezzük **prototípus-kezelőnek**.
   A prototípus-kezelő egy társításos tároló (asszociatív tár), amely egy adott kulcshoz tartozó prototípust ad vissza. Vannak benne műveletek a prototípusok kulcsokhoz történő bejegyzésére és a bejegyzés törlésére. Az ügyfelek futásidőben módosíthatják a nyilvántartót, illetve tallózhatnak is abban. Ez lehetővé teszi az ügyfelek számára, hogy kódírás nélkül felvegyék a rendszer leltárát.
2. *A Klónoz művelet megvalósítása.* A Prototípus minta legnehezebb része a Klónoz művelet helyes megvalósítása. Különösen trükkös olyankor, amikor az objektumszerkezetek körkörös hivatkozásokat tartalmaznak.
   A legtöbb nyelv támogatja valamennyire az objektumklónozást. A Smalltalk például a `copy` művelet egy megvalósítását biztosítja ehhez, amelyet aztán az Object (Objektum) osztály minden alosztálya örököl. A C++ egy másoló konstruktort tartalmaz. Ezek a lehetőségek azonban nem oldják meg a „sekély másolat vagy mélymásolat”

problémát [GR83], azaz hogy az objektum klónozásakor valóban másolat készül-e a példányváltozókról, vagy csak meg lesznek osztva a klónozás után az eredeti példány változói.

A „sekély" másolat készítése egyszerű és gyakran elegendő is, ezt csinálja alapértelmezés szerint a Smalltalk. A C++ alapértelmezett másoló konstruktora tagszerű másolást végez, ami azt jelenti, hogy a mutatók meg lesznek osztva a másolat és az eredeti között. Az összetett felépítésű prototípusok klónozása során azonban általában mélymásolat készítésére van szükség, mivel a klónnak és az eredeti példánynak egymástól függetlennek kell lennie. Ezért biztosítani kell, hogy a klón elemei a prototípus elemeinek klónjai legyenek. A klónozás rákényszerít minket, hogy eldöntsük, mit akarunk megosztva használni, már ha egyáltalán van ilyen.

Ha a rendszerben lévő objektumokhoz tartozik Ment (Save) és Betölt (Load) művelet, akkor használhatók azok a Klónoz alapértelmezett megvalósításának előállítására, ehhez nem is kell mást tenni, csak menteni az objektumot, majd azonnal újra betölteni. A Ment művelet egy memóriatárba menti az objektumot, a Betölt pedig másolatot készít belőle, újra előállítva az objektumot a tárból.

3. *A klónok előkészítése.* Miközben egyes ügyfelek tökéletesen elégedettek a klónnal úgy, ahogy az van, mások általuk választott kezdőértékekkel akarják ellátni annak néhány vagy az összes belső állapotát. Ezeket az értékeket a klónozási művelettel általában nem lehet átadni, mert a számuk prototípus-osztályonként változó. Egyes prototípusoknak több előkészítő paraméterre is szükségük lehet, másoknak egyre sincs. A Klónoz művelet során történő paraméterátadás eleve kizárná egy egységes klónozó felület használatának lehetőségét.

   Az is megeshet, hogy a használandó prototípus-osztályok már előre meghatároznak bizonyos műveleteket az állapot kulcsértékeinek be- vagy alaphelyzetbe állításához. Ha ez a helyzet, az ügyfelek ezeket a műveleteket a klónozás megtörténte után azonnal használhatják, egyéb esetben viszont esetleg nekünk kell bevezetni az `Initialize` (Előkészít) műveletet (lásd a Példakód részt), amely az előkészítő paramétereket kapja argumentumként, és azok alapján beállítja a klón belső állapotát. Vigyázzunk, hogy a klónozó műveletekből ne készítsünk mélymásolatokat – ezeket ugyanis esetleg törölni kell (akár kifejezetten, akár az `Initialize` műveleten belül), mielőtt újra előkészíthetnénk őket.

Példakód

A korábban már látott `MazeFactory` osztály `MazePrototypeFactory` (LabirintusPrototípusGyár) alosztályát fogjuk létrehozni. A `MazePrototypeFactory` alosztályt az általa létrehozandó objektum prototípusaival fogjuk előkészíteni, hogy ne kelljen alosztályokat készíteni belőle a létrehozott falak és szobák megváltoztatásához.

A `MazePrototypeFactory` kiegészíti a `MazeFactory` felületét egy konstruktorral, amely a prototípusokat veszi fel argumentumként:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Az új konstruktor egyszerűen csak előkészíti saját prototípusait:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

A falakat, szobákat és ajtókat létrehozó tagfüggvények hasonlóak: mindegyik klónoz, majd előkészít egy prototípust. Alább a `MakeWall` (KészítFal) és a `MakeDoor` (KészítAjtó) művelet meghatározása látható:

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

A `MazePrototypeFactory` alosztályt prototípusos és alapértelmezett labirintus létrehozására használhatjuk, mindössze annyi a teendőnk, hogy a labirintus alapösszetevőinek prototípusaival állítsuk be:

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

A labirintus típusának megváltoztatásához másféle prototípushalmazzal állítjuk be a `MazePrototypeFactory` alosztályt. A következő hívás egy olyan labirintust hoz létre, amelynek egyik szobájában bomba robbant (`RoomWithABomb`), és ettől kiszakadt az ajtó is (`BombedDoor`):

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

A prototípusként használható objektumnak – ez lehet például a `Wall` osztály egy példánya – támogatnia kell a `Clone` műveletet, és rendelkeznie kell egy másoló konstruktorral, ami klónozza. Emellett szüksége van még egy külön műveletre, amely újra előkészíti a belső állapotot. Az `Initialize` műveletet a `Door` osztályhoz adjuk hozzá, hogy az ügyfél előkészíthesse a klónozott szobákat.

Hasonlítsuk össze a `Door` osztály alábbi meghatározását a fejezet elején találhatóval:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}
```

```
void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

A `BombedWall` alosztálynak felül kell bírálnia a `Clone` műveletet, és meg kell valósítania egy megfelelő másoló konstruktort:

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Bár a `BombedWall::Clone` művelet egy `Wall*` mutatót ad vissza, megvalósítása egy olyan mutatót, amely egy új alosztálypéldányra (`BombedWall*`) mutat. A `Clone` műveletet az alaposztályban határozzuk meg így, ezzel biztosítva, hogy a prototípust klónozó ügyfeleknek ne kelljen tudniuk saját konkrét alosztályaikról. Az ügyfeleknek sohasem kell lefelé irányuló átalakítást végezniük a `Clone` művelet által visszaadott értéken.

A Smalltalk nyelvben az `Object` osztályból örökölt szabványos `copy` metódust újra felhasználhatjuk bármely `MapSite` osztály klónozására. A `MazeFactory` osztály a szükséges prototípusok előállítására használható: a `#room` nevet megadva például létrehozhatunk egy szobát. A `MazeFactory` alosztályhoz tartozik egy szótár is, amely a neveket a prototípusokhoz rendeli. Az alosztály `make:` metódusa így néz ki:

```
make: partName
    ^ (partCatalog at: partName) copy
```

Amennyiben adottak a `MazeFactory` prototípusokkal való előkészítéséhez szükséges metódusok, az alábbi kóddal létrehozhatunk egy egyszerű labirintust:

```
CreateMaze
    on: (MazeFactory new
        with: Door new named: #door;
        with: Wall new named: #wall;
        with: Room new named: #room;
        yourself)
```

A `CreateMaze` alosztály fenti kódban szereplő `on:` osztálymetódusának meghatározása a következő lehet:

```
on: aFactory
    | room1 room2 |
    room1 := (aFactory make: #room) location: 1@1.
    room2 := (aFactory make: #room) location: 2@1.
    door := (aFactory make: #door) from: room1 to: room2.

    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: door;
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: (aFactory make: #wall).
    room2
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: (aFactory make: #wall);
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: door.
    ^ Maze new
        addRoom: room1;
        addRoom: room2;
        yourself
```

Ismert felhasználások

A Prototípus minta felhasználásának első példája talán Ivan Sutherland Sketchpad rendszere [Sut63] lehetett. Az első széles körben ismert alkalmazás, amely a mintát egy objektumközpontú nyelvben használta, a ThingLab volt, amelyben a felhasználók egy összetett objektumot állíthattak össze, majd továbbadhatták azt egy prototípusnak úgy, hogy egy olyan könyvtárban helyezték el, amelyben újrafelhasználható objektumok voltak találhatók [Bor81]. Goldberg és Robson is említi a mintaként használható prototípusokat [GR83], de Coplien [Cop92] sokkal teljesebb leírást ad róluk, ismertetve a C++ nyelvben a Prototípus mintához kapcsolódó megoldásokat, és bemutat számos példát és változatot.

Az etgdb egy hibakereső-felületi alkalmazás, amely az ET++-on alapul, és grafikus (egérműveletekkel kezelhető) felhasználói felületet biztosít különféle programsorból futtatható hibakeresőkhöz. Mindegyik hibakeresőnek megvan a megfelelő DebuggerAdaptor (HibakeresőIllesztő) alosztálya. A GdbAdaptor például a GNU gdb nyelvtanához illeszti az etgdb programot, míg a SunDbxAdaptor a Sun dbx hibakeresőjéhez. Az etgdb programba nincs mereven bekódolva a DebuggerAdaptor osztályhalmaz. Ehelyett a program egy környezeti változóból olvassa ki a használandó illesztő nevét, kikeresi egy globális táblázatból az adott nevű prototípust, majd klónozza. Az etgdb programhoz új hibakereső programok is hozzáadhatók, ehhez csak az adott hibakeresőhöz tartozó DebuggerAdaptor illesztőt kell csatolnunk.

A Mode Composerben található „együttműködési könyvtár" (interaction technique library) tárolja azon objektumok prototípusait, amelyek támogatják a különféle interaktív eljárásokat [Sha90]. A Mode Composer által létrehozott összes ilyen eljárás használható prototípusként, ha a fent említett könyvtárban helyezzük el. A Prototípus minta lehetővé teszi, hogy a Mode Composer korlátlan számú interaktív eljárást használhasson.

A korábban említett kottaszerkesztő a Unidraw rajzoló keretrendszeren alapszik [VL90].

Kapcsolódó minták

A Prototípus és az Elvont gyár bizonyos fokig vetélytársai egymásnak, amint arról majd a fejezet végén szó lesz, de együtt is használhatók. Az Elvont gyár tárolhat egy olyan prototípushalmazt, amelynek elemeit aztán klónozzuk, és az így készült termékobjektumokat adjuk vissza.

Az Összetétel és a Díszítő mintákat sokat használó programok gyakran szintén nagy hasznát veszik a Prototípus mintának.

Egyke

Objektum-létrehozási minta

Egyéb nevek

Singleton

Cél

Egy osztályból csak egy példányt engedélyezni, és ehhez globális hozzáférési pontot megadni.

Feladat

Egyes osztályok esetében fontos, hogy pontosan egy példány legyen belőlük. Bár egy rendszerben több nyomtató is lehet, nyomtatási sorból csak egyet szabad használni. Csak egy fájlrendszer és csak egy ablakkezelő futhat. Egy digitális szűrőhöz egyetlen analóg–digitális átalakító tartozhat. Egy könyvelőrendszer egy cég kiszolgálására van beállítva.

Hogyan biztosíthatjuk, hogy egy osztályból csak egyetlen példány legyen, viszont azt könnyen el lehessen érni? Egy globális változóval az objektum elérhetővé tehető, de ez nem jelent védelmet az ellen, hogy az objektumból több példány készüljön.

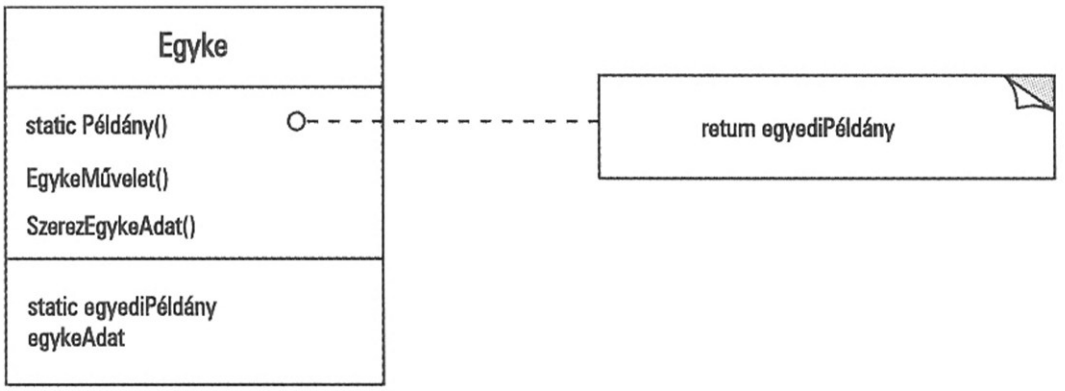
Jobb megoldás, ha magát az osztályt tesszük felelőssé annak nyilvántartásáért, hogy készült-e már példány belőle. Az osztály biztosítani tudja, hogy több példányt ne lehessen belőle létrehozni (úgy, hogy elfogja az új objektum készítésére vonatkozó kérelmeket), és képes biztosítani azt is, hogy a példányt el lehessen érni. Ez az Egyke minta.

Alkalmazhatóság

Az Egyke mintát a következő esetekben használjuk:

- Pontosan egy példányra van szükség valamelyik osztályból, és annak elérhetőnek kell lennie az ügyfelek számára a jól ismert elérési pontokból.
- Ennek az egyetlen példánynak alosztályokkal bővíthetőnek kell lennie, és az ügyfeleknek képeseknek kell lenniük saját kódjuk módosítása nélkül használni a bővített példányt.

Szerkezet



Résztvevők

- Egyke
  - Meghatároz egy olyan Példány (Instance) műveletet, amely lehetővé teszi, hogy az ügyfelek hozzáférjenek az osztály egyedi példányához. A Példány osztályművelet (azaz osztálymetódus a Smalltalk nyelvben, illetve statikus tagfüggvény a C++-ban).
  - Felelős lehet saját egyedi példányának létrehozásáért.

Együttműködés

- Az ügyfelek az Egyke példányt kizárólag az Egyke Példány műveletén át érik el.

Következmények

Az Egyke mintának számos előnye van:

1. *Szabályozott hozzáférés az egyetlen példányhoz.* Mivel az Egyke osztály magába zárja saját egyetlen példányát, szigorúan szabályozhatja, hogy az ügyfelek mikor és hogyan férhessenek hozzá.
2. *Csökkentett névtér.* Az Egyke minta jobb a globális változóknál, mert elkerülhető vele a névtérnek az egyetlen példányt tároló globális változókkal szennyezése.
3. *Megengedi a műveletek és a megjelenítés finomítását.* Az Egyke osztályból létrehozhatók alosztályok, és az alkalmazások futásidőben egyszerűen beállíthatók a szükséges bővített osztály egy példányával.
4. *Megengedi változó számú példány használatát.* A minta megkönnyíti, hogy ha meggondoljuk magunkat, az Egyke osztályból egynél több példány használatát is engedélyezzük. Emellett ugyanezt a megközelítést használhatjuk az alkalmazás által használt példányok számának szabályozására is, ehhez csak azt a műveletet kell megváltoztatnunk, amely az Egyke példányhoz való hozzáférést engedélyezi.

5. *Rugalmasabb, mint az osztálymüveletek.* Az egykék szolgáltatásait osztálymüveletekkel is megvalósíthatjuk, azaz a C++ nyelvben statikus tagfüggvényeket, a Smalltalkban osztálymetódusokat használva, de mindkét megoldás megnehezíti a kialakítás oly módon történő megváltoztatását, hogy az megengedje egy osztály több példányának használatát is. Ezenkívül a C++ nyelvben a statikus tagfüggvények sohasem virtuálisak, így az alosztályok nem tudják azokat a többalakúság segítségével felülbírálni.

Megvalósítás

Az Egyke minta megvalósításánál az alábbiakat kell szem előtt tartanunk:

1. *Egyedi példány biztosítása.* Az Egyke minta az egyetlen példányt normál osztálypéldánnyá alakítja, de ezt az osztályt úgy kell megírni, hogy mindig csak egy példányt lehessen belőle létrehozni. Erre gyakran használt módszer, hogy a példányt létrehozó műveletet elrejtik egy olyan osztálymüvelet (azaz statikus tagfüggvény vagy osztálymetódus) mögé, ami garantálja, hogy csak egy példányt lehet létrehozni egy osztályból. Ez a művelet hozzáférhet ahhoz a változóhoz, amely az egyedi példányt tárolja, és gondoskodik arról, hogy a változó értékének visszaadása előtt az egyedi példányt kapja kezdőértékként. Ez a megközelítés biztosítja azt, hogy az egykét már első használatba vétele előtt létrehozza és előkészíti a program.
   Az osztálymüvelet a C++ nyelvben a `Singleton` osztály `Instance` statikus függvényével határozható meg. A `Singleton` osztály meghatározza az `_instance` statikus tagváltozót is, amely az osztály egyetlen példányát címző mutatót tartalmazza.
   A `Singleton` osztályt a következőképpen vezetjük be:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

   Az ennek megfelelő megvalósítás:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

   Az ügyfelek az egykéket kizárólag az `Instance` tagfüggvényen át érhetik el. Az `_instance` változó kezdőértéke 0, az `Instance` statikus tagfüggvény pedig ennél az értéknél az egyedi példányra állítja, és visszaadja az értékét. Az `Instance` tag-

függvény lusta előkészítést használ, az általa visszaadott értéket a program nem hozza létre és nem tárolja addig, amíg először hozzá nem férnek a példányhoz.

Figyeljük meg, hogy a konstruktor védett. Ha egy ügyfél közvetlenül próbál példányt készíteni a `Singleton` osztályból, fordításkor hibaüzenetet kap. Ez biztosítja azt, hogy mindig csak egy példányt lehessen létrehozni.

Ezenkívül, mivel az `_instance` egy egyke objektumra hivatkozó mutató, az `Instance` tagfüggvény egy Egyke alosztályt címző mutatót rendelhet e változóhoz, amint azt a Példakód részben is láthatjuk.

Van még valami, amit érdemes megjegyezni a C++ nyelven történő megvalósítással kapcsolatban. Nem elég az egykét globális vagy statikus objektumként meghatározni, majd az automatikus előkészítésben bízni. Ennek három oka van:

(a) Nem garantálható, hogy a statikus objektumnak mindig csak egy példánya lesz bevezetve.

(b) Lehet, hogy a statikus előkészítés idején nem lesz elég információnk minden egyke példányosításához. Az egyke kérhet olyan értékeket, amelyeket a program a végrehajtás egy későbbi szakaszában számít ki.

(c) A C++ nem határozza meg, hogy a globális objektumok konstruktorait milyen sorrendben kell meghívni a fordítási egységekben [ES90]. Ez azt jelenti, hogy az egykék közt nem lehet semmilyen függőségi viszony. Ha mégis van, a hibaüzenetek elkerülhetetlenek.

Egy további (jóllehet kicsi) felelősség a globális–statikus objektumos megközelítésben, hogy kényszerűen létre kell hozni az egykéket, ha szükség van rájuk, ha nincs. Statikus tagfüggvényt használva minden ilyen gond elkerülhető.

A Smalltalk nyelvben az egyedi példányt visszaadó függvény a Singleton osztály osztálymetódusa. Annak biztosítására, hogy csak egyetlen példány készüljön, a `new` műveletet bíráljuk felül. Az így kapott egykeosztálynak a következő két osztálymetódusa lehet, ahol a `SoleInstance` (EgyetlenPéldány) egy olyan osztályváltozó, amelyet sehol máshol nem használunk:

```
new
    self error: 'nem hozható létre új objektum'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Alosztályok készítése az Egyke osztályhoz.* A fő gond nem is igazán az alosztályok meghatározása, sokkal inkább az egyedi példány oly módon történő telepítése, hogy az ügyfelek képesek legyenek azt használni. Lényegében az egykepéldányra hivatkozó változónak az alosztály egy példányát kell adni kezdőértékként. A legegyszerűbb módszer, ha meghatározzuk, melyik egykét szeretnénk használni az Egyke osztály Példány (`Instance`) műveletével. A Példakód részben egy példa szemlélteti, hogy ezt a módszert hogyan lehet környezeti változók segítségével megvalósítani.

   Az Egyke osztály alosztályának kiválasztására egy másik módszer, ha az `Instance` művelet megvalósítását kivesszük a szülőosztályból (például a `MazeFactory` osz-

tályból), és egy alosztályba helyezzük. Ekkor a C++ programozó összeszerkesztéskor döntheti el (például egy másfajta megvalósítást tartalmazó objektumfájlhoz csatolva), hogy melyik egykeosztályt akarja használni, de továbbra is rejtve tartja az egykét az azt használó ügyfelek elől.

Ez a megközelítés az összeszerkesztésre időzíti az egykeosztály kiválasztását, ami megnehezíti azt, hogy az egykeosztályt futásidőben választhassuk ki. Ha feltételes utasításokat használunk az alosztály meghatározására, az rugalmasabb, de mereven bekódolja a választható Egyke osztályokat. Egyik megoldás sem elég rugalmas ahhoz, hogy minden esetben kielégítő eredményt adjon.

Sokkal rugalmasabb módszer ezeknél az **egykenyilvántartó** használata. Ahelyett, hogy egy `Instance` művelettel határoznánk meg a választható Egyke osztályokat, azok név szerint bejegyeztethetik egykepéldányukat egy ismert nyilvántartóba.

A nyilvántartó elvégzi a név-karakterláncok és az egykék egymáshoz rendelését. Amikor az `Instance` műveletnek egy egykére van szüksége, a nyilvántartóhoz fordul, és név szerint kéri a kívánt egykét. A nyilvántartó megkeresi a megfelelőt (ha az létezik), és visszaadja az `Instance` műveletnek. Ezt a megközelítést használva az `Instance` műveletnek nem kell ismernie az összes választható egykeosztályt vagy példányt, az egyetlen követelmény, hogy legyen egy olyan közös felület az összes Egyke (Singleton) osztályhoz, amely tartalmazza a nyilvántartóval kapcsolatos műveleteket:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

A `Register` (Nyilvántartó) a megadott néven tartja nyilván a Singleton példányt. Hogy a nyilvántartó egyszerű maradjon, tároltatnunk kell vele egy `NameSingletonPair` (NévEgykePár) objektumlistát, amelyben minden `NameSingletonPair` elem egy nevet rendel egy egykéhez. A `Lookup` (Keresés) művelet a nevük alapján keresi meg az egykéket. Tételezzük fel, hogy a keresett egyke nevét egy környezeti változó adja meg:

```
Singleton* Singleton::Instance () {
  if (_instance == 0) {
      const char* singletonName = getenv("SINGLETON");
      //ezt a felhasználó vagy a környezet adja meg indításkor

      _instance = Lookup(singletonName);
      //Ha nincs ilyen egyke, a Lookup művelet 0-t ad vissza
  }
  return _instance;
}
```

Hol jegyeztetik be magukat a Singleton osztályok? Például a konstruktorukban (mint egyik lehetőség). A `MySingleton` alosztály például a következőt teheti:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Természetesen a konstruktort nem lehet meghívni, csak ha valaki előbb példányosítja az osztályt – ami ugyanazt a problémát tükrözi, mint amit az Egyke minta megpróbál megoldani! A C++ nyelvben a probléma úgy kerülhető meg, ha egy statikus példányt határozunk meg a `MySingleton` alosztályból. A `MySingleton` megvalósítását tartalmazó fájlban meghatározhatjuk például ezt:

```
static MySingleton theSingleton;
```

A Singleton osztály ezek után már nem felelős az egyke létrehozásáért, helyette elsődleges felelőssége az, hogy a tetszés szerinti egykeobjektumot elérhetővé tegye a rendszerben. Ennek a statikus objektumos megközelítésnek még mindig van egy lehetséges hátulütője – nevezetesen az, hogy az összes lehetséges Egyke alosztály példányait létre kell hozni, különben nem lesznek bejegyezve a nyilvántartóba.

Példakód

Tételezzük fel, hogy egy `MazeFactory` osztályt határozunk meg labirintusok létrehozásához, amint azt a fejezet elején láthattuk. A `MazeFactory` osztály egy felületet határoz meg, amelynek segítségével a labirintus különféle elemei hozhatók létre. Az alosztályok felülírják a műveleteket, hogy egyedi célú termékosztály-példányokat adjanak vissza, amilyenek például a `BombedWall` objektumok az egyszerű `Wall` objektumok helyén.

Ami itt fontos, az az, hogy a Maze alkalmazásnak a labirintusgyárakból csak egy példányra van szüksége, és hogy ennek a példánynak hozzáférhetőnek kell lennie a labirintus bármelyik részét felépítő kódok számára. Ez az a pont, ahol az Egyke minta szerepet kap. Ha a `MazeFactory` osztályt egykeként hozzuk létre, a labirintusobjektum mindenhonnan elérhető lesz globális változók használata nélkül is.

Az egyszerűség kedvéért tételezzük fel, hogy soha nem kell alosztályt készítenünk a `MazeFactory` osztályból (kisvártatva a másik lehetőséget is meg fogjuk vizsgálni). Ekkor a `MazeFactory` osztályt Egyke osztállyá változtatjuk, a C++ nyelvben egy statikus `Instance` műveletet és az egyetlen példányt tároló statikus `_instance` tagot adva hozzá. A konstruktort védenünk is kell, hogy megakadályozzuk a véletlen példányosítást, aminek következtében esetleg több példány jöhetne létre.

```
class MazeFactory {
public:
    static MazeFactory* Instance();
```

```
    //ide jön a már meglévő felület
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

A megfelelő megvalósítás a következő:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Most vegyük azt az esetet, amikor alosztályokat is készítünk a `MazeFactory` osztályból, és az alkalmazásnak kell eldöntenie, hogy melyiket használja. A labirintus típusát egy környezeti változó segítségével választjuk ki, és a kódot kiegészítjük egy olyan kódrésszel, amely a megfelelő `MazeFactory` alosztályokat példányosítja a környezeti változó értéke alapján. Az `Instance` művelet kiváló hely ennek a kódnak a hozzáadására, mivel már önmagában is példányosítja a `MazeFactory` osztályt:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        //...további lehetséges alosztályok

        } else {        // alapértelmezés
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Figyeljük meg, hogy az `Instance` műveletet minden esetben módosítani kell, amikor új alosztályt határozunk meg a `MazeFactory` osztályhoz. Ez ebben az alkalmazásban talán nem gond, de a keretrendszerrel meghatározott elvont gyárak esetében az lehet.

Az egyik lehetséges megoldás lehet a Megvalósítás részben ismertetett nyilvántartó használata. Hasznos lehet a dinamikus kötés is, ami megakadályozza, hogy az alkalmazás a nem használt alosztályokat is betöltse.

Ismert felhasználások

A Smalltalk 80-ban [Par90] használt Egyke mintára egy példa a `ChangeSet current` nevű kódmódosítás-halmaz. Egy kifinomultabb példa az osztályok és **metaosztályaik** közötti kapcsolat. A metaosztály az osztályok osztálya, és mindegyik metaosztálynak egy példánya van. A metaosztályoknak nincs nevük (kivétel: közvetve az egyetlen példányukon keresztül), de nyilvántartják az egyetlen példányukat, és normál esetben nem hoznak létre másikat.

Az InterViews felhasználói felületi elemkészlet [LCI+92] többek közt Session és WidgetKit osztályai egyedi példányainak elérésére használ egykéket. A Session (Munkamenet) az alkalmazás fő eseményelosztó ciklusát határozza meg, tárolja a felhasználó stílusbeállításait tartalmazó adatbázist, és kezeli az egy vagy több fizikai kijelzővel való kapcsolatokat. A WidgetKit egy Elvont gyár, amely a felhasználói felület vizuális kialakításában szereplő elemeket határozza meg. A `WidgetKit::Instance()` művelet azt az adott WidgetKit alosztályt határozza meg, amelyet egy, a Session osztály által meghatározott környezeti változó alapján példányosít a rendszer. A Session osztályon végrehajtott hasonló művelet meghatározza, hogy a rendszer a monokróm vagy a színes kijelzőket támogatja-e, és ennek megfelelően beállítja a Session egykepéldányt.

Kapcsolódó minták

Az Egyke mintával számos minta megvalósítható, például az Elvont gyár, az Építő és a Prototípus.

A létrehozási mintákról

A rendszernek az általa létrehozott osztályokkal történő paraméterezésére általában két módszert használnak. Az egyik az alosztályok készítése az objektumot létrehozó osztályokból, ennek megfelelője a Gyártófüggvény minta használata. Ennek a módszernek a legfőbb hátulütője, hogy esetenként új alosztályt kell létrehozni csak azért, hogy megváltoztassuk a termékosztályt, és ezek a módosítások halmozódhatnak. Ha például magát a terméklétrehozót is gyártófüggvénnyel hoztuk létre, akkor felül kell bírálnunk a készítő osztályt is.

A rendszer paraméterezésének másik módja sokkal inkább az objektum-összetételen alapszik: meghatározunk egy objektumot, amely felelős azért, hogy ismerje a termékobjektumokat, és ezt az objektumot használjuk rendszerparaméterként. Ez a kulcsa az Elvont gyár, az Építő és a Prototípus mintáknak. Mindhárom mintánál létrehozunk egy új „gyárobjektumot", amelynek felelőssége a termékobjektumok létrehozása. Az Elvont gyár mintában a gyárobjektum több osztály objektumait állítja elő. Az Építő mintában a gyárobjektum fokozatosan összetett terméket készít, egy megfelelően összetett protokollt használva. A Prototípus mintában a gyárobjektum a prototípusobjektumot lemásolva hozza létre a termékeket. Ebben az esetben a gyárobjektum és a prototípus ugyanaz az objektum, mivel a prototípus felel a termék visszaadásáért.

Vegyük most a Prototípus mintánál említett rajzszerkesztő keretrendszert. A GrafikusEszköz osztályt a termékosztály többféleképpen is paraméterezheti:

- A Gyártófüggvény mintával a GrafikusEszköz osztály egy alosztálya jön létre a palettán lévő minden Grafika alosztályhoz. A GrafikusEszköz osztálynak lesz egy ÚjGrafika (NewGraphic) művelete, amelyet minden GrafikusEszköz alosztály felülír.
- Az Elvont gyár mintával a GrafikaGyár (GraphicsFactory) osztályok osztályhierarchiája jön létre, minden Grafika alosztályhoz egy. Ebben az esetben mindegyik gyár csak egy terméket állít elő: a KörGyár (CircleFactory) köröket, a VonalGyár (LineFactory) vonalakat stb. A GrafikusEszköz osztályok paramétere az adott Grafika alosztálytípust előállító gyár lesz.
- A Prototípus mintában a Grafika osztály minden alosztálya a Klónoz (Clone) műveletet valósítja meg, és a GrafikusEszköz osztályok az általuk létrehozott Grafika osztály prototípusát kapják paraméterként.

Az, hogy melyik mintát a legcélszerűbb alkalmazni, számos tényezőtől függ. A rajzszerkesztő keretrendszerben a Gyártófüggvény minta használata tűnik elsőre a legkönnyebbnek. Egyszerű a GrafikusEszköz osztályhoz új alosztályt meghatározni, és a GrafikusEszköz példányait csak akkor hozza létre a program, amikor a paletta már elkészült. A fő hátrány itt az, hogy a GrafikusEszköz alosztályok elburjánzanak, és egyik sem végez túl sok munkát.

Az Elvont gyár sem sokkal jobb, mert ehhez egy ugyanolyan nagy GrafikaGyár osztályhierarchia szükséges. Az Elvont gyár a Gyártófüggvényhez képest csak akkor jelent előnyt, ha a GrafikaGyár hierarchia már létezik – vagy azért, mert a fordítóprogram automatikusan biztosítja (mint a Smalltalk és az Objective C nyelvben), vagy azért, mert a rendszer egy másik részéhez szükség van rá.

Mindent összevetve a rajszerkesztő keretrendszerhez valószínűleg a Prototípus minta a legmegfelelőbb, mert ezt használva csak a Klónoz műveletet kell megvalósítani minden Graphics osztályon. Ezzel csökken az osztályok száma, és a Klónoz művelet a puszta példányosításon kívül más célokra is használható (például egy Másolatkészítés nevű menüpont műveleteként).

A Gyártófüggvény mintát használva testreszabhatóbbak lesznek a programok, és csak alig valamivel bonyolultabbak. A többi tervezési minta új osztályokat igényel, míg a Gyártófüggvény csak egy új műveletet. A Gyártófüggvény mintát gyakran használják objektumok létrehozásának szokásos megoldásaként, de ha a példányosított osztály soha nem változik meg, vagy ha a példányosítás olyan műveleten belül történik, amelyet az alosztályok könnyen felülbírálhatnak (amilyen például az előkészítési művelet), akkor nincs rá szükség.

Az Elvont gyár, a Prototípus és az Építő mintát használó programok még rugalmasabbak, mint a Gyártófüggvényt használók, de bonyolultabbak is. A tervezés gyakran a Gyártófüggvény mintát használva indul el, és onnan fejlődik tovább a többi létrehozási minta használatának irányába, ahogy a programtervező rájön, hogy nagyobb rugalmasságra van szükség. Ha több tervezési mintát ismerünk, több lehetőségünk lesz, hogy válasszunk közülük.

4

Szerkezeti minták

A szerkezeti minták középpontjában az áll, hogy az osztályokból és objektumokból hogyan alkothatunk nagyobb szerkezeteket. A szerkezeti *osztálypminták* örökléssel felületeket vagy megvalósításokat építenek fel, egyszerű példaként gondoljunk csak arra, hogyan lehet többszörös örökléssel két vagy több osztályt egybe „keverni", amely aztán szülőosztályainak tulajdonságait egyesíti. A minta különösen hasznos lehet, ha önállóan fejlesztett osztálykönyvtárak együttműködését szeretnénk biztosítani. Egy másik példa az Illesztő minta osztályformája: az illesztő általában arra szolgál, hogy egy felületet (az illesztendő felületét) egy másikhoz igazítson, s így különböző felületek egységes elvont ábrázolását nyújtsa. Az osztályillesztők ezt egy illesztendő osztályból való privát örökléssel érik el; az illesztő ezután felületét az illesztendő nyomán határozza meg.

A szerkezeti *objektumminták* felületek vagy megvalósítások összetétele helyett azt írják le, hogyan ragaszthatunk össze objektumokat, hogy új szolgáltatásokat nyújtsunk. Az objektum-összetétel rugalmasságát az összetétel futásidejű megváltoztathatósága biztosítja, ami a statikus osztályösszetétellel nem érhető el.

Az Összetétel szerkezeti objektumminta, amelynek segítségével osztályhierarchiát építhetünk fel kétféle objektumosztályból: alapvető (primitív) és összetett objektumokból. Az alap- és összetett objektumokból tetszőlegesen bonyolult szerkezetek építhetők. A Helyettes mintában a helyettes egy másik objektum „helyőrzője" vagy helyettesítője. A helyettes számos módon használható: lehet egy távoli címtér objektumának helyi képviselője, jelölhet egy igény szerint betöltendő nagy objektumot, de arra is használható, hogy megakadályozzuk egy érzékeny objektum elérését. A helyettesekkel elérhető, hogy az objektumok egyes tulajdonságaihoz csak közvetetten férhessünk hozzá; segítségükkel e tulajdonságok korlátozhatók, bővíthetők vagy módosíthatók.

A Pehelysúlyú minta objektumok megosztására biztosít szerkezetet. Objektumokat legalább két okból oszthatunk meg: a hatékonyság növelése vagy a következetesség biztosítása céljából. A Pehelysúlyú minta a jobb tárkihasználást célozza meg. A számos objektumot használó alkalmazásoknak különösen ügyelniük kell; az objektumok lemásolása helyett azok megosztása jelentősen csökkentheti a költségeket. Mindazonáltal az objektumok csak akkor oszthatók meg, ha állapotuk nem függ a környezettől; a Pehelysúlyú objektumok nem is tárolnak környezetfüggő állapotot, a feladatuk elvégzéséhez igényelt kiegészítő információkat akkor kapják meg, amikor szükségük van rájuk. Környezetfüggő állapot híján így a Pehelysúlyú objektumok szabadon megoszthatók.

Amíg a Pehelysúlyú minta azt mutatja meg, hogyan készíthetünk sok-sok apró objektumot, a Homlokzat minta arra ad megoldást, hogyan ábrázolhatunk egy teljes alrendszert egyetlen objektummal. A homlokzat objektumok halmazát képviseli, feladatait pedig úgy hajtja végre, hogy üzeneteket továbbít ezen objektumoknak. A Híd minta az elvont ábrázolást és a megvalósítást választja el egymástól, így azok egymástól függetlenül módosíthatók.

A Díszítő minta azt írja le, hogyan adhatunk objektumokhoz dinamikusan felelősségi köröket. A Díszítő olyan szerkezeti minta, amely önhívással (rekurzióval) állít össze objektumokat, így téve lehetővé korlátlan számú új szolgáltatás felvételét. Egy felhasználói felületi elemet tartalmazó Díszítő objektum például szegéllyel vagy árnyékolással, esetleg olyan szolgáltatásokkal, mint a görgethetőség vagy a nagyíthatóság egészítheti ki az elemet. Kétféle díszítés hozzáadásához egyszerűen csak be kell ágyaznunk egy Díszítő objektumot egy másikba; további egymásba ágyazással pedig még több díszítést alkalmazhatunk. Mindehhez az szükséges, hogy a Díszítő objektumok illeszkedjenek a hozzájuk tartozó elem felületéhez, és üzeneteket küldjenek annak. A Díszítő feladatát (például egy szegély rajzolását az elem köré) az üzenet továbbítása előtt és után is elvégezheti.

A szerkezeti minták közül számos kapcsolódik egymáshoz; ezeket a kapcsolatokat a fejezet végén tárgyaljuk.

Illesztő

Szerkezeti objektumminta/osztályminta

Cél

Az adott osztály felületét az ügyfelek által igényelt felületté alakítani. E módszerrel az egyébként összeférhetetlen felületű osztályok együttműködését biztosíthatjuk.

Egyéb nevek

Adapter, Burkoló (Wrapper)

Feladat

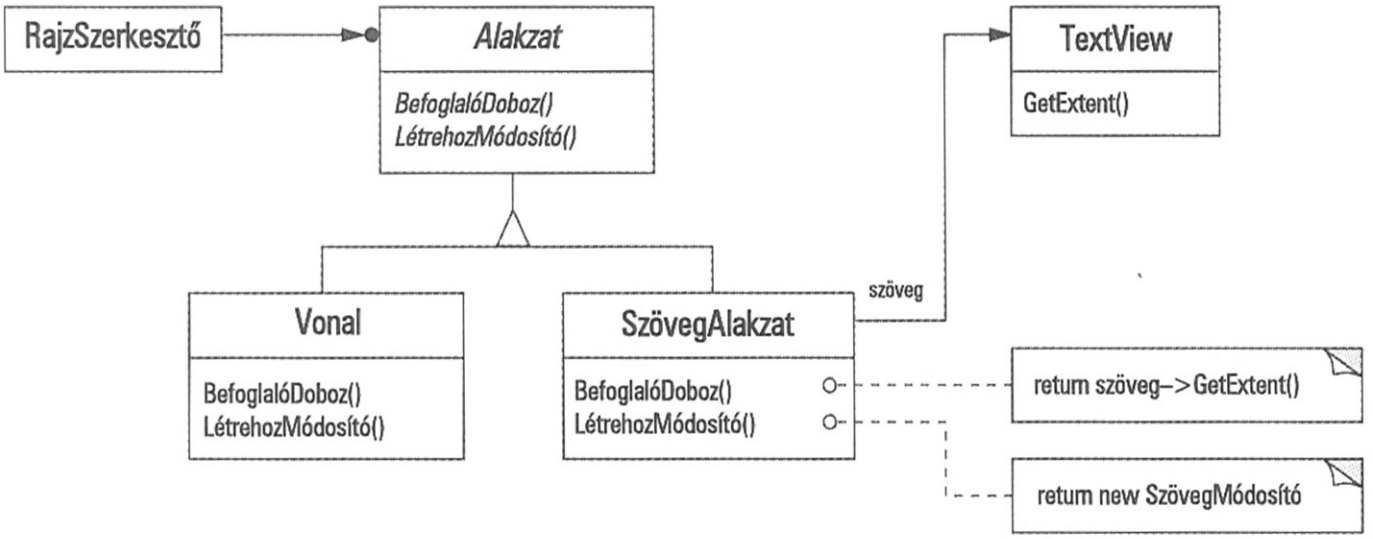
Néha előfordul, hogy egy újrahasznosításra tervezett elemkészlet mégsem használható fel, mert felülete nem felel meg az adott alkalmazás tartományfüggő felületének.

Vegyünk például egy rajzszerkesztőt, amelynek segítségével a felhasználók grafikus elemeket (vonalakat, sokszögeket, szöveget stb.) rajzolhatnak és rendezhetnek képekbe, illetve diagramokba. A rajzprogram kulcsfogalma a rajzobjektum, amelynek szerkeszthető alakja van, és képes kirajzolni önmagát. A rajzobjektumok felületét az Alakzat (Shape) elvont osztály határozza meg; a program ebből származtat alosztályokat az egyes rajzobjektum-típusok számára, így lesz például a VonalAlakzat (LineShape) a vonalak osztálya, a SokszögAlakzat (PolygonShape) a sokszögeké, és így tovább.

Az alapvető mértani alakzatok osztályai – amilyen a VonalAlakzat és a SokszögAlakzat is – viszonylag könnyen megvalósíthatók, mert rajzolási és szerkesztési lehetőségeik korlátozottak. Egy szöveg megjelenítésére és szerkesztésére képes SzövegAlakzat (TextShape) nevű alosztály megvalósítása azonban már lényegesen nehezebb, mert még az alapvető szövegszerkesztési műveletek is bonyolult képernyőfrissítés- és átmenetitár-kezelést igényelnek. Egy boltban megvásárolható felhasználói felületi elemkészletben persze valószínűleg találunk egy megfelelően bonyolult, előre elkészített TextView (SzövegNézet) osztályt, amellyel szöveget jeleníthetünk meg és szerkeszthetünk. Ideális esetben ennek újrahasznosításával megvalósíthatnánk a SzövegAlakzat osztályt – csakhogy a TextView-t nem a mi Alakzat osztályainkhoz tervezték, ezért a TextView és Alakzat objektumok egymással nem cserélhetők fel.

Hogyan használható fel tehát egy olyan „idegen" osztály, mint a TextView, egy olyan alkalmazásban, amely más, eltérő felületű osztályokat vár? Esetleg módosíthatnánk az osztályt, hogy megfeleljen az Alakzat felületnek, de az elemkészlet forráskódjának híján erről le kell tennünk. De még ha rendelkeznénk is a forrással, akkor sem lenne sok értelme a TextView módosításának, hiszen az elemkészletet nem arra szánták, hogy pusztán egy adott alkalmazás működését biztosítandó alkalmazkodjon az adott tartományra jellemző felülethez.

Ehelyett a SzövegAlakzat osztályt kell úgy meghatároznunk, hogy a TextView felületét az Alakzatéhoz *illessze*. Ezt kétféleképpen érhetjük el: (1) az Alakzat felületének és a TextView megvalósításának öröklésével, vagy (2) egy TextView példány összeállításával egy SzövegAlakzaton belül, és a SzövegAlakzatnak a TextView felület alapján történő megvalósításával. A két megközelítés megfelel az Illesztő minta osztály- és objektumváltozatának. Az Illesztő szerepét a SzövegAlakzat tölti be.



A fenti diagram az objektumillesztő esetét mutatja. Látható, hogy az Alakzat osztályban bevezetett BefoglalóDoboz (BoundingBox) kérelmeket a TextView-ban meghatározott GetExtent (SzerezKiterjedés) kérelmekké alakítjuk. Mivel a SzövegAlakzat a TextView-t az Alakzat felülethez illeszti, a rajzolóprogram felhasználhatja az egyébként nem összeegyeztethető felületű TextView osztályt.

Az illesztő gyakran olyan szolgáltatásokért felelős, amelyeket az illesztett osztály nem biztosít. A diagramon látható, hogyan képes erre. A felhasználónak például képesnek kell lennie arra, hogy az egeret húzva új helyre helyezze az Alakzat objektumokat, de a TextView nem nyújt ilyen képességet. A SzövegAlakzat úgy pótolhatja ezt a hiányosságot, hogy megvalósítja az Alakzat LétrehozMódosító (CreateManipulator) műveletét, ami a megfelelő Módosító (Manipulator) alosztály egy példányát adja vissza.

A Módosító azon objektumok elvont osztálya, amelyek tudják, hogyan mozgassanak egy Alakzat objektumot a felhasználó tevékenységével összhangban, például új helyre húzzanak egy alakzatot. A különböző alakzatokhoz külön-külön Módosító alosztályok tartoznak, a SzövegMódosító (TextManipulator) például a SzövegAlakzatnak megfelelő alosztály. Egy SzövegMódosító példány visszaadásával a SzövegAlakzat biztosíthatja a TextView-ból hiányzó, de az Alakzat által igényelt szolgáltatást.

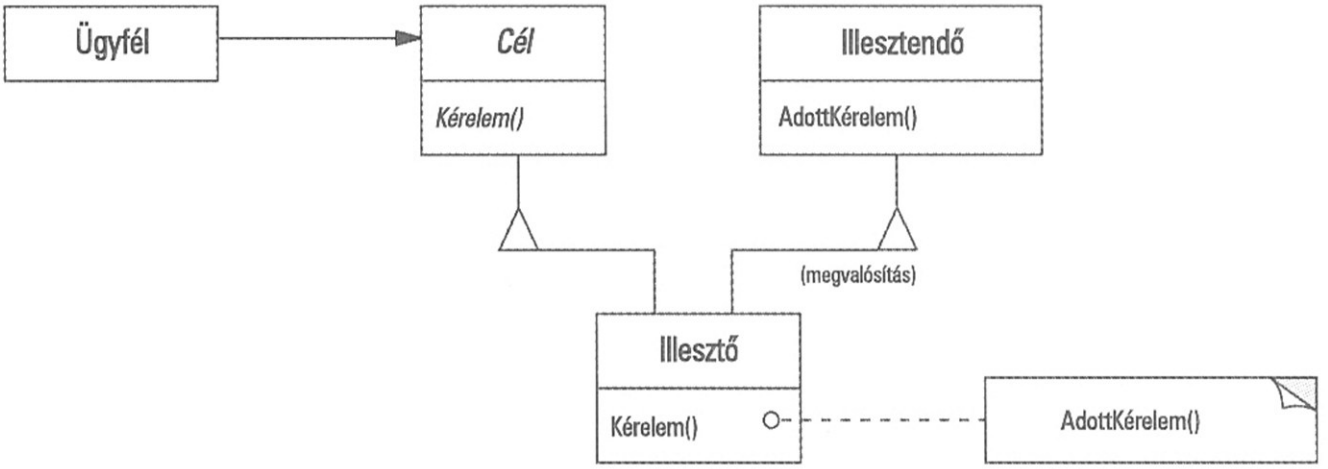
Alkalmazhatóság

Az Illesztő minta a következő esetekben alkalmazható:

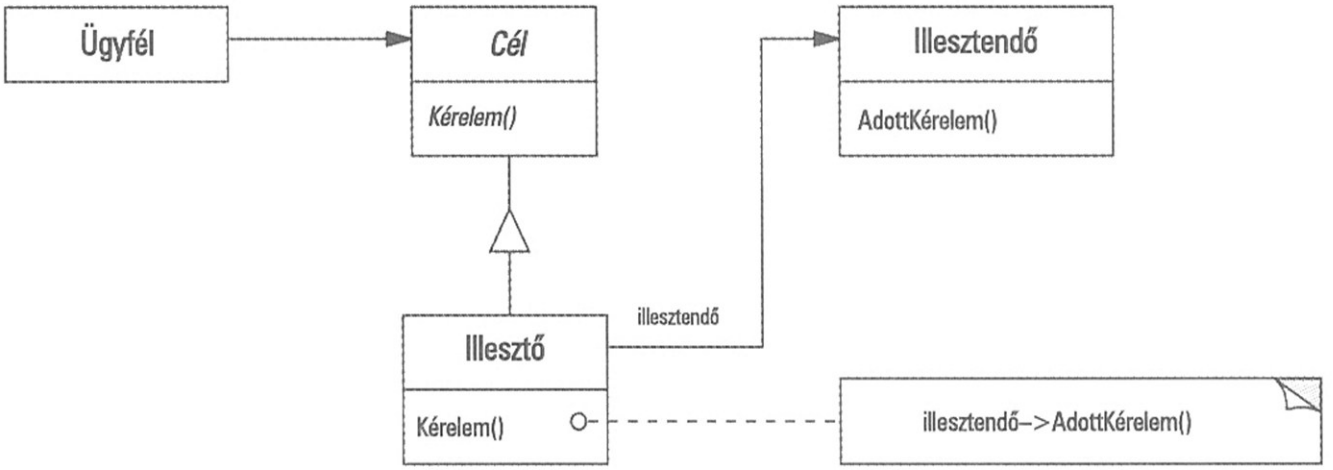
- Egy meglevő osztályt szeretnénk használni, de annak felülete nem a kívánságainknak megfelelő.
- Olyan újrahasznosítható osztályt szeretnénk készíteni, amely képes együttműködni vele kapcsolatban nem álló, vagy előre nem ismert osztályokkal, vagyis olyan osztályokkal, amelyek felülete nem feltétlenül illeszkedik.
- (*Csak az objektumillesztők esetében*) Számos már létező alosztályt kell használnunk, de felületük alosztály-származtatással történő illesztése nem praktikus. Az objektumillesztő szülőosztályának felületét képes más felületekhez illeszteni.

Szerkezet

Az osztályillesztő többszörös öröklléssel illeszt egy felületet egy másikhoz:



Az objektumillesztő objektum-összetételre épül:



Résztvevők

- **Cél** (Alakzat)
  - Meghatározza az Ügyfél által használt tartományra jellemző felületet.
- **Ügyfél** (RajzSzerkesztő)
  - Együttműködik a Cél felületnek megfelelő objektumokkal.
- **Illesztendő** (TextView)
  - Meghatároz egy már létező felületet, amelyet illeszteni kell.
- **Illesztő** (SzövegAlakzat)
  - Az Illesztendő felületét a Cél felülethez illeszti.

Együttműködés

- Az Ügyfelek egy Illesztő példányra hívnak meg műveleteket, az illesztő pedig az Illesztendő műveleteit hívja meg a kérelmek teljesítéséhez.

Következmények

Az osztály- és objektumillesztők előnyei és hátrányai különböznek. Az osztályillesztőkre a következők jellemzők:

- Az Illesztendőt egy konkrét Illesztendő osztályhoz ragaszkodva illesztik a Célhoz. Ennek következményeképpen az osztályillesztők nem működőképesek, amennyiben egy osztályt *és* annak minden alosztályát szeretnénk illeszteni.
- Lehetővé teszik, hogy az Illesztő felülbírálja az Illesztendő néhány viselkedését, hiszen az Illesztő az Illesztendő alosztálya.
- Csak egy objektumot kell bevezetni, és az illesztendőhöz való eljutáshoz nincs szükség további mutatókra.

Az objektumillesztőkkel kapcsolatban a következőket kell megjegyeznünk:

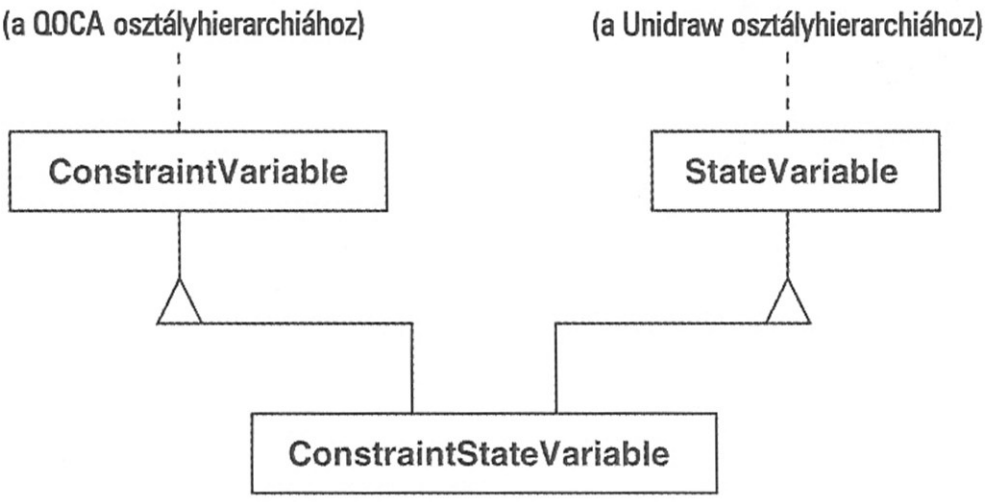
- Egyetlen Illesztő több illesztendővel is működhet, vagyis magával az Illesztendővel, és annak minden alosztályával (ha vannak ilyenek). Az Illesztő emellett minden Illesztendőt egyszerre bővíthet szolgáltatásokkal.
- Az Illesztendő viselkedését nehezebb felülbírálni. Ehhez alosztályok származtatása szükséges az Illesztendő osztályból, az Illesztőnek pedig az alosztályokra kell hivatkoznia, nem pedig közvetlenül az Illesztendőre.

Az Illesztő minta alkalmazásával kapcsolatban az alábbi kérdésekkel kell még foglalkoznunk:

1. *Mi mindent illeszt az Illesztő?* Az illesztők eltérő mennyiségű munkát végezhetnek az Illesztendőnek a Cél felülethez való illesztéséhez. A lehetséges műveletek skálája az egyszerű felületátalakítástól (például a műveletek nevének megváltoztatásával)

a teljesen különböző művelethalmazok támogatásáig terjedhet. Az Illesztő által elvégzendő munka mennyisége attól függ, mennyire hasonlít a Cél és az Illesztendő felülete.

2. *Csatlakoztatható illesztők.* Egy osztály jobban újrahasznosítható, ha a lehető legkevesebbre csökkentjük azon feltételezések számát, amelyekkel más osztályoknak élniük kell, ha az osztályt használni kívánják. Azzal, hogy a felületillesztést egy osztályba helyezzük, megszabadulunk a feltételezéstől, hogy a többi osztály ugyanazt a felületet látja. Más szavakkal, a felületillesztés lehetővé teszi, hogy osztályunkat olyan meglevő rendszerekbe építsük be, amelyek más felületet várhatnak az osztálytól. A beépített felületillesztéssel rendelkező osztályok leírására az ObjectWorks\Smalltalk [Par90] a **csatlakozhatható illesztő** (pluggable adapter) kifejezést használja.
   Vegyünk egy FaNézet (TreeDisplay) grafikus vezérlőt, amely faszerkezetek grafikus megjelenítésére képes. Ha ezt a vezérlőt kifejezetten egy adott alkalmazásban való használatra szánták, az általa megjelenített objektumoktól egy bizonyos felületet követelhet meg, mondjuk mindegyiknek a Fa (Tree) nevű elvont osztályból kell származnia. Ha a FaNézetet újrahasznosíthatóbbá szeretnénk tenni (mondjuk egy hasznos vezérlőket tartalmazó elemkészletbe szeretnénk építeni), ez a követelmény ésszerűtlen. Az egyes alkalmazások a faszerkezetekre saját osztályokat határoznak meg, így nem szabad, hogy arra kényszerítsük őket, hogy a mi Fa elvont osztályunkat használják, hiszen a különböző faszerkezetek felülete is különböző.
   Egy könyvtárhierarchiában a gyermekeket a SzerezAlkönyvtárak (GetSubdirectories) művelettel érhetjük el, míg egy öröklési hierarchiában a megfelelő művelet neve SzerezAlosztályok (GetSubclasses) lehet. Egy újrahasznosítható FaNézet vezérlőnek mindkét típusú hierarchiát meg kell tudnia jeleníteni, akkor is, ha azok felülete különbözik. Tehát a FaNézetnek beépített felületillesztést kell tartalmaznia.
   A Megvalósítás részben a felületillesztés osztályokba építésére több megoldást is megvizsgálunk.
3. *Kétirányú illesztők használata az átlátszóság biztosítására.* Az illesztőkkel kapcsolatban az egyik lehetséges gond, hogy nem minden ügyfél számára „átlátszók". Az illesztett objektumok többé nem felelnek meg az Illesztendő felületnek, így nem használhatók bárhol, ahol az Illesztendő objektumok igen. A szükséges átlátszóságot a **kétirányú illesztők** biztosíthatják. Különösen akkor hasznosak, ha két ügyfélnek különbözőképpen kell látnia egy objektumot.
   Vegyünk egy kétirányú illesztőt, amely összekapcsolja a Unidraw grafikus szerkesztő keretrendszert [VL90], illetve a QOCA megszorításfeloldó elemkészletet [HHMV92]. Mindkét rendszer rendelkezik olyan osztályokkal, amelyek kifejezetten ábrázolnak változókat: a Unidraw-ban ilyen a StateVariable (ÁllapotVáltozó), a QOCA-ban pedig a ConstraintVariable (KötésVáltozó). Ahhoz, hogy a Unidraw együttműködhessen a QOCA-val, a ConstraintVariable-t a StateVariable felületéhez kell illeszteni, ahhoz pedig, hogy a QOCA nyújthasson megoldásokat a Unidraw-nak, a StateVariable kell, hogy igazodjon a ConstraintVariable felületéhez.

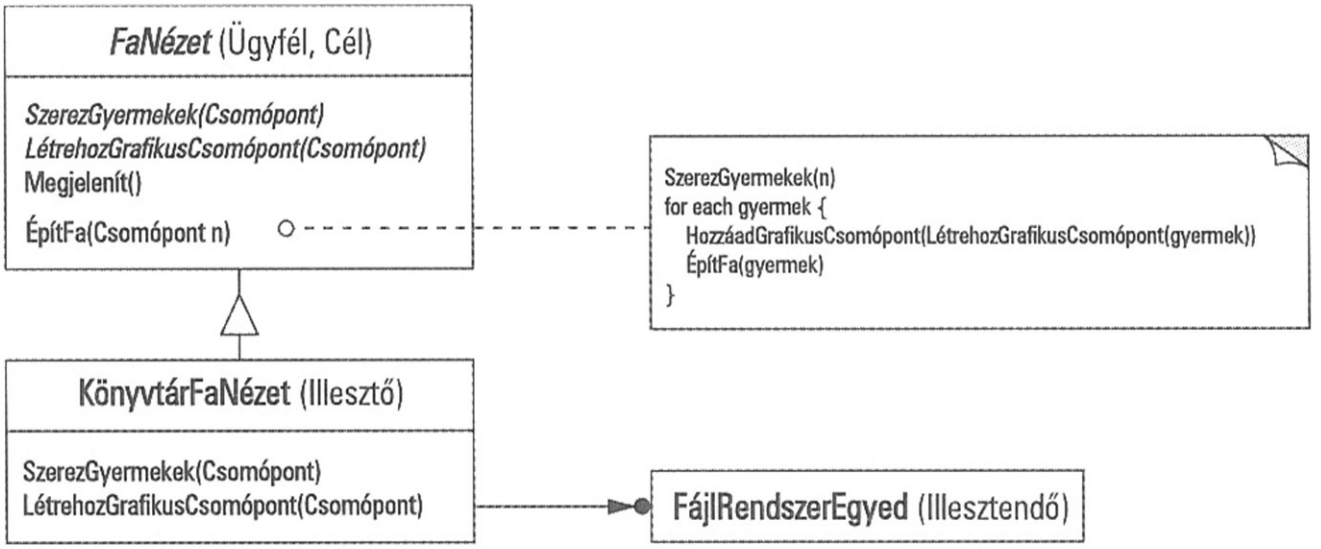


A megoldás kulcsa a ConstraintStateVariable kétirányú osztályillesztő, amely mind a StateVariable, mind a ConstraintVariable alosztálya, feladata pedig a két felület egymáshoz illesztése. A többszörös öröklés ebben az esetben megfelelő megoldás, mert az illesztett osztályok felülete jelentősen különbözik. A kétirányú osztályillesztő mindkét osztálynak megfelel, így mind a két rendszerben működik.

Megvalósítás

Bár az Illesztő minta megvalósítása általában elég egyértelmű, néhány dologra figyelnünk kell:

1. *Az osztályillesztők megvalósítása a C++-ban.* Az osztályillesztők C++ nyelvű megvalósításában az Illesztő nyilvánosan örököl a Céltól, és privát módon az Illesztendőtől, így a Célnak altípusa, de az Illesztendőnek nem.
2. *Csatlakoztatható illesztők.* Tekintsünk át három módszert a korábban bemutatott, hierarchikus szerkezeteket automatikusan megjeleníteni képes FaNézet (TreeDisplay) vezérlő csatlakoztatható illesztőinek megvalósítására.
   Az első lépés, ami közös az itt bemutatott mindhárom megvalósításban, az Illesztendő „keskeny" felületének megtalálása, vagyis az illesztéshez szükséges legkisebb művelethalmaz meghatározása. Egy csak néhány műveletből álló keskeny felület könnyebben illeszthető, mint egy műveletek tucatjait tartalmazó. A FaNézet esetében az illesztendő bármilyen hierarchikus szerkezet lehet. Az elképzelhető legkisebb felület két műveletet tartalmaz: az egyik meghatározza, hogyan jeleníthetünk meg grafikusan egy csomópontot a hierarchiában, a másik pedig megkeresi a csomópont gyermekeit.
   A keskeny felülettel a megvalósításnak három megközelítése lehet:
   (a) *Elvont műveletek használata.* A FaNézet osztályban megfelelő elvont műveleteket határozunk meg a keskeny Illesztendő felület számára. Az alosztályoknak meg kell valósítaniuk ezeket az elvont műveleteket, és illeszteniük kell a hierarchikus szerkezetű objektumot. Egy KönyvtárFaNézet (DirectoryTreeDisplay) nevű alosztály például a könyvtárszerkezet elérésével valósíthatja meg a műveleteket.

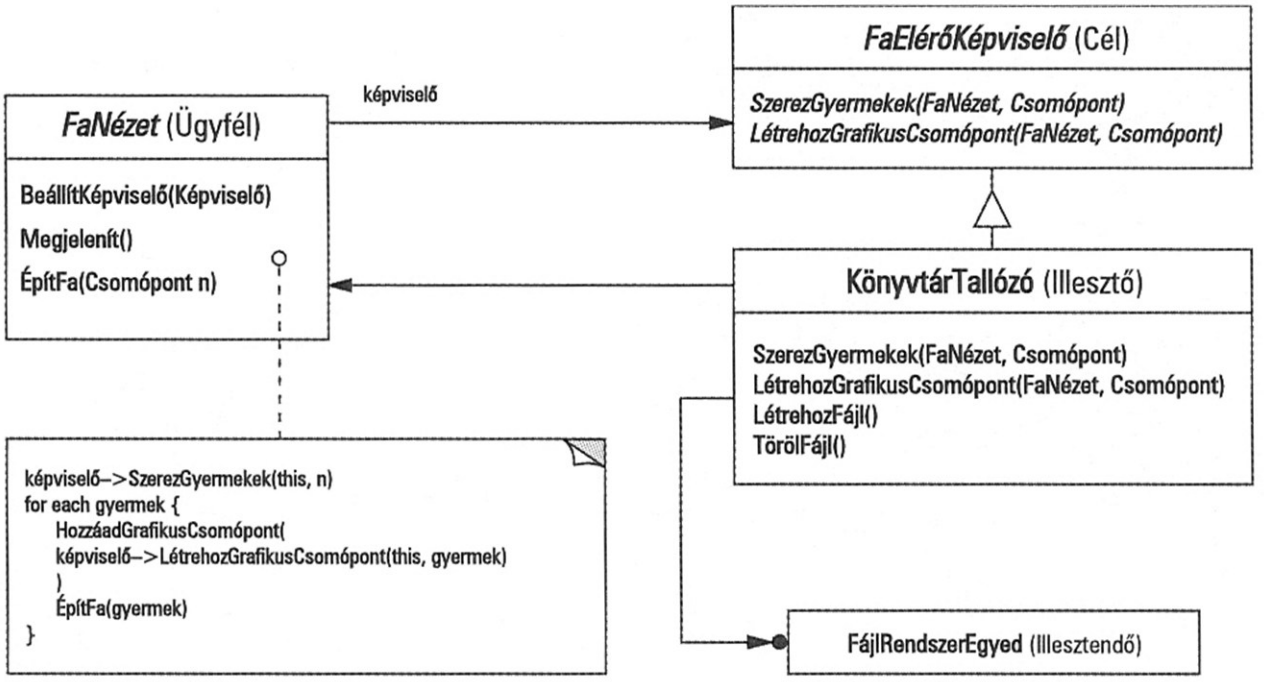


A KönyvtárFaNézet (DirectoryTreeDisplay) arra „szakosítja" a keskeny felületet, hogy FájlRendszerEgyed (FileSytemEntity) objektumokból álló könyvtárszerkezeteket jelenítsen meg.

(b) *Képviselő objektumok használata.* Ennél a megközelítésnél a FaNézet a hierarchikus szerkezet elérésére irányuló kérelmeket egy **képviselő** (delegate) objektumnak továbbítja. A képviselő cseréjével a FaNézet különféle illesztési módszereket használhat.

Tegyük fel például, hogy van egy KönyvtárTallózó (DirectoryBrowser) nevű elemünk, amely a FaNézetet használja. A KönyvtárTallózó kiválóan megfelel képviselőnek, ha a FaNézetnek a hierarchikus könyvtárszerkezethez való illesztéséről van szó. A dinamikus típusokra épülő nyelvekben, amilyen a Smalltalk vagy az Objective C, ez a megközelítés csak egy, a képviselőt az illesztő számára bejegyző felületet igényel. Ezután a FaNézet egyszerűen továbbíthatja a kérelmeket a képviselőnek. A NEXTSTEP [Add94] gyakran él ezzel a megoldással az alosztályok számának csökkentése céljából.

A C++-hoz hasonlóan statikus típusokkal dolgozó nyelvek kifejezett felület-meghatározást követelnek meg a képviselő számára. Egy ilyen felületet úgy határozhatunk meg, hogy a FaNézet igényelte keskeny felületet egy elvont FaElérő-Képviselő (TreeAccessorDelegate) osztályba helyezzük. Ezt a felületet ezután örökléssel a kívánt képviselőbe (esetünkben a KönyvtárTallózóba) „keverhetjük". Amennyiben a KönyvtárTallózónak nincs szülőosztálya, egyszeres öröklést, ha van, többszörös öröklést használunk. Az osztályok ily módon történő összekeverése könnyebb, mint egy új FaNézet alosztály bevezetése és műveleteinek egyenkénti megvalósítása.



(c) *Paraméteres illesztők.* A csatlakoztatható illesztők támogatásának általános módja a Smalltalkban, hogy az illesztőt egy vagy több blokkal paraméterezzük. A blokkszerkezet alosztályok létrehozása nélkül támogatja az illesztést. Az egyes blokkok kérelmeket illeszthetnek, az illesztő pedig minden kérelemhez egy-egy blokkot tárolhat. Példánkban ez azt jelenti, hogy a FaNézet egy blokkot tárol a csomópontok (Node) GrafikusCsomóponttá (GraphicNode) alakításához, egy másikat pedig a csomópont gyermekeinek eléréséhez.

Egy könyvtárszerkezet fanézetének (TreeDisplay) elkészítéséhez például a következő kódot írhatjuk:

```
directoryDisplay :=
        (TreeDisplay on: treeRoot)
                getChildrenBlock:
                        [:node | node getSubdirectories]
                createGraphicNodeBlock:
                        [:node | node createGraphicNode].
```

Ha egy osztályba felületillesztést építünk be, ezzel a megközelítéssel kényelmesen helyettesíthetjük az alosztályok létrehozását.

Példakód

Az alábbiakban rövid vázlatát nyújtjuk a Feladat részben bemutatott osztály- és objektumillesztők megvalósításának. Kezdjük a `Shape` (Alakzat) és `TextView` (SzövegNézet) osztályokkal:

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
```

```
            Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

A `Shape` egy olyan befoglaló dobozt (bounding box) feltételez, amelyet ellentétes sarkai határoznak meg. A `TextView`-t ezzel szemben a kezdőpont (origin), a szélesség (width) és a magasság (height) írja le. A `Shape` ezenkívül meghatároz egy `CreateManipulator` (LétrehozMódosító) nevű műveletet is a `Manipulator` (Módosító) objektumok létrehozására, amelyek tudják, hogyan kell mozgatni az alakzatokat a felhasználó tevékenységének megfelelően.[1] A `TextView` nem rendelkezik hasonló művelettel. A `TextShape` (SzövegAlakzat) osztály ezen különböző felületek illesztője.

Az osztályillesztők többszörös öröklléssel illesztik a felületeket. Működésük kulcsát az jelenti, hogy külön öröklési ágon öröklik a felületet, illetve a megvalósítást. E különbségtétel a C++-ban általában úgy jelenik meg, hogy a felületet nyilvánosan, míg a megvalósítást privát módon örököljük. A `TextShape` illesztőt ennek a hagyománynak megfelelően határozzuk meg.

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
            Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

A `BoundingBox` művelet a `TextView` felületének átalakítását végzi, hogy az megfeleljen a `Shape`-ének.

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;
```

[1] A `CreateManipulator` a Gyártófüggvény példája.

```
    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

Az `IsEmpty` (Üres) művelet az illesztők megvalósításában gyakori megoldást, a kérelmek közvetlen továbbítását mutatja be:

```
bool TextShape::IsEmpty() const {
    return TextView::IsEmpty();
}
```

Végezetül meghatározzuk a `CreateManipulator`-t (amelyet a `TextView` nem támogat); ezt teljes egészében nekünk kell megtennünk. Tegyük fel, hogy már megvalósítottunk egy `TextManipulator` nevű osztályt, ami a `TextShape` objektumok kezelésére szolgál.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Az objektumillesztő objektum-összetétellel illeszti egymáshoz a különböző felületű osztályokat. Ennél a megközelítésnél a `TextShape` illesztő egy mutatóval rendelkezik a `TextView`-ra.

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
            Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

A `TextShape`-nek a `TextView` példányra kell állítania a mutatót; ezt a konstruktorban meg is teszi. Emellett minden esetben, amikor műveleteit hívják, `TextView` objektumára is meg kell hívnia a műveleteket. A példában feltételezzük, hogy az ügyfél létrehozza a `TextView` objektumot, és átadja azt a `TextShape` konstruktorának:

```
TextShape::TextShape (TextView* t) {
    _text = t
}
```

```
void TextShape::BoundingBox (
        Point& bottomLeft, Point& topRight
) const {
        Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

A `CreateManipulator` megvalósítása nem különbözik az osztályváltozatnál látottól, hiszen mi készítettük el a semmiből, a `TextView` szolgáltatásaiból semmit sem használ.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Hasonlítsuk össze a kódot az osztályillesztőével, és meglátjuk, az objektumillesztő megírása ugyan valamivel több munkát igényel, de rugalmasabb. A `TextShape` objektumillesztő változata például a `TextView` alosztályaival is jól működik: az ügyfél egyszerűen átadja a `TextShape` konstruktorának a megfelelő alosztály egy példányát.

Ismert felhasználások

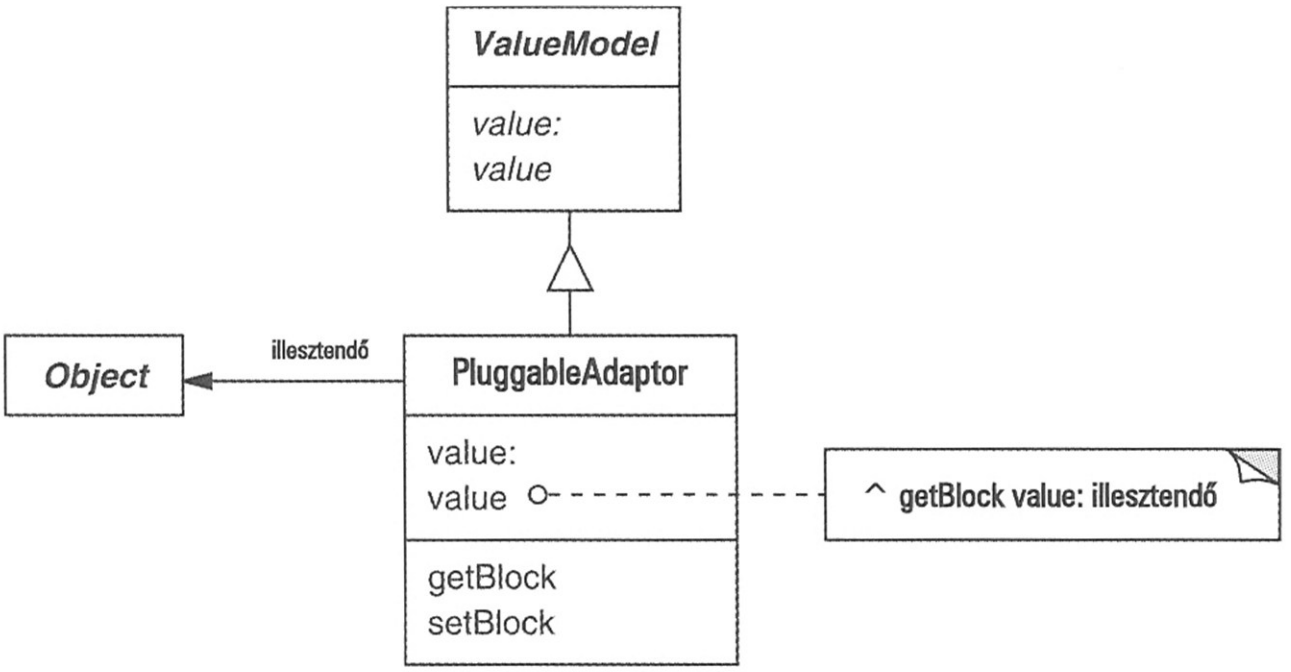
A Feladat részben látott példa az ET++Draw rajzolóprogramból származik, amely az ET++-ra épül [WGM88]. A program az ET++ osztályok szövegszerkesztésre való újrahasznosításához egy TextShape nevű illesztőosztályt használ.

Az InterViews 2.6 egy Interactor nevű elvont osztályt tartalmaz az olyan felhasználói felületi elemekhez, mint a gördítősávok, a gombok, vagy a menük [VL88]. Emellett egy Graphic nevű elvont osztályt is meghatároz az olyan strukturált grafikus objektumokhoz, mint a vonalak, körök, sokszögek és görbék. Mind az Interactor, mind a Graphic rendelkeznek grafikus megjelenéssel, de felületük és megvalósításuk különböző (nem osztoznak közös szülőosztályon), így nem összeegyeztethetők, vagyis egy strukturált grafikus objektum nem ágyazható be közvetlenül mondjuk egy párbeszédablakba.

Ehelyett az InterViews 2.6 egy GraphicBlock (GrafikusBlokk) nevű objektumillesztőt határoz meg, amely az Interactor alosztálya és egy Graphic példányt tartalmaz. A GraphicBlock illeszti a Graphic osztály felületét az Interactoréhoz, és a Graphic példányok ennek segítségével jeleníthetők meg, görgethetők és nagyíthatók egy Interactor szerkezeten belül.

A csatlakoztatható illesztők általánosak az ObjectWorks\Smalltalk-ban [Par90]. A szabványos Smalltalk egy ValueModel (ÉrtékModell) nevű osztályt határoz meg az egyetlen értéket megjelenítő nézetek számára. Az érték elérésére a ValueModel a `value, value:` elvont metódusokból álló felületet biztosítja. Az alkalmazáskészítők erre a célra olyan, az adott tartományra jellemzőbb neveket használhatnak, mint a `width` és a `width:`, de nem szabad, hogy ezeket úgy illesszék a ValueModel felülethez, hogy a ValueModel-ből alosztályokat származtatnak.

Az ObjectWorks\Smalltalk ehelyett a ValueModel PluggableAdaptor (Csatlakoztatható-Illesztő) alosztályát alkalmazza. A PluggableAdaptor objektumok más objektumokat illesztenek a ValueModel felületéhez (`value, value:`), a kívánt értékek beállításához és lekérdezéséhez pedig blokkokkal paraméterezhetők; a PluggableAdaptor belsőleg ezekkel a blokkokkal valósítja meg a `value, value:` felületet. Az alosztály emellett azt is lehetővé teszi, hogy a kényelmesebb nyelvi megfogalmazás érdekében közvetlenül adjuk át a szelektorneveket (pl. `width, width:`). A szelektorok megfelelő blokkokká átalakítása automatikusan történik.



Egy másik példa az ObjectWorks\Smalltalkból a TableAdaptor (TáblázatIllesztő) osztály, amely objektumsorozatokat illeszt egy táblázatos megjelenítéshez. A táblázat soronként egy objektumot jelenít meg. Az ügyfél a TableAdaptornak paraméterként azt az üzenethalmazt adja át, amelynek segítségével a táblázatok lekérdezhetik az oszlopértékeket az objektumoktól.

A NeXT AppKit-jének [Add94] néhány osztálya képviselő objektumokkal hajt végre felületillesztést. Ennek egyik példája az NXBrowser (NXTallózó) osztály, amely adatok hierarchikus listájának megjelenítésére képes. Az NXBrowser egy képviselő objektummal éri el és illeszti az adatokat.

Meyer „kényelmi házassága" (Marriage of Convenience) [Mey88] szintén egyfajta osztályillesztő. Meyer leírja, hogyan illeszti egy FixedStack (RögzítettVerem) nevű osztály egy Array (Tömb) osztály megvalósítását egy Stack (Verem) osztály felületéhez. Az eredmény egy verem, amely rögzített számú elemet tartalmaz.

Kapcsolódó minták

A hidak (lásd: Híd minta) szerkezete hasonló az objektumillesztőkéhez, de céljuk más: arra valók, hogy egy felületet elválasszunk a megvalósításától, hogy azok könnyen és egymástól függetlenül módosíthatók legyenek. Az illesztők ezzel szemben egy *meglevő* objektum felületét változtatják meg.

A Díszítő mintában anélkül bővítünk ki egy másik objektumot, hogy megváltoztatnánk annak felületét. A díszítők így „átlátszóbbak" az alkalmazás számára, mint az illesztők, amiből az is következik, hogy a Díszítő minta támogatja az önhívó összetételt, ami a tiszta illesztőkkel nem lehetséges.

A Helyettes minta képviselőt vagy helyettesítőt biztosít egy másik objektum számára, de nem változtatja meg annak felületét.

Híd

Szerkezeti objektumminta

Cél

Az elvont ábrázolást elválasztani a megvalósítástól, hogy a kettő egymástól függetlenül módosítható legyen.

Egyéb nevek

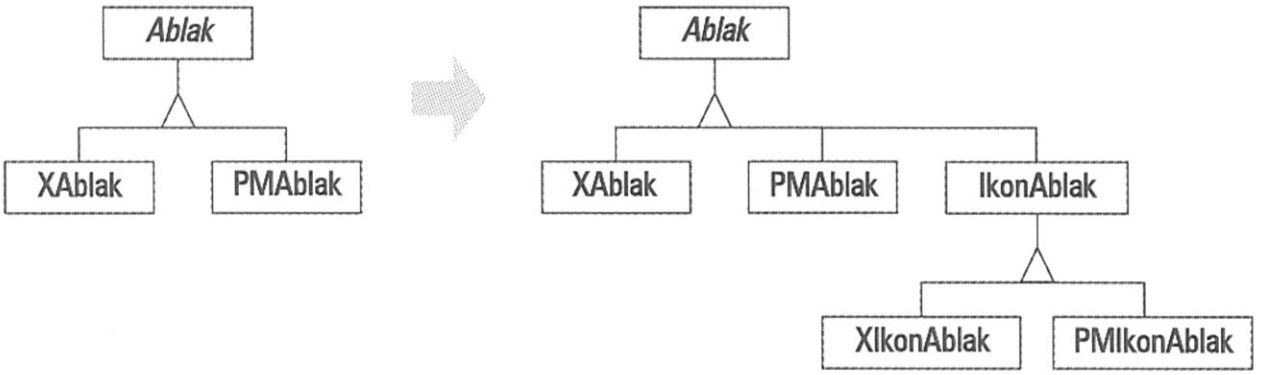
Bridge, Handle/Body (Leíró/Törzs)

Feladat

Amikor egy elvont fogalomhoz több lehetséges megvalósítás közül választhatunk egyet, a megvalósításokat általában örökléssel készítjük el. Egy elvont osztály határozza meg a felületet, amit a konkrét alosztályok különbözőképpen valósítanak meg. Ez a megközelítés azonban nem mindig kellőképpen rugalmas. Az öröklés maradandó kötést hoz létre az elvont fogalom és a megvalósítás között, ami megnehezíti azok egymástól független módosítását, bővítését és újrahasznosítását.

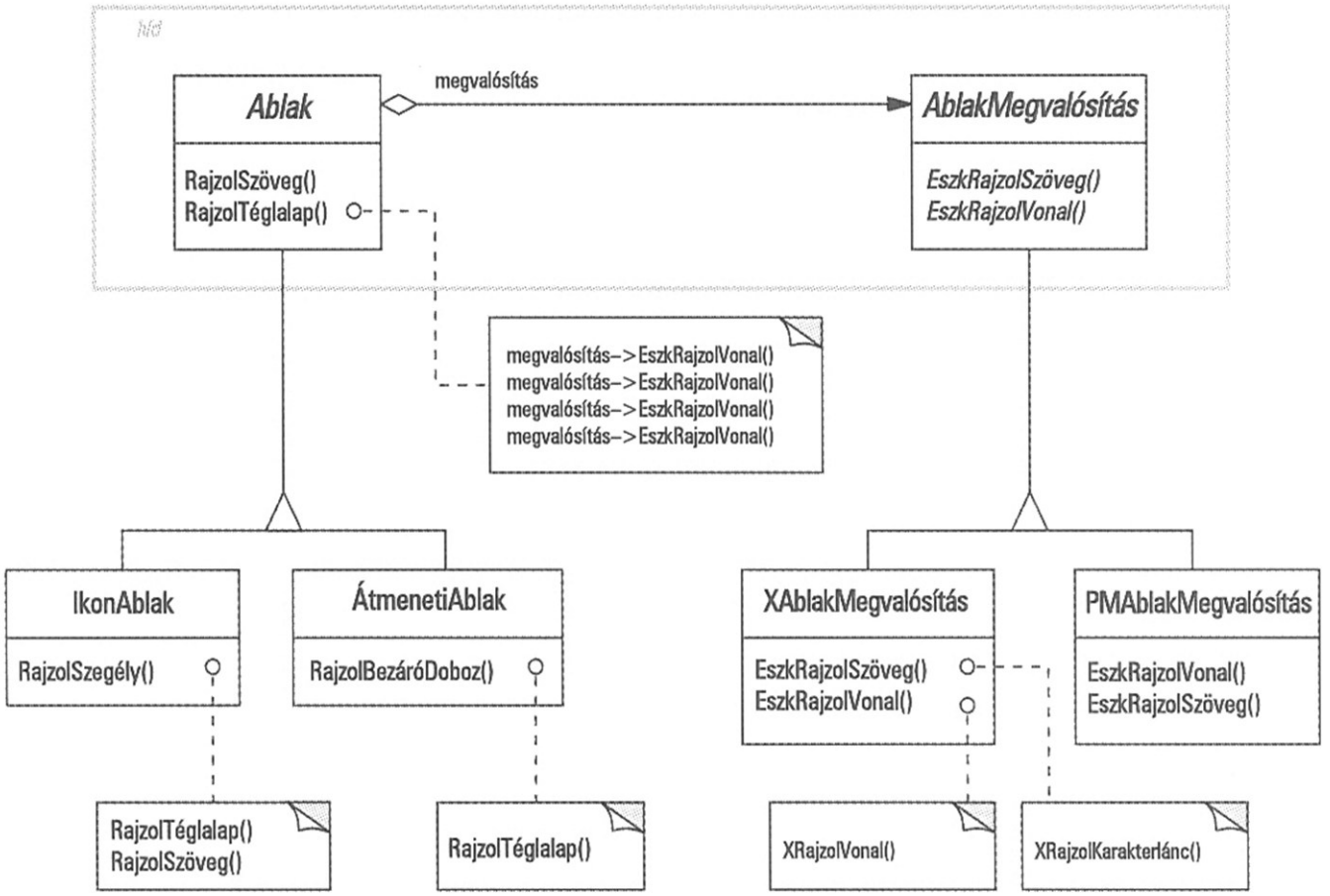
Vegyük például egy rendszerek között hordozható Ablak (Window) megvalósítását egy felhasználói felületi elemkészletben. Az elvont ábrázolásnak lehetővé kell tennie, hogy olyan alkalmazásokat írhassunk, amelyek mind az X Window System, mind az IBM Presentation Manager (PM) felületén működnek. Örökléssel meghatározhatunk egy Ablak nevű elvont osztályt, amelynek XAblak (XWindow) és PMAblak (PMWindow) alosztályai valósítják meg az Ablak felületet a különböző rendszerek számára. Ennek a megoldásnak két hátulütője van:

1. Az elvont Ablak bővítése más ablaktípusok vagy új rendszerek támogatásához kényelmetlen. Képzeljük el az Ablak IkonAblak (IconWindow) nevű alosztályát, amely az Ablak fogalmát ikonokra terjeszti ki. Ahhoz, hogy az IkonAblak támogatása mindkét rendszerre megoldott legyen, *két* új osztályt – XIkonAblak (XIconWindow) és PMIkonAblak (PMIconWindow) – kell megvalósítanunk. Ami még ennél is rosszabb, *minden* ablaktípusra két osztályt kell meghatároznunk, egy harmadik rendszer támogatása pedig újabb Ablak alosztályok készítését igényli a különféle ablaktípusokhoz.



2. Az ügyfél kódját rendszerfüggővé teszi. Amikor az ügyfél létrehoz egy ablakot, egy adott megvalósítással rendelkező konkrét osztályt példányosít. Egy XAblak objektum létrehozásakor például az X Window megvalósításhoz kötjük az Ablak fogalmát, aminek eredményeképp az ügyfél kódja ettől a megvalósítástól fog függni, és nehéz lesz azt más rendszerekre átültetni.
   Az ügyfeleknek képesnek kell lenniük arra, hogy anélkül hozzanak létre egy ablakot, hogy egy bizonyos megvalósításhoz ragaszkodnának. Kizárólag az ablak megvalósításának szabad függnie a rendszertől, amelyen az alkalmazás fut. Mindebből az következik, hogy az ügyfélkódnak az ablakok példányosítását rendszerfüggetlenül kell végeznie.

A Híd tervezési minta úgy oldja meg ezt a problémát, hogy az Ablak fogalmát és annak megvalósítását külön osztályhierarchiákba helyezi. Egy osztályhierarchia ábrázolja az ablakfelületeket (Ablak, IkonAblak, ÁtmenetiAblak), egy másik, önálló hierarchia pedig az egyes rendszerekhez nyújtott ablak-megvalósításokat, amelynek gyökere az AblakMegvalósítás (WindowImp). Az XAblakMegvalósítás (XWindowImp) alosztály például az X Window System alapú megvalósítást biztosítja.



Az Ablak alosztályokon végzett valamennyi műveletet az AblakMegvalósítás felület elvont műveletei alapján valósítjuk meg, így elválasztjuk az elvont ábrázolást a különböző rendszerfüggő megvalósításoktól. Az Ablak és az AblakMegvalósítás közötti kapcsolat a **híd**, amelyen keresztül az elvont fogalom és megvalósítása egymáshoz kapcsolódik, és amely azok egymástól független módosítását lehetővé teszi.

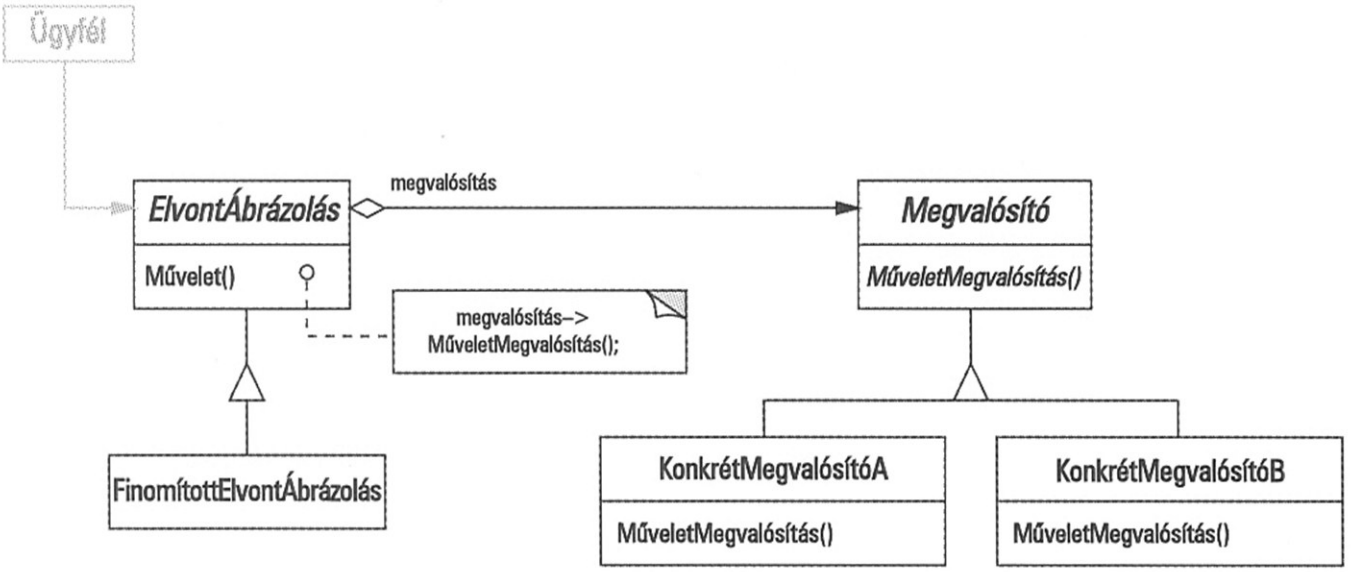
Alkalmazhatóság

A Híd minta használata a következő esetekben célszerű:

- El szeretnénk kerülni az elvont fogalom és megvalósítása közti maradandó kötést. Erre akkor lehet például szükség, ha futásidőben kell választani a megvalósítások közül, vagy átkapcsolni közöttük.
- Mind az elvont ábrázolásnak, mind a megvalósításnak bővíthetőnek kell lennie alosztályok származtatásával. A Híd minta lehetővé teszi a különböző elvont fogalmak és megvalósítások párosítását és egymástól független bővítését.
- Az elvont ábrázolás megvalósításában eszközölt változásoknak nem szabad hatással lenniük az ügyfelekre, vagyis nem igényelhetik azok kódjának újrafordítását.
- (C++) Egy fogalom megvalósítását teljesen el kívánjuk rejteni az ügyfelek elől. A C++-ban az osztályok ábrázolása látható az osztály felületében.

- Túl sok osztály jött létre, mint a Feladat részben található első diagramon láttuk. Az ilyen osztályhierarchia annak szükségességére világít rá, hogy egy adott objektumot két részre kell vágnunk. Rumbaugh a „beágyazott általánosítások" (nested generalizations) kifejezést használja az ilyen hierarchiákra [RBP+91].
- Meg szeretnénk osztani egy megvalósítást több objektum között (mondjuk hivatkozás-számlálással), és ezt el szeretnénk rejteni az ügyfél elől. Ennek egyszerű példája Coplien String osztálya [Cop92], ahol több objektum osztozhat ugyanazon a karakterlánc ábrázoláson (StringRep).

Szerkezet



Résztvevők

- **ElvontÁbrázolás** (Ablak)
  - Meghatározza az elvont fogalom felületét.
  - Egy Megvalósító típusú objektumra hivatkozik.
- **FinomítottElvontÁbrázolás** (IkonAblak)
  - Kibővíti az ElvontÁbrázolás által meghatározott felületet.
- **Megvalósító** (AblakMegvalósítás)
  - Meghatározza a megvalósító osztályok felületét. E felületnek nem kell pontosan megfelelnie az ElvontÁbrázolás felületének, sőt, a kettő teljesen különböző is lehet. A Megvalósító felület általában csak alapműveleteket nyújt, míg az ElvontÁbrázolás magasabb szintű műveleteket határoz meg, amelyek ezeken az alapműveleteken alapulnak.
- **KonkrétMegvalósító** (XAblakMegvalósítás, PMAblakMegvalósítás)
  - Megvalósítja a Megvalósító felületet, és meghatározza annak konkrét megvalósítását.

Együttműködés

- Az ElvontÁbrázolás (Abstraction) továbbítja az ügyfél kérelmeit a hozzá tartozó Megvalósító (Implementor) objektumhoz.

Következmények

A Híd minta előnyei a következők:

1. *A felület és a megvalósítás szétválasztása.* A megvalósítások nem kötődnek maradandóan a felülethez, és futásidőben beállíthatók. Még az is lehetséges, hogy egy objektum futás közben változtassa meg a megvalósítását.
   Az ElvontÁbrázolás és a Megvalósító szétválasztása a fordítási idejű megvalósításfüggőséget is megszünteti. A megvalósító osztály megváltoztatása nem igényli az ElvontÁbrázolás osztály és ügyfelei újrafordítását, ami létfontosságú, ha egy osztálykönyvtár különböző változatai között biztosítanunk kell a bináris megfelelőséget.
   Emellett ez a szétválasztás rétegezésre buzdít, aminek eredménye egy jobban szervezett rendszer lehet. A rendszer magas szintű részének csak az ElvontÁbrázolás és a Megvalósító osztályokat kell ismernie.
2. *Jobb bővíthetőség.* Az ElvontÁbrázolás és Megvalósító hierarchiák egymástól függetlenül bővíthetők.
3. *A megvalósítás részleteinek elrejtése az ügyfelek elől.* Az ügyfeleket elzárhatjuk a megvalósítás olyan részletei elől, mint amilyen a megvalósító objektumok megosztása és az ahhoz kapcsolódó hivatkozás-számlálás (ha van ilyen).

Megvalósítás

A Híd minta megvalósításával kapcsolatban a következő dolgok lényegesek:

1. *Csak egy Megvalósító.* Az olyan helyzetekben, amikor csak egyetlen megvalósítás létezik, az elvont Megvalósító osztály létrehozása nem szükséges. Ez a Híd minta „sovány" változata; az ElvontÁbrázolás és a Megvalósító között ekkor egy az egyhez kapcsolat áll fenn. A szétválasztás azonban akkor is hasznos, ha egy osztály megvalósításának módosítása nem szabad, hogy hatással legyen az osztály meglevő ügyfeleire, vagyis hogy azokat újrafordítani ne kelljen, csak újra beszerkeszteni.
   Carolan [Car89] erre a fajta szétválasztásra a „Cheshire Cat" elnevezést használja. A C++-ban a Megvalósító osztályfelülete egy olyan privát fejléc-állományban határozható meg, amelyet az ügyfelek nem kapnak meg, így az osztályok megvalósítása teljesen elrejthető az ügyfelek elől.
2. *A megfelelő Megvalósító objektum létrehozása.* Ha egynél több van belőlük, hogyan, hol és mikor döntjük el, melyik Megvalósító osztályt kell példányosítani?
   Ha az ElvontÁbrázolás valamennyi KonkrétMegvalósító (ConcreteImplementor) osztályt ismeri, konstruktorában példányosíthatja egyiküket, és a konstruktornak átadott paraméterek alapján választhat közülük. Ha egy gyűjteményosztály például

több megvalósítást is támogat, a döntés meghozható a gyűjtemény mérete alapján. A kisebb gyűjteményekhez használhatunk egy láncolt lista alapú megvalósítást, míg a nagyobbakhoz egy hasító- vagy kivonattáblát (hash).

Egy másik megközelítés, ha kezdetben megadunk egy alapértelmezett megvalósítást, és később igény szerint módosítjuk. Például ha a gyűjtemény növekedése közben túllép egy bizonyos küszöböt, megvalósítását a nagyobb számú elemnek megfelelőre cserélheti.

A döntés teljes egészében át is ruházható egy másik objektumra. Az Ablak–AblakMegvalósítás példában mondjuk bevezethetünk egy gyár objektumot (lásd az Elvont gyár mintát a 3. fejezetben), amelynek egyetlen feladata a rendszerfüggő jellemzők egységbe zárása. A gyár tudja, milyen típusú AblakMegvalósítás objektumot kell létrehozni az éppen használt rendszeren, így amikor egy Ablak ilyet kér tőle, a megfelelőt adja vissza. E megközelítés előnye, hogy az ElvontÁbrázolás egyik Megvalósító osztályhoz sem kötődik közvetlenül.

3. *A megvalósítók megosztása.* Coplien [Cop92] bemutatja, hogy a C++ leíró–törzs (Handle/Body) idiómája hogyan használható a megvalósítások több objektum közötti megosztására. A törzs (Body) egy hivatkozásszámlálót tartalmaz, amelynek értékét a leíró (a Handle osztály) növeli vagy csökkenti. A megosztott törzsű leírók hozzárendelési kódjának általános alakja a következő:

```
Handle& Handle::operator= (const Handle& other) {
        other._body->Ref();
        _body->Unref();

        if (_body->RefCount() == 0) {
                delete _body;
        }
        _body = other._body;

        return *this;
}
```

4. *Többszörös öröklés használata.* A C++-ban egy felületet többszörös örökléssel is összekapcsolhatunk a megvalósításával [Mar91]. Egy osztály például örökölhet nyilvánosan az ElvontÁbrázolás (Abstraction) osztálytól és privát módon egy KonkrétMegvalósítótól. Ez a megoldás azonban statikus öröklésen alapul, ezért a megvalósítást maradandóan a felülethez köti. Ez az oka annak, hogy egy valódi Híd mintát nem valósíthatunk meg többszörös öröklés segítségével – legalábbis a C++-ban nem.

Példakód

Az alább bemutatott C++ kód a Feladat részben bemutatott Ablak–AblakMegvalósítás (Window–WindowImp) példát valósítja meg. A `Window` (Ablak) osztály az ablak fogalmát határozza meg az ügyfélprogramok számára:

```
class Window {
public:
   Window(View* contents);

   //az ablak által kezelt kérelmek
   virtual void DrawContents();

   virtual void Open();
   virtual void Close();
   virtual void Iconify();
   virtual void Deiconify();

   // a megvalósításhoz továbbított kérelmek
   virtual void SetOrigin(const Point& at);
   virtual void SetExtent(const Point& extent);
   virtual void Raise();
   virtual void Lower();

   virtual void DrawLine(const Point&, const Point&);
   virtual void DrawRect(const Point&, const Point&);
   virtual void DrawPolygon(const Point[], int n);
   virtual void DrawText(const char, const Point&);

protected:
   WindowImp* GetWindowImp();
   View* GetView();

private:
   WindowImp* _imp;
   View* _contents;     // az ablak tartalma
};
```

A `Window` egy `WindowImp`-re (AblakMegvalósítás) hivatkozik, amely a háttérben megbúvó ablakkezelő rendszerhez felületet bevezető elvont osztály.

```
class WindowImp {
public:
   virtual void ImpTop() = 0;
   virtual void ImpBottom() = 0;
   virtual void ImpSetExtent(const Point&) = 0;
   virtual void ImpSetOrigin(const Point&) = 0;
```

```
    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // számos további függvény az ablakra rajzoláshoz...
protected:
    WindowImp();
};
```

Az alkalmazás által használható ablakfajtákat – ablakok, ikonok, párbeszédablakok (átmeneti ablakok, TransientWindow), lebegő eszközpaletták stb. – a `Window` alosztályai határozzák meg.

Az `ApplicationWindow` (AlkalmazásAblak) például a `DrawContents` (RajzolTartalom) műveletet valósítja meg, ami a tárolt `View` (Nézet) példányt rajzolja meg:

```
class ApplicationWindow : public Window {
public:
    //...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

Az `IconWindow` (IkonAblak) az általa megjelenítendő ikon bitképének nevét tárolja...

```
class IconWindow : public Window {
public:
    //...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

...és a `DrawContents`-t úgy valósítja meg, hogy az kirajzolja a bitképet az ablakban:

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
            imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

A Window-nak számos további változata lehetséges. Egy TransientWindow-nak (átmeneti ablak, párbeszédablak) működés közben szüksége lehet arra, hogy kapcsolatba lépjen azzal az ablakkal, amelyik létrehozta, ezért hivatkozást tárol rá. A paletták (PaletteWindow) mindig más ablakok felett lebegnek, az ikontárolók (IconDockWindow) pedig IconWindow-kat tárolnak, és szépen elrendezik azokat.

A Window műveletei a WindowImp felületen alapulnak. A DrawRect (RajzolTéglalap) például négy koordinátát számít ki két Point (Pont) paraméteréből, mielőtt meghívná a téglalapot az ablakban kirajzoló WindowImp-műveletet:

```
void Window::DrawRect (const Point& p1, const Point& p2) {
   WindowImp* imp = GetWindowImp();
   imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

A WindowImp konkrét alosztályai különböző ablakkezelő rendszereket támogatnak; az XWindowImp alosztály például az X Window rendszert:

```
class XWindowImp : public WindowImp {
public:
   XWindowImp();

   virtual void DeviceRect(Coord, Coord, Coord, Coord);
   // a nyilvános felület többi része...
private:
   // X Window rendszerre jellemző állapotok, például:
   Display* _dpy;
   Drawable _winid;     // ablakazonosító
   GC _gc;              // ablakkörnyezet
};
```

A Presentation Manager (PM) esetében a PMWindowImp osztályt határozzuk meg:

```
class PMWindowImp : public WindowImp {
public:
   PMWindowImp();
   virtual void DeviceRect(Coord, Coord, Coord, Coord);

   // a nyilvános felület többi része...
private:
   // PM rendszerre jellemző állapotok, például:
   HPS _hps;
};
```

Ezek az alosztályok a `WindowImp` műveleteit az adott ablakkezelő rendszer alapműveleteire építve valósítják meg. A `DeviceRect` (EszközTéglalap) X alapú megvalósítása például a következő:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

A PM megvalósítás formája az alábbi lehet:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left; point[0].y = top;
    point[1].x = right; point[1].y = top;
    point[2].x = right; point[2].y = bottom;
    point[3].x = left; point[3].y = bottom;

    if (
            (GpiBeginPath(_hps, 1L) == false) ||
            (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
            (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
            (GpiEndPath(_hps) == false)
    ) {
            // hibaüzenet

    } else {
            GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Hogyan szerzik meg az ablakok a megfelelő `WindowImp` alosztály példányát? A példában feltesszük, hogy ez a `Window` felelőssége. Az osztály `GetWindowImp` (SzerezAblakMegvalósítás) művelete egy, az ablakkezelő rendszer jellemzőit egységbe záró elvont gyártól (lásd az Elvont gyár mintát a 3. fejezetben) kéri el a megfelelő példányt.

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

A `WindowSystemFactory::Instance()` egy elvont gyárat ad vissza, amely az ablakkezelő rendszerre jellemző objektumokat készíti el. Az egyszerűség kedvéért egykévé tettük (az Egyke mintát lásd az előző fejezetben), a `Window` osztálynak pedig megengedtük, hogy közvetlenül érje el a gyárat.

Ismert felhasználások

Az ablakos példa az ET++-ból [WGM88] származik. Az ET++-ban az AblakMegvalósítást „WindowPort"-nak hívják, amelynek olyan alosztályai vannak, mint az XWindowPort és a SunWindowPort. Az Ablak (Window) objektum a neki megfelelő Megvalósító (Implementor) objektumot úgy hozza létre, hogy egy „WindowSystem" nevű elvont gyártól kéri el. A WindowSystem biztosítja az olyan rendszerfüggő objektumok létrehozásának felületét, mint a betűtípusok, egérmutatók, bitképek és így tovább.

Az ET++ Window–WindowPort megoldása annyiban bővíti a Híd mintát, hogy a WindowPort visszafelé, a Window-ra is hivatkozik. A WindowPort-megvalósító osztály ezt a hivatkozást arra használja, hogy értesítse a Window-t a WindowPort-ra jellemző eseményekről (a felhasználói műveletekről, az ablakok átméretezéséről stb.).

Coplien [Cop92] és Stroustrup [Str91] is megemlíti a leíró (Handle) osztályokat, és példákat is adnak. Ezek a példák a memóriakezelésre, például a karakterláncos ábrázolások megosztására, illetve a változó méretű objektumok támogatására fektetik a hangsúlyt. Mi inkább az elvont fogalom és megvalósítása egymástól független bővíthetőségére összpontosítottunk.

A libg++ könyvtár [Lea88] olyan osztályokat határoz meg, amelyek általános adatszerkezeteket (például Set, LinkedSet, HashSet, LinkedList, HashTable) valósítanak meg. A Set a halmaz fogalmát meghatározó elvont osztály, míg a LinkedList és a HashTable a láncolt listák, illetve a hasító- vagy kivonattáblák konkrét megvalósítói. A LinkedSet és a HashSet olyan Set-megvalósítók, amelyek hidat képeznek az elvont Set és a konkrét LinkedList, illetve HashTable között. Ez a „keskeny" híd példája, mivel nincs benne elvont Megvalósító osztály.

A NeXT AppKit-je [Add94] a grafikus képek megvalósításához és megjelenítéséhez használja a Híd mintát. Egy kép többféleképpen is megjeleníthető; optimális megjelenítése a képernyő tulajdonságaitól függ, különösképpen a színmegjelenítő képességektől, illetve a felbontástól. Az AppKit segítsége nélkül a fejlesztőknek kellene megállapítaniuk, melyik megvalósítás használandó az egyes programokban, különböző körülmények között.

Az AppKit az NXImage–NXImageRep híddal mentesíti ez alól a fejlesztőket. Az NXImage (NXKép) a képek kezelésének felületét írja le, míg a képek megvalósítását az önálló NXImageRep (NXKépÁbr) osztályhierarchia határozza meg, amelyben olyan alosztályokat találunk, mint az NXEPSImageRep, az NXCachedImageRep vagy az NXBitMapImageRep. Az NXImage hivatkozást tart fenn egy vagy több NXImageRep objektumra. Ha egynél több képmegvalósítás létezik, az NXImage kiválasztja az aktuális képernyőnek legjobban megfelelőt. Az NXImage még arra is képes, hogy egy megvalósítást szükség esetén átalakítson egy másikká. E hídváltozat érdekes jellemzője, hogy az NXImage egyidőben több NXImageRep megvalósítást is tárolhat.

Kapcsolódó minták

Hidat létrehozhatunk és beállíthatunk elvont gyárral is.

Az Illesztő minta lényege az egymással kapcsolatban nem álló osztályok együttműködésének biztosítása. Általában olyan rendszerekben használják, amelyek már készen vannak. Ezzel szemben a Híd minta alkalmazása a tervezés elején történik, hogy lehetővé tegyük az elvont fogalmak és a megvalósítások egymástól független változtatását.

Összetétel

Szerkezeti objektumminta

Egyéb nevek

Composite, Kompozit, Kompozíció, Összetett

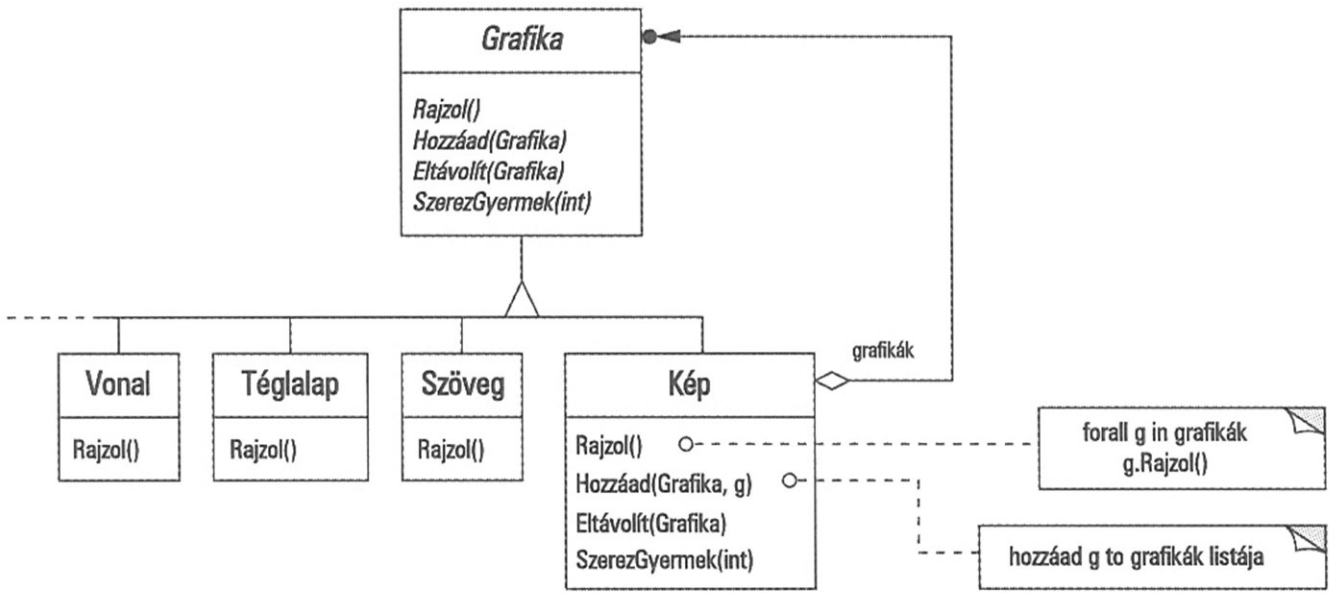
Cél

Objektumokat faszerkezetbe rendezni, hogy ábrázolhassuk a rész–egész viszonyokat. A módszer révén az önálló objektumokat és az objektum-összetételeket egységesen kezelhetjük.

Feladat

Az olyan grafikus alkalmazások, mint a rajzolóprogramok és sémarögzítő rendszerek lehetővé teszik a felhasználóknak, hogy egyszerű alapelemekből bonyolult diagramokat építsenek. Az elemek nagyobb elemekbe csoportosíthatók, amelyekből még nagyobb elemek hozhatók létre. Egy egyszerű megvalósításban ehhez meghatározhatjuk a grafikai alapelemek (primitívek) olyan osztályait, mint a Szöveg (Text) és a Vonal (Line), illetve azon osztályokat, amelyek ezen alapelemek tárolóiként szolgálnak.

Ezzel a megközelítéssel azonban van egy gond: az említett osztályokra épülő kódnak az alapelemeket és a tároló objektumokat különbözőképpen kell kezelnie, még akkor is, ha a felhasználó többnyire azonos módon bánik velük, az objektumok ezen megkülönböztetésének szükségessége pedig bonyolultabbá teszi az alkalmazást. Az Összetétel minta azt írja le, hogyan használhatunk önhívó összetételt (rekurzív kompozíciót), hogy az ügyfélprogramoknak ne kelljen ezzel a megkülönböztetéssel élniük.

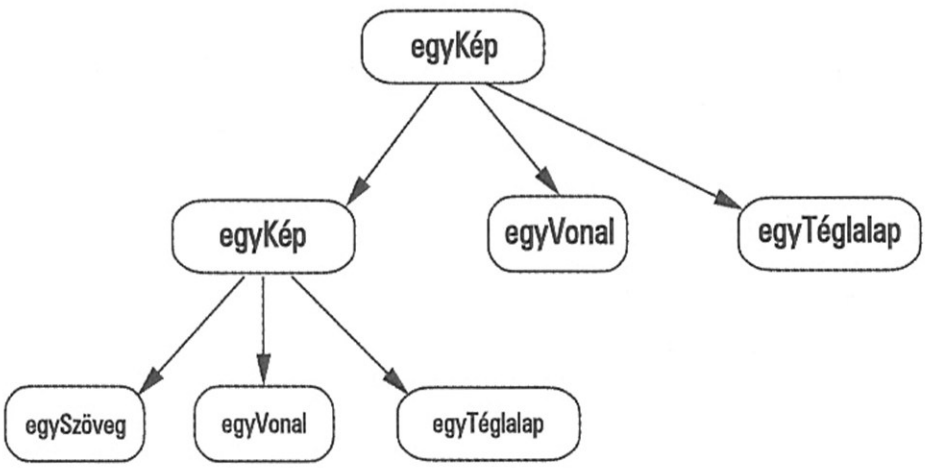


Az Összetétel minta kulcsa egy elvont osztály, amely *egyszerre* ábrázolja az alapelemeket és tárolóikat. Az ábrán látható grafikai rendszerben ez az osztály a Grafika (Graphic). A Grafika a Rajzol-hoz (Draw) hasonló műveleteket vezet be, amelyek az egyes grafikus objektumokra jellemzőek, illetve olyan műveleteket, amelyeket az összetett objektumok közösen használnak, például gyermekeik elérésére és kezelésére.

A Vonal, Téglalap és Szöveg (Line, Rectangle, Text) alosztályok (lásd az előző osztálydiagramot) a grafikus alapobjektumokat határozzák meg, és a Rajzol megvalósításával vonalakat, téglalapokat és szöveget rajzolnak ki. Mivel a grafikus alapelemeknek nincsenek gyermekeik, az említett alosztályok egyike sem valósít meg gyermekekkel kapcsolatos műveleteket.

A Kép (Picture) osztály a Grafika objektumok összetételét (aggregátumát) határozza meg. Ez az osztály úgy valósítja meg a Rajzol műveletet, hogy meghívja azt gyermekeire, és biztosítja hozzá a megfelelő gyermekfüggő műveleteket. A Kép felület illeszkedik a Grafika felülethez, így a Kép objektumokból önhívással újabb Kép objektumok állíthatók elő.

Az alábbi diagram a Grafika objektumokból önhívással felépített objektum-összetételek jellegzetes szerkezetét mutatja:

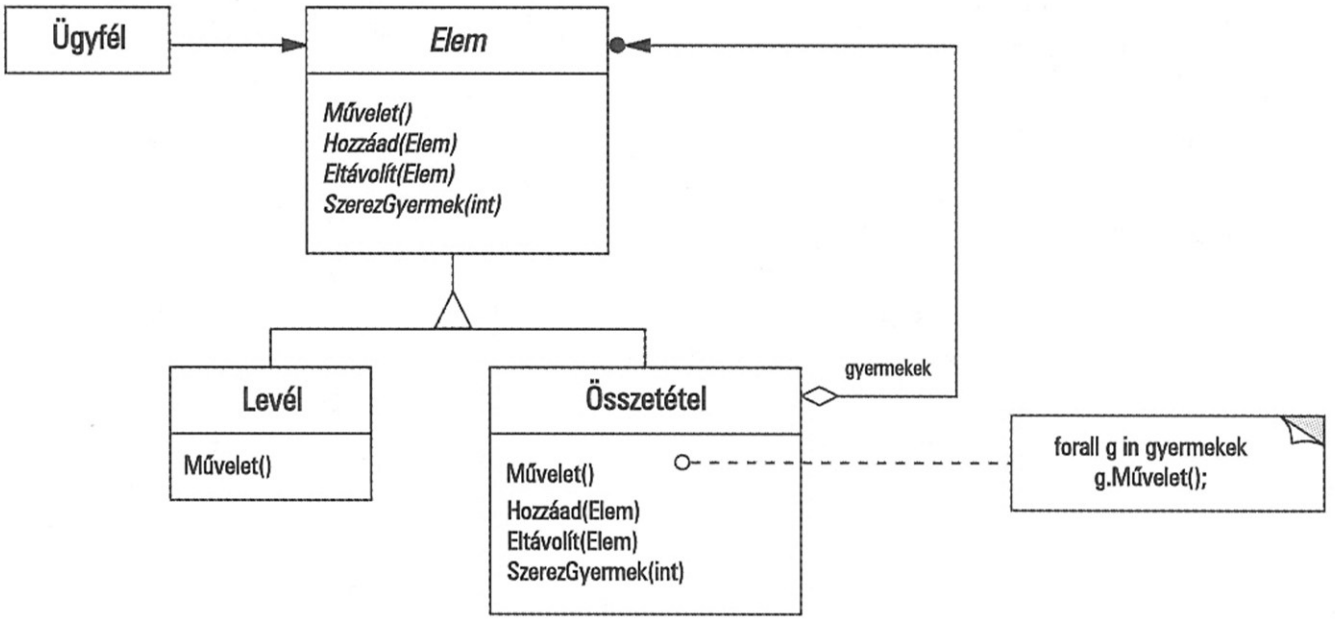


Alkalmazhatóság

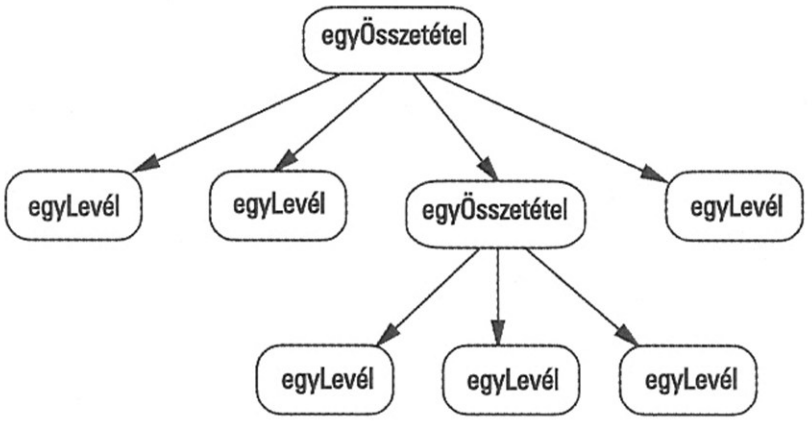
Az Összetétel minta használata a következő esetekben célszerű:

- Objektumok rész–egész viszonyait szeretnénk ábrázolni.
- Azt szeretnénk, hogy az ügyfelek figyelmen kívül hagyhassák az önálló objektumok és az objektum-összetételek közötti különbséget, és az összetett szerkezet minden objektumát egyformán kezelhessék.

Szerkezet



Egy jellegzetes Összetétel objektum szerkezete így festhet:



Résztvevők

- **Elem** (Grafika)
  - Bevezeti az összetétel objektumainak felületét.
  - Megfelelő alapértelmezett viselkedést valósít meg az osztályok közös felülete számára.
  - Felületet vezet be gyermekelemei elérésére és kezelésére.
  - (Nem kötelező) Felületet határoz meg egy elem szülőjének elérésére, és ha szükséges, meg is valósítja.
- **Levél** (Téglalap, Vonal, Szöveg stb.)
  - Az összetétel levélobjektumait képviseli. A levelek olyan objektumok, amelyeknek nincsenek gyermekeik.
  - Meghatározza az összetétel alapobjektumainak viselkedését.

- **Összetétel** (Kép)
  - Meghatározza a gyermekekkel rendelkező elemek viselkedését.
  - Gyermekelemeket tárol.
  - Megvalósítja az Elem (Component) felület gyermekekkel kapcsolatos műveleteit.
- **Ügyfél**
  - Műveleteket végez az összetétel objektumaival az Elem felületen keresztül.

Együttműködés

- Az ügyfelek az Elem osztály felülete segítségével létesítenek kapcsolatot az összetétel objektumaival. Ha a címzett egy Levél (Leaf), a kérelem kezelése közvetlenül történik, ha pedig Összetétel (Composite), akkor a kérelem általában továbbítódik az összetétel gyermekelemeihez. A továbbítás előtt, illetve után további műveletekre is sor kerülhet.

Következmények

Az Összetétel minta előnyei és hátrányai a következők:

1. *Alap- és összetett objektumokból álló osztályhierarchiákat határoz meg.* Az alapobjektumokból bonyolultabb objektumok alkothatók, amelyek maguk is összeépíthetők, és így tovább. Az alapobjektumot váró ügyfélkódnak összetett objektum is átadható.
2. *Egyszerűsíti az ügyfélprogramot.* Az ügyfelek az önálló objektumokat és az összetett szerkezeteket egységesen kezelhetik. Normál esetben nem tudják (és nem is szükséges tudniuk), hogy levél- vagy összetett objektummal van-e dolguk. Ez egyszerűbbé teszi az ügyfél kódját, mert így nem kell esetágakat tartalmazó függvényeket írnunk az összetételt felépítő minden osztályhoz.
3. *Megkönnyíti az új elemek felvételét.* Az új Összetétel és Levél alosztályok automatikusan működnek együtt a már meglevő szerkezetekkel és ügyfélkóddal; az ügyfeleket nem kell módosítani, amikor új Elem osztályokat hozunk létre.
4. *A program túlságosan általános lehet.* Az új elemek hozzáadásának megkönnyítése azzal a hátránnyal jár, hogy nehezebben szabályozhatjuk az összetételek elemeit – egyes esetekben ugyanis arra lehet szükség, hogy egy összetétel csak bizonyos elemeket tartalmazzon. Az Összetétel mintát használva nem támaszkodhatunk a típusrendszerre, így az ilyen megkötéseket annak segítségével nem kényszeríthetjük ki, helyette futásidejű típusellenőrzést kell alkalmaznunk.

Megvalósítás

Az Összetétel minta megvalósításánál a következőkre kell figyelnünk:

1. *Kifejezett szülőhivatkozások.* Ha egy gyermekelemből annak szülőjére hivatkozunk, megkönnyítjük az összetett szerkezet bejárását és kezelését. A szülőhivatkozások egyszerűbbé teszik a feljebb lépést a szerkezetben, illetve az elemek törlését, emellett segítenek a Felelősséglánc minta támogatásában is.
   A szülőhivatkozás meghatározásának helye általában az Elem osztály. A Levél és Összetétel osztályok örökölhetik a hivatkozást, illetve az azt kezelő műveleteket.
   Szülőhivatkozások használata esetén mindenképpen fenn kell tartanunk azt az invariánst (nem változó állítást), hogy az adott összetétel minden gyermekének szülője az az összetett objektum, amely gyermekként tartalmazza őket. Ennek biztosítására a legegyszerűbb mód, ha az elemek szülőjét *kizárólag* akkor változtatjuk meg, amikor egy összetételhez hozzáadjuk vagy onnan eltávolítjuk őket. Ha ennek megvalósítását az Összetétel osztály Hozzáad (Add) és Eltávolít (Remove) műveleteibe helyezzük, az alosztályok valamennyien örökölhetik, az invariáns fenntartása pedig automatikus lesz.
2. *Az elemek megosztása.* Az elemeket gyakran célszerű megosztani, például a tárigény csökkentése végett, de amikor egy elemnek nem lehet egynél több szülője, az ilyen megosztás nem könnyű.
   Egy lehetséges megoldás, ha a gyermekek több szülőt tárolnak, de ez zavart okozhat, ahogy a kérelmek feljebb haladnak a szerkezetben. A Pehelysúlyú mintánál majd látni fogjuk, hogyan tervezhetjük át a programot úgy, hogy egyáltalán ne legyen szükség a szülők tárolására. A megoldás abban az esetben működik, ha a gyermekek elkerülhetik, hogy állapotuk egy részének vagy egészének felfedése nélkül kérelmeket küldjenek a szülőknek.
3. *Az Elem felület lehető legnagyobbra bővítése.* Az Összetétel minta egyik célja, hogy az ügyfelek ne tudjanak róla, melyik Levél vagy Összetétel osztályokat is használják. E cél eléréséhez az Elem osztálynak a lehető legtöbb műveletet kell meghatároznia az Összetétel és Levél osztályokhoz. Az osztály általában alapértelmezett megvalósítást is nyújt ezekhez a műveletekhez, amit a Levél és Összetétel alosztályok felülbírálnak. Mindazonáltal a fenti célkitűzés néha összeütközésbe kerül az osztályhierarchia-tervezés azon alapelvével, miszerint egy osztálynak csak azokat a műveleteket szabad meghatároznia, amelyek értelemmel bírnak alosztályai számára. Az Elem által támogatott műveletek közül azonban számosnak láthatólag nincs értelme a Levél osztályok esetében. Hogyan adhat hát az Elem alapértelmezett megvalósítást hozzájuk?
   Némi kreativitás szükségeltetik: egy művelet, amelynek csak Összetétel osztályok esetében lenne értelme, minden Elemre megvalósítható, ha az Elem osztályba helyezzük. A gyermekek elérésére szolgáló felület például alapvető része az Összetétel osztályoknak, míg a Levél osztályoknak nem feltétlenül, de ha a Levél osztályt olyan Elemnek tekintjük, amelynek *soha* nincsenek gyermekei, az Elem osztályban máris meghatározhatunk egy alapértelmezett gyermekelérő műveletet, amely soha nem

ad vissza gyermekeket. A Levél osztályok ezt az alapértelmezett megvalósítást használhatják, míg az Összetétel osztályok a műveletnek új megvalósítást adva visszaadhatják gyermekeiket.

A gyermekkezelő műveletek több gondot jelentenek, így ezeket külön pontban tárgyaljuk.

4. *A gyermekkezelő műveletek bevezetése.* Bár az Összetétel osztály *megvalósítja* a gyermekek kezelésére szolgáló Hozzáad (Add) és Eltávolít (Remove) műveleteket, az Összetétel minta egyik fontos kérdése, hogy a hierarchia mely osztályai vezetik be (deklarálják) ezen műveleteket. Vezessük be őket az Elem osztályban és értelmezzük a Levél osztályokra, vagy bevezetésük és meghatározásuk csak az Összetétel osztályban és alosztályaiban történjen?

   A döntést az befolyásolja, hogy a biztonság vagy az átlátszóság fontosabb:

   - Ha a gyermekkezelő felületet az osztályhierarchia gyökerében határozzuk meg, az átlátszóság mellett döntünk; ekkor minden elemet egységesen kezelhetünk. Ezzel azonban feláldozzuk a biztonságot, mert az ügyfeleknek lehetőséget teremtünk arra, hogy értelmetlen dolgokkal próbálkozzanak, például objektumokat próbáljanak levelekhez adni vagy eltávolítani onnan.
   - Ha a gyermekkezelést az Összetétel osztályba helyezzük, a program biztonságosabb lesz, mert a C++-hoz hasonló statikus típusokra épülő nyelvekben fordításkor elfoghatjuk az előző pontban említett értelmetlen műveletekre irányuló kísérleteket. Az átlátszóságot azonban elveszítjük, hiszen a levelek és az összetételek felülete különböző lesz.

A tárgyalt tervezési mintában eddig az átlátszóságot részesítettük előnyben a biztonsággal szemben. Ha mégis a biztonságra voksolnánk, előfordulhat, hogy típusinformációt vesztünk, és egy elemet összetétellé kell alakítanunk. Hogyan tehetjük ezt meg anélkül, hogy nem biztonságos típuskényszerítéshez (cast) folyamodnánk?

Az egyik megoldás egy `Összetétel* SzerezÖsszetétel() (Composite* GetComposite())` művelet bevezetése az Elem osztályban. Az Elem biztosít egy alapértelmezett műveletet, amely egy nullmutatót ad vissza. Az Összetétel osztály e művelet felülírásával saját magát adja vissza a `this` mutatón keresztül:

```
class Composite;

class Component {
public:
        //...
        virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
        void Add(Component*);
        //...
        virtual Composite* GetComposite() { return this }
};
```

```
class Leaf : public Component {
        //...
};
```

A `GetComposite` (SzerezÖsszetétel) lehetővé teszi, hogy megkérdezzük az elemet, hogy összetett objektum-e. A visszaadott összetételen biztonságosan végrehajthatók az `Add` és `Remove` műveletek.

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
        test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
        test->Add(new Leaf);// nem ad hozzá levelet
}
```

Egy összetételre hasonló ellenőrzést a C++ `dynamic_cast` szerkezetével végezhetünk. A gond itt természetesen az, hogy az egyes elemeket nem egységesen kezeljük, így ellenőriznünk kell a típust, mielőtt végrehajthatnánk a megfelelő műveletet.

Az átlátszóság biztosítására az egyetlen út, ha az Elemben alapértelmezett `Add` és `Remove` műveleteket határozunk meg, ami viszont újabb gonddal jár:

a `Component::Add` nem valósítható meg anélkül, hogy fenn ne álljon annak a lehetősége, hogy a művelet nem jár sikerrel. Megtehetjük, hogy a műveletnek azt mondjuk, ne csináljon semmit, de ekkor figyelmen kívül hagyjuk azt a tényt, hogy a levélhez való hozzáadás kísérlete valószínűleg hibát jelez, mely esetben az `Add` művelet szemetet „állít elő". Esetleg arra utasíthatjuk, hogy törölje argumentumát, de az ügyfelek valószínűleg nem számítanak erre.

Ha az adott elemnek nem lehet gyermeke, vagy ha a `Remove` argumentuma nem az elem gyermeke, általában jobb, ha az `Add` és a `Remove` alapértelmezés szerint nem járnak sikerrel (például kivételt váltanak ki).

Egy másik lehetőség, hogy némileg módosítjuk az „eltávolítás" (remove) jelentését. Ha az elem hivatkozik a szülőjére, a `Component::Remove` műveletet felülírhatjuk úgy, hogy az elem eltávolítsa magát a szülőjéből, de a megfelelő `Add` műveletet ekkor sem tehetjük értelmessé.

5. *Tartalmazza-e az Elem az elemek listáját?* Csábíthat a lehetőség, hogy az Elem osztályban, ahol a gyermekelérő és -kezelő műveleteket bevezetjük, példányváltozóként meghatározzuk a gyermekek halmazát. A gyermekmutatónak az alaposztályba

helyezése azonban külön tárigényt jelent minden levél esetében, még akkor is, ha a leveleknek soha nincsenek gyermekeik. A megoldás csak akkor kifizetődő, ha a szerkezetben viszonylag kevés gyermek található.

6. *A gyermekek sorrendje.* Az összetételek gyermekeinek sorrendjét gyakran meghatározzák. A korábbi Grafika példában a sorrend az előtér–háttér sorrendet tükrözheti. Ha az összetételek elemzőfákat képviselnek, az összeállító utasítások egy olyan Összetétel példányai lehetnek, amelynek gyermekeit sorrendbe kell állítani, hogy tükrözzék a program felépítését.
   Ha a gyermekek sorrendje fontos, a gyermekelérő és -kezelő felületeket gondosan kell megterveznünk, hogy megfelelően kezelhessük a gyermekek sorozatát. Ebben a Bejáró tervezési minta segíthet.
7. *Átmeneti tár (cache) használata a teljesítmény javítására.* Ha gyakran kell bejárást vagy keresést végeznünk az összetételekben, a gyermekekkel kapcsolatos bejárási és keresési információkat az Összetétel osztály átmenetileg tárolhatja. A tárban elhelyezhetjük az aktuális eredményeket, de szorítkozhatunk azokra az információkra is, amelyek a bejárás vagy keresés lerövidítését szolgálják. A Feladat részben bemutatott Kép (Picture) osztály például tárolhatja gyermekei befoglaló dobozát, így rajzolás vagy kijelölés közben elkerülheti, hogy műveleteket végezzen azokon a gyermekeken, amelyek éppen nem láthatók az ablakban.
   Ha egy elem megváltozik, szülőinek átmeneti tárát érvényteleníteni kell. Ez akkor a legkönnyebb, amikor az elemek ismerik szülőiket. Tehát ha átmeneti tárolást alkalmazunk, meg kell határoznunk egy felületet, amelynek segítségével tájékoztathatjuk az összetételeket, hogy táruk érvénytelen.
8. *Ki törölje az elemeket?* A szemétgyűjtés nélküli nyelvekben általában az a legjobb, ha az Összetételek felelnek gyermekeik törléséért, amikor az összetétel megsemmisül. E szabály alól kivételt jelent, amikor a levélobjektumok nem módosulók (immutable), és így megoszthatók.
9. *Melyik a legjobb adatszerkezet az elemek tárolására?* Az összetételek különféle adatszerkezeteket használhatnak gyermekeik tárolására: láncolt listákat, fákat, tömböket és kivonattáblákat (hasítótábla, hash). A választott adatszerkezet (mint mindig) attól kell, hogy függjön, melyik a hatékonyabb. Valójában még az sem szükséges, hogy általános célú adatszerkezetet használjunk. Az összetételek időnként külön változókat tartanak fenn az egyes gyermekek számára, pedig ez azt igényli, hogy az Összetétel minden alosztálya saját kezelőfelületet valósítson meg. Példát az Értelmező mintánál láthatunk, a következő fejezetben.

Példakód

Az olyan részegységekből álló eszközöket, mint a számítógép vagy egy hi-fi berendezés, gyakran rész–egész vagy „tartalmazza" viszonyok alapján ábrázoljuk. A számítógépház például a meghajtókat és az alaplapot tartalmazza, a busz a kártyákat, és így tovább. Az ilyen felépítés természetes módon modellezhető az Összetétel minta segítségével.

Az Equipment (Eszközök) osztály határozza meg a rész–egész hierarchia összes részegységére érvényes felületet:

```
class Equipment {
public:
   virtual ~Equipment();

   const char* Name() { return _name; }

   virtual Watt Power();
   virtual Currency NetPrice();
   virtual Currency DiscountPrice();

   virtual void Add(Equipment*);
   virtual void Remove(Equipment*);
   virtual Iterator<Equipment*>* CreateIterator();
protected:
   Equipment(const char*);
private:
   const char* _name;
};
```

Az Equipment olyan műveleteket vezet be, amelyek a részegységek tulajdonságait adják vissza, például fogyasztásukat és árukat. Az alosztályok ezeket a műveleteket adott részegység-típusokra valósítják meg. Az Equipment emellett bevezeti a CreateIterator (LétrehozBejáró) műveletet is, amely egy, az elemek elérésére szolgáló bejárót (Iterator, lásd a C függeléket) ad vissza. E művelet alapértelmezett megvalósítása egy NullIterator-t ad vissza, amely az üres halmaz bejárására szolgál.

Az Equipment alosztályai olyan levélosztályokat tartalmazhatnak, amelyek a lemezmeghajtókat, integrált áramköröket és kapcsolókat jelképezik:

```
class FloppyDisk : public Equipment {
public:
   FloppyDisk(const char*);
   virtual ~FloppyDisk();

   virtual Watt Power();
   virtual Currency NetPrice();
   virtual Currency DiscountPrice();
};
```

A más részegységeket tartalmazó részegységek alaposztálya a CompositeEquipment (ÖsszetettEszközök), amely egyben az Equipment alosztálya:

```
class CompositeEquipment : public Equipment {
public:
   virtual ~CompositeEquipment();

   virtual Watt Power();
   virtual Currency NetPrice();
   virtual Currency DiscountPrice();

   virtual void Add(Equipment*);
   virtual void Remove(Equipment*);
   virtual Iterator<Equipment*>* CreateIterator();

protected:
   CompositeEquipment(const char*);
private:
   List<Equipment*> _equipment;
};
```

A `CompositeEquipment` az alegységek elérésére és kezelésére szolgáló műveleteket határozza meg. Az `Add` (Hozzáad) művelet alegységeket szúr be az alegységek `_equipment` tagban tárolt listájába, míg a `Remove` (Eltávolít) törli azokat onnan. A `CreateIterator` művelet egy bejárót ad vissza (mégpedig egy `ListIterator`-példányt), amely majd bejárja a listát.

A `NetPrice` (NettóÁr) alapértelmezett megvalósítása a `CreateIterator` segítségével összegzi az alegységek nettó árát[2]:

```
Currency CompositeEquipment::NetPrice () {
   Iterator<Equipment*>* i = CreateIterator();
   Currency total = 0;

   for (i->First(); !i->IsDone(); i->Next()) {
         total += i->CurrentItem()->NetPrice();
   }
   delete i;
   return total;
}
```

Most már ábrázolhatunk egy számítógép-házat, a `CompositeEquipment` osztály `Chassis` (Ház) nevű alosztályaként. A `Chassis` a gyermekekkel kapcsolatos műveleteket örökli a `CompositeEquipment`-től.

```
class Chassis : public CompositeEquipment {
public:
   Chassis(const char*)
```

[2] A bejáró törléséről könnyű megfeledkezni, amikor már végeztünk a használatával. A Bejáró mintánál megmutatjuk, hogyan védekezhetünk az ilyen hibák ellen.

```
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

A hasonló tárolókat ugyanígy határozhatjuk meg, és máris felépíthetjük (elég egyszerű) személyi számítógépünket:

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "A nettó ár: " << chassis->NetPrice() << endl;
```

Ismert felhasználások

Az Összetétel minta alkalmazására szinte minden objektumközpontú rendszerben találhatunk példát. A Smalltalk Model/View/Controller [KP88] eredeti View osztálya is összetétel volt, és szinte valamennyi felhasználói felületi elemkészlet és keretrendszer követte a példáját, köztük az ET++ a VObjects-szel [WGM88], az InterViews a Styles-szal [LCI+92], a Graphics [VL88] és a Glyphs [CL90]. Az érdekesség kedvéért megemlítendő, hogy az eredeti View-hoz alnézetek halmaza tartozott, vagyis a View egyszerre töltötte be az Elem és az Összetétel osztály szerepét. A Smalltalk-80 4.0-ás kiadásában a Model/View/Controller rendszert átdolgozták; itt már egy VisualComponent (VizuálisElem) nevű osztályt találunk, View és CompositeView (ÖsszetettNézet) alosztályokkal.

Az RTL Smalltalk-fordítói keretrendszer [JML92] kiterjedten használja az Összetétel mintát. Az RTLExpression (RTLKifejezés) elemzőfák Component (Elem) osztálya, amelynek alosztályai, például a BinaryExpression (BinárisKifejezés), gyermek RTLExpression objektumokat tartalmaznak. Ezek az osztályok építik fel az elemzőfák összetett szerkezetét. A programok köztes Single Static Assignment (SSA) formájának Elem osztálya a RegisterTransfer, melynek levél-alosztályai különböző statikus hozzárendeléseket határoznak meg:

- alapvető hozzárendeléseket, melyek során két regiszteren hajtanak végre egy műveletet, majd az eredményt egy harmadikhoz rendelik;
- a forrásregiszteren kívül célregiszterrel nem rendelkező hozzárendelést, ami azt jelzi, hogy a regisztert egy függvény visszatérése után használjuk;

- forrással nem, csak célregiszterrel rendelkező hozzárendelést, ami azt jelzi, hogy a regiszter használatára a függvény elindulása előtt kerül sor.

A RegisterTransferSet alosztály az egyszerre több regisztert módosító hozzárendelések Összetétel osztálya.

A minta pénzügyi területen való alkalmazására példa, amikor egy portfolió pénzügyi eszközöket összesít. Ha a portfoliót olyan összetételként valósítjuk meg, amely megfelel az egyes eszközök felületének, bonyolult összesítéseket végezhetünk [BE93].

A Parancs minta leírja, hogyan építhetők össze és állíthatók sorba a Parancs objektumok egy MakróParancs (MacroCommand) Összetétel osztály segítségével.

Kapcsolódó minták

Az elem–szülő hivatkozásokat gyakran alkalmazzák felelősségláncok építésére.

A Díszítő minta gyakran használatos együtt az Összetétel mintával. Ilyenkor a díszítők és összetételek szülőosztálya általában közös, ezért a díszítőknek olyan műveletekkel kell támogatniuk az Elem felületet, mint a Hozzáad (Add), az Eltávolít (Remove) vagy a SzerezGyermek (GetChild).

A Pehelysúlyú minta révén megoszthatjuk az elemeket, de ekkor azok nem hivatkozhatnak többé szülőjükre.

Az összetételek bejárására a Bejáró minta alkalmazható.

A Látogatók olyan viselkedést és műveleteket gyűjthetnek egy helyre, amelyek másképp megoszlanának az Összetétel és Levél osztályok között.

Díszítő

Szerkezeti objektumminta

Cél

Az objektumokhoz dinamikusan további felelősségi köröket rendelni. A kiegészítő szolgáltatások biztosítása terén e módszer rugalmas alternatívája az alosztályok létrehozásának.

Egyéb nevek

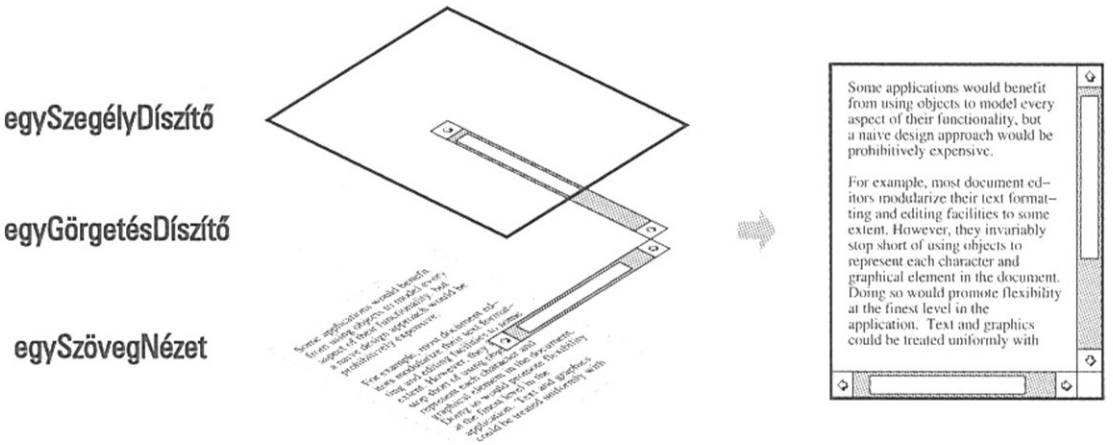
Decorator, Wrapper (Burkoló)

Feladat

Időnként egyes objektumokhoz, nem pedig egy teljes osztályhoz szeretnénk felelősségeket rendelni. Egy grafikus felhasználói felületi elemkészlet például lehetővé kell tegye olyan tulajdonságok, illetve viselkedések felvételét bármely felületelemhez, mint amilyenek a szegélyek vagy a görgetés.

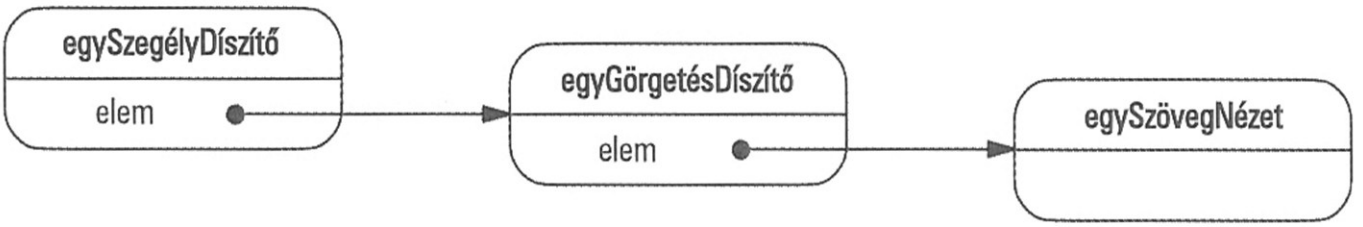
A felelősségek hozzáadására az egyik mód az öröklés. Ha egy szegélyt például egy adott osztálytól öröklünk, annak valamennyi alosztály-példánya körül megjelenik a szegély. Ez a megoldás azonban rugalmatlan, mert azt, hogy van-e szegély, statikusan döntjük el, az ügyfelek nem szólhatnak bele, mikor és hogyan díszítjük az elemet szegéllyel.

Ennél rugalmasabb megoldás, ha az elemet egy másik objektumba ágyazzuk, amely gondoskodik a szegély hozzáadásáról. A beágyazó objektumot nevezzük **díszítőnek**. A díszítő a díszített elem felületéhez igazodik, így jelenléte észrevétlen marad az elem ügyfelei számára. A díszítő továbbítja a kérelmeket az elemhez, és a továbbítás előtt vagy után egyéb műveleteket is végrehajthat (például szegélyt rajzolhat). Az átlátszóság révén több díszítő is egymásba ágyazható, így korlátlan számú tulajdonsággal vagy tevékenységgel egészíthetjük ki a rendszert.

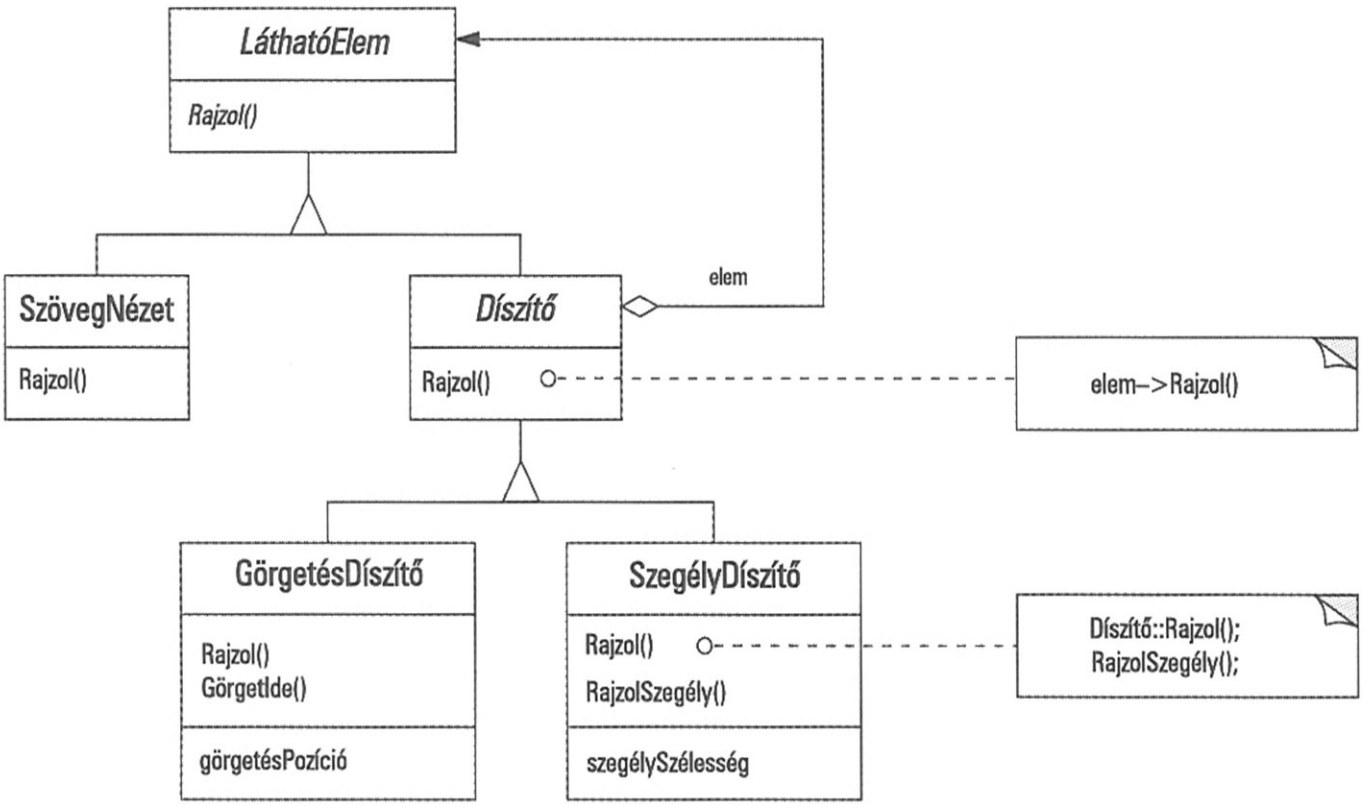


Példaként tegyük fel, hogy van egy SzövegNézet (TextView) objektumunk, amely szöveget jelenít meg egy ablakban. A SzövegNézet alapállapotban nem rendelkezik gördítősávval, hiszen arra nem mindig van szükség. Amikor kell, hozzáadásáról egy GörgetésDíszítő (ScrollDecorator) objektummal gondoskodunk. Tegyük fel, hogy emellett vastag fekete szegélyt is szeretnénk rajzolni a SzövegNézet köré – ezt egy SzegélyDíszítő (BorderDecorator) végezheti. A díszítőket egyszerűen összeépítjük a SzövegNézet objektummal, és máris elérjük a kívánt eredményt.

Az alábbi objektumdiagram azt mutatja, hogyan építhető össze a SzövegNézet objektum a SzegélyDíszítő, illetve GörgetésDíszítő objektumokkal, hogy egy szegéllyel ellátott, görgethető szövegnézőkét hozhassunk létre:



A GörgetésDíszítő és a SzegélyDíszítő a Díszítő (Decorator) alosztályai, amely a más látható elemeket díszítő látható elemek elvont osztálya.



A LáthatóElem (VisualComponent) a látható elemek elvont osztálya; azok rajzoló és eseménykezelő felületét határozza meg. Megfigyelhetjük, hogy a Díszítő (Decorator) osztály egyszerűen továbbítja a rajzolási kérelmeket a hozzá tartozó elemhez, alosztályai pedig kibővítik ezt a műveletet.

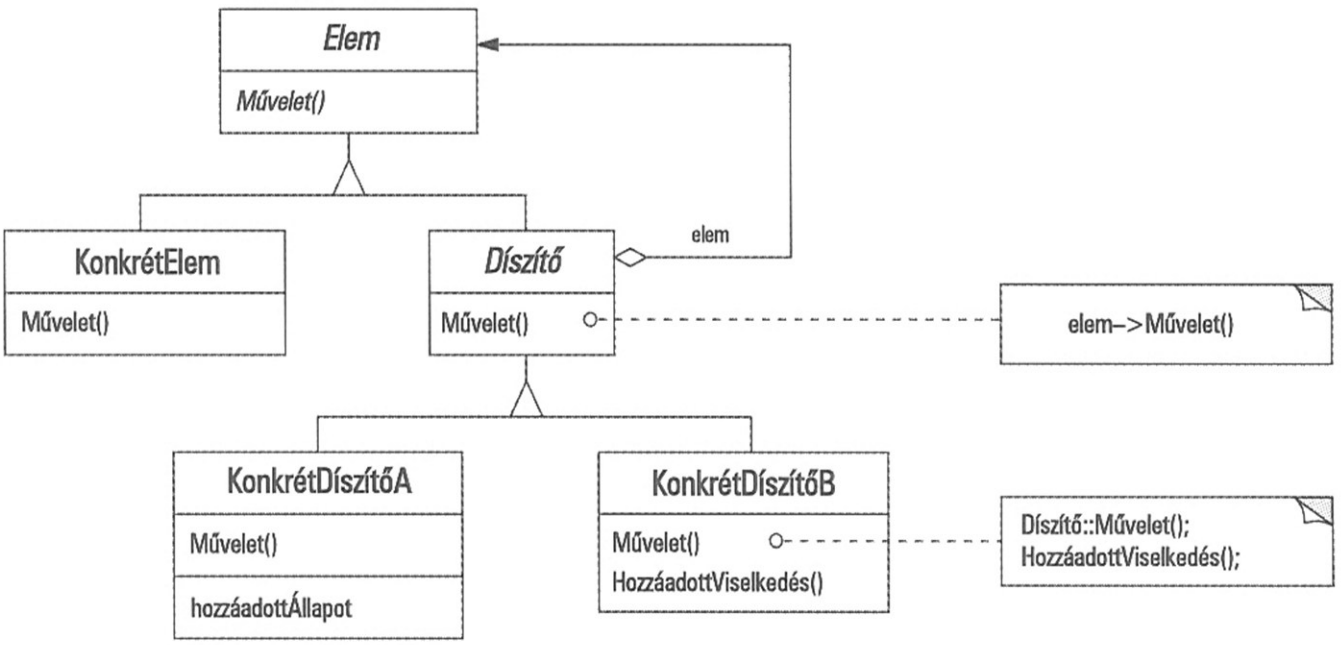
A Díszítő alosztályok szabadon biztosíthatnak további műveleteket. A GörgetésDíszítő (ScrollDecorator) GörgetésIde (ScrollTo) művelete például lehetővé teszi más objektumok számára, hogy görgessék a felületet, *ha* tudják, hogy a felület GörgetésDíszítő objektumot tartalmaz. A tervezési minta lényeges vonása, hogy díszítők bárhol alkalmazhatók, ahol egy LáthatóElem megjelenhet. Emiatt az ügyfelek általában nem tudnak különbséget tenni egy díszített és egy díszítetlen elem között, így teljesen függetlenek maradhatnak a díszítéstől.

Alkalmazhatóság

A Díszítő minta az alábbi esetekben alkalmazható:

- Egyes objektumokat dinamikusan és átlátszóan, vagyis más objektumokat nem érintve bizonyos felelősségi körökkel szeretnénk kiegészíteni.
- Eltávolítható jellemzőket szeretnénk felvenni.
- Az alosztályokkal való bővítés nem célszerű. Előfordulhat, hogy nagy számú önálló bővítés lehetséges, de az egyes kombinációk támogatása az alosztályok számának robbanásszerű növekedéséhez vezetne. Az is lehetséges, hogy a kívánt osztály-meghatározás rejtett vagy más módon hozzáférhetetlen alosztály-létrehozás céljára.

Szerkezet



Résztvevők

- **Elem** (LáthatóElem)
  - Meghatározza azon objektumok felületét, amelyek különféle felelősségekkel dinamikusan bővíthetők.
- **KonkrétElem** (SzövegNézet)
  - Egy objektumot határoz meg, amelyhez kiegészítő felelősségek csatolhatók.
- **Díszítő**
  - Egy Elem (Component) objektumra hivatkozik, és olyan felületet határoz meg, amely megfelel az Elem felületének.
- **KonkrétDíszítő** (SzegélyDíszítő, GörgetésDíszítő)
  - Felelősségeket rendel az elemhez.

Együttműködés

- A Díszítő (Decorator) kérelmeket továbbít Elem (Component) objektumához, ezen kívül a kérelem továbbítása előtt vagy után egyéb műveleteket is végrehajthat.

Következmények

A Díszítő mintának legalább két lényeges előnye és két hátránya van:

1. *A statikus öröklésnél rugalmasabb.* A Díszítő minta rugalmasabb módot nyújt az objektumok felelősségi körökkel való bővítésére, mint a statikus (többszörös) öröklés; az új képességek hozzáadása és eltávolítása futásidőben egyszerűen, pusztán a díszítők csatolásával és leválasztásával lehetséges. Ezzel szemben az öröklés új osztályok (pl. SzegélyesGörgethetőSzövegNézet, SzegélyesSzövegNézet) létrehozását igényli minden új képességhez, ami rengeteg osztályhoz, s így bonyolultabb rendszerhez vezet. Ezen kívül, ha egy adott Elem osztályhoz különböző Díszítő osztályokat biztosítunk, a felelősségi köröket egymáshoz illeszthetjük és keverhetjük is.
   A díszítők azt is egyszerűbbé teszik, hogy egy tulajdonságot kétszer vegyünk fel. Ha egy SzövegNézethez például kettős szegélyt szeretnénk adni, csak két SzegélyDíszítőt (BorderDecorator) kell hozzácsatolnunk. A Szegély (Border) osztályból való kétszeres öröklés ezzel szemben legalábbis könnyebben hibát eredményezhet.
2. *Elkerülhető a képességeket tartalmazó osztályok túl magasra helyezése a hierarchiában.* A Díszítő minta használatakor a felelősségek hozzáadásának költségeit fokozatosan fizetjük meg. Nem kell azzal próbálkoznunk, hogy egy bonyolult, testreszabható osztályban támogassunk minden elképzelhető képességet; helyette elég egy egyszerű osztályt meghatároznunk, és díszítő objektumokkal fokozatosan hozzáadni a kívánt szolgáltatásokat. A szolgáltatások így egyszerű elemekből építhetők fel, az alkalmazásnak pedig nem kell fizetnie olyan képességekért, amelyeket nem használ. Emellett egyszerű a bővítendő objektumok osztályaitól függetlenül új fajta díszítőket meghatározni, még előre nem látható bővítések esetében is. Ezzel szemben egy bonyolult osztály bővítése a hozzáadott képességekkel kapcsolatban nem álló részleteket fedne fel.

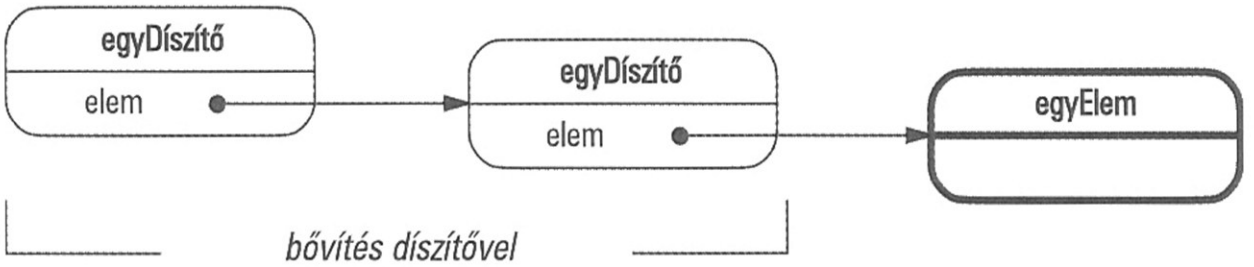
3. *A díszítő és a hozzá tartozó elem nem azonos.* A díszítők átlátszó egységként viselkednek, de az objektumok azonossága szempontjából egy díszített objektum nem azonos az eredeti objektummal, így nem is építhetünk erre, amikor díszítőket alkalmazunk.
4. *Számos kisméretű objektum jön létre.* A Díszítő minta használata gyakran olyan rendszert eredményez, amely számos hasonló kinézetű apró objektumból áll. Ezen objektumok csak kapcsolódásuk módjában különböznek, osztályukat és a változóikban tárolt értékeket tekintve nem. Bár egy ilyen rendszer a szakértő számára könnyen testreszabható, nehéz átlátni, és a hibakeresést is megnehezíti.

Megvalósítás

A Díszítő minta megvalósítása során az alábbiakra kell tekintettel lennünk:

1. *A felület megfelelősége.* A díszítő objektum felületének illeszkednie kell az általa díszített elem felületéhez. A KonkrétDíszítő (ConcreteDecorator) osztályoknak ezért egy közös osztályból kell öröklődniük (legalábbis a C++-ban).
2. *Az elvont Díszítő osztály kihagyása.* Ha csak egy képességet szeretnénk felvenni, nincs szükség elvont Díszítő osztály meghatározására. Ha nem új osztályhierarchiát építünk, hanem egy meglevővel dolgozunk, gyakran ez a helyzet. Ebben az esetben a Díszítő azon szolgáltatását, hogy kérelmeket továbbít az elemhez, a KonkrétDíszítő osztályba helyezhetjük.
3. *Az Elem osztályok pehelysúlyúvá tétele.* A felület megfelelőségének biztosítása érdekében az elemeket és díszítőiket egy közös Elem osztályból kell származtatni. Fontos, hogy ez a közös osztály pehelysúlyú legyen, vagyis a felület meghatározására, nem pedig adatok tárolására összpontosítson. Az adatábrázolás meghatározását az alosztályokra kell hagyni, másképp az Elem osztály bonyolultsága túl „nehézzé" teszi a díszítőket ahhoz, hogy sokat lehessen használni belőlük. Emellett az Elem osztály számos szolgáltatással való terhelése annak valószínűségét is növeli, hogy a konkrét alosztályok olyan szolgáltatásokért fizetnek, amelyekre nincs is szükségük.
4. *Az objektumok „bőrének" megváltoztatása a „belsőségek" helyett.* A díszítőkre úgy gondolhatunk, mint egy viselkedését változtató objektum bőrére. Ezzel szemben a Stratégia mintában például az objektum belső részeit módosítjuk.
   A stratégiák alkalmazása az olyan helyzetekben jobb választás, amikor az Elem osztály eredendően nehézsúlyú, így a Díszítő minta használata túl költséges lenne. A Stratégia mintában az elemek viselkedésük egy részét külön stratégia objektumokba helyezik; ezek cseréjével változtathatók vagy bővíthetők az adott elem szolgáltatásai.
   Különböző szegélystílusokat például úgy támogathatunk, ha az elem szegélyrajzoló képességét egy önálló Szegély (Border) objektumra ruházzuk át. A Szegély egy olyan Stratégia objektum, amely egy szegélyrajzoló stratégiát zár egységbe. A stratégiák számának bővítésével ugyanaz a hatás érhető el, mint a díszítők egymásba ágyazásával.
   A MacApp 3.0-ban [App89] és a Bedrock-ban [Sym93a] például grafikus objektumok (úgynevezett „nézetek") „dísz" (adorner) objektumok listáját tartják fenn, amelyek a szegélyekhez hasonló díszítéseket csatolhatnak egy nézet objektumhoz. Ha egy né-

zethez díszek csatlakoznak, a nézet megengedi nekik, hogy feldíszítsék. A MacApp és a Bedrock azért szorul erre a megoldásra, mert a View (Nézet) osztály nehézsúlyú, így túl költséges lenne egy teljes értékű View használata pusztán egy szegély hozzáadására. Mivel a Díszítő minta csak kívülről változtat meg egy elemet, az elemnek semmit sem kell tudnia díszítőiről, vagyis azok átlátszóak számára:



Stratégiák használata esetén az elem ismeri a lehetséges bővítéseket, így hivatkoznia kell a megfelelő stratégiákra:

A stratégia alapú megközelítés az elem módosítását igényelheti, hogy az új bővítésekre lehetőséget biztosítsunk. Másfelől, egy stratégiának egyedi felülete lehet, míg egy díszítőnek illeszkednie kell az elem felületéhez. Egy szegélyrajzoló stratégiának például csak a szegélyt megjelenítő felületet (RajzolSzegély, SzerezSzélesség – DrawBorder, GetWidth stb.) kell meghatároznia, ami azt jelenti, hogy a stratégia akkor is pehelysúlyú lehet, ha maga az Elem osztály nehézsúlyú.

A MacApp és a Bedrock nem csak nézetek díszítésére használják ezt a megoldást, hanem az objektumok eseménykezelő viselkedésének bővítésére is. A nézetek mindkét rendszerben egy listát tartanak fenn a „viselkedésobjektumokról", amelyek képesek eseményeket elfogni és módosítani. A nézet a be nem jegyzett viselkedések előtt minden bejegyzett viselkedésobjektumának lehetőséget ad az esemény kezelésére, ezzel gyakorlatilag felülírja azokat. Például egy nézet úgy egészíthető ki billentyűkezeléssel, hogy bejegyzünk egy viselkedésobjektumot, amely elfogja és kezeli a billentyűzet felől érkező eseményeket.

Példakód

A következő kód azt mutatja, hogyan valósíthatók meg a felhasználói felületi díszítők a C++-ban. Feltesszük, hogy egy `VisualComponent` (LáthatóElem) nevű Elem osztállyal rendelkezünk.

```
class VisualComponent {
public:
   VisualComponent();

    virtual void Draw();
    virtual void Resize();
    //...
};
```

Meghatározzuk a `VisualComponent` `Decorator` (Díszítő) nevű alosztályát, amelyből a különböző díszítések alosztályait fogjuk származtatni.

```
class Decorator : public VisualComponent {
public:
   Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    //...
private:
    VisualComponent* _component;
};
```

A `Decorator` a konstruktorban előkészített `_component` példányváltozó által hivatkozott `VisualComponent`-et díszíti. A `Decorator` a `VisualComponent` felületében szereplő minden művelethez alapértelmezett megvalósítást ad, amelyek a kérelmet a `_component`-hez továbbítják:

```
void Decorator::Draw () {
   _component->Draw();
}

void Decorator::Resize () {
   _component->Resize();
}
```

A különböző díszítéseket a `Decorator` alosztályai határozzák meg; a `BorderDecorator` (SzegélyDíszítő) osztály például szegélyt ad befoglaló eleméhez, a `Draw` (Rajzol) művelet felülírásával. A `BorderDecorator` emellett egy `DrawBorder` (RajzolSzegély) nevű privát segédműveletet is meghatároz, amely a tényleges rajzolást végzi. Az alosztály minden más művelet megvalósítását a `Decorator`-tól örökli.

```
class BorderDecorator : public Decorator {
public:
   BorderDecorator(VisualComponent*, int borderWidth);

   virtual void Draw();
private:
   void DrawBorder(int);
private:
   int _width;
};

void BorderDecorator::Draw () {
   Decorator::Draw();
   DrawBorder(_width);
}
```

A látható elemhez görgetési képességet és árnyékolást adó `ScrollDecorator` (GörgetésDíszítő), illetve `DropShadowDecorator` (ÁrnyékVetőDíszítő) megvalósítása hasonló.

Az említett osztályokból példányokat készítünk, amelyek biztosítják a különféle díszítéseket. Az alábbi kód bemutatja, hogy a díszítők használatával hogyan hozhatunk létre egy szegéllyel ellátott, görgethető szövegnézetet (`TextView`).

Először módot kell adnunk a látható elemek ablak objektumokba helyezésére. Feltesszük, hogy `Window` (Ablak) osztályunk erre a célra egy `SetContents` (BeállítTartalom) nevű műveletet biztosít:

```
void Window::SetContents (VisualComponent* contents) {
   //...
}
```

Most már létrehozhatjuk a szövegnézőt, és egy ablakot, amelyben elhelyezzük:

```
Window* window = new Window;
TextView* textView = new TextView;
```

A `TextView` olyan `VisualComponent`, amely az ablakba helyezhető:

```
window->SetContents(textView);
```

Csakhogy mi szegéllyel ellátott és görgethető `TextView`-t szeretnénk, ezért az ablakba helyezés előtt megfelelően díszítjük:

```
window->SetContents(
   new BorderDecorator(
         new ScrollDecorator(textView), 1
   )
);
```

Mivel a `Window` tartalmát a `VisualComponent` felületen keresztül éri el, nem érzékeli a díszítő jelenlétét. Az ügyfél azonban nyomon követheti a szövegnéző állapotát, ha közvetlenül kell hozzáférnie, például ha olyan műveleteket kell meghívnia, amelyek nem részei a `VisualComponent` felületnek. Az elem azonosságára alapozó ügyfeleknek szintén közvetlenül kell hivatkozniuk az elemre.

Ismert felhasználások

A vezérlők grafikus díszítésére számos objektumközpontú felhasználói felületi elemkészlet használ díszítőket. Ilyen például az InterViews [LVC89, LCI+92], az ET++ [WGM88] vagy az ObjectWorks\Smalltalk osztálykönyvtár [Par90]. A minta különlegesebb alkalmazására példa a DebuggingGlyph az InterViews-ból, illetve a PassivityWrapper a ParcPlace Smalltalk-ból. A DebuggingGlyph hibakeresési információt ír ki, mielőtt, illetve miután továbbít egy elrendezési kérést a hozzá tartozó elemhez. Ez a nyomkövetési információ az összetételek objektumai elrendezési viselkedésének elemzésére és hibakeresésére használható. A PassivityWrapper az elemmel végezhető felhasználói műveletek engedélyezésére, illetve letiltására szolgál.

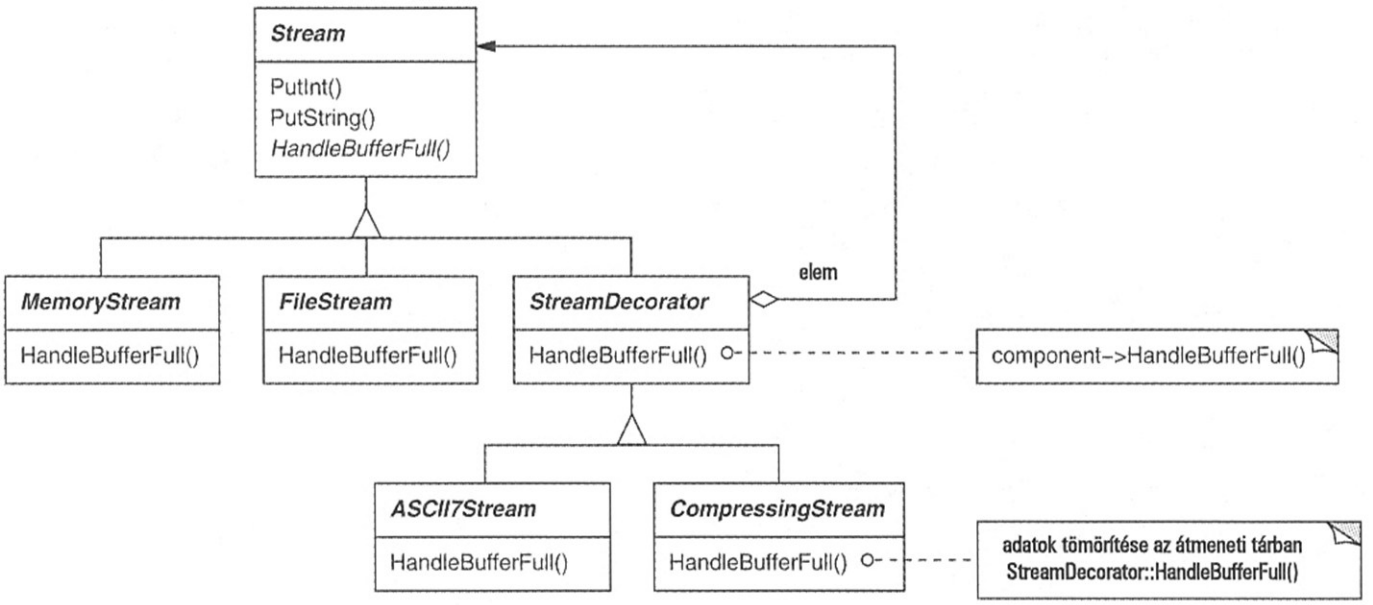
A Díszítő minta azonban nem korlátozódik a grafikus felhasználói felületekre, amint azt a következő (az ET++ folyamosztályain [WGM88] alapuló) példa is illusztrálja.

Az adatfolyamok (stream) alapvető jelentőségű fogalmai a bemeneti–kimeneti (I/O) rendszereknek. A folyam objektumok bájt- vagy karaktersorozattá való alakítására biztosít felületet, így egy objektumot későbbi felhasználás céljából fájllá vagy memóriában tárolt karakterlánccá alakíthatunk. Ennek legegyszerűbb módja egy elvont Stream (Folyam) osztály meghatározása, amely egy MemoryStream és egy FileStream alosztállyal rendelkezik. De tegyük fel, hogy az alábbiakra is lehetőséget szeretnénk adni:

- az adatfolyam tömörítésére különböző tömörítő algoritmusok segítségével (futáshosszú kódolás, Lempel-Ziv stb.);
- a folyamban tárolt adatok 7 bites ASCII karakterekké csökkentésére, hogy átvihetők legyenek egy ASCII kommunikációs csatornán.

A Díszítő mintával ezek a feladatok könnyen az adatfolyamokhoz adhatók. A következő oldalon látható diagram egy példát mutat a probléma megoldására.

Az elvont Stream osztály egy belső átmeneti tárat (buffer) tart fenn, és műveleteket biztosít az adatok folyamba helyezésére (PutInt, PutString). Amikor a tár megtelik, a Stream meghívja az elvont HandleBufferFull műveletet, amely a tényleges adatátvitelt végzi. A műveletet felülíró FileStream-változat a tár tartalmát egy fájlba írja.



A legfontosabb osztály itt a StreamDecorator (FolyamDíszítő), amely egy elemfolyamra hivatkozik, és ahhoz kérelmeket továbbít. A StreamDecorator alosztályai felülírják a HandleBufferFull műveletet, és további tevékenységeket végeznek, mielőtt meghívnák a StreamDecorator HandleBufferFull műveletét. A CompressingStream (TömörítőFolyam) alosztály például tömöríti az adatokat, míg az ASCII7Stream 7 bites ASCII-vé alakítja azokat. Ahhoz, hogy olyan FileStream-et hozzunk létre, amely tömöríti adatait és a tömörített bináris adatokat 7 bites ASCII-vé alakítja, a FileStream-et egy CompressingStream-mel és egy ASCII7Stream-mel díszítjük:

```
Stream* aStream = new CompressingStream(
   new ASCII7Stream(
         new FileStream("egyFájlNév")
   )
);
aStream->PutInt(12);
aStream->PutString("egyKarakterlánc");
```

Kapcsolódó minták

Illesztő: A díszítők annyiban különböznek az illesztőktől, hogy csak az objektumok feladatát, nem pedig azok felületét változtatják meg. Az illesztők teljesen új felülettel látják el az objektumokat.

Összetétel: A díszítők csökevényes összetételeknek tekinthetők, amelyeknek csak egy elemük van. Mindazonáltal a díszítők felelősségi körök hozzáadására valók, nem objektumösszetételre.

Stratégia: A díszítők segítségével az objektumok „bőre" változtatható meg, míg a stratégiák révén azok belseje, vagyis az objektumok módosításának két módját jelentik.

Homlokzat

Szerkezeti objektumminta

Egyéb nevek

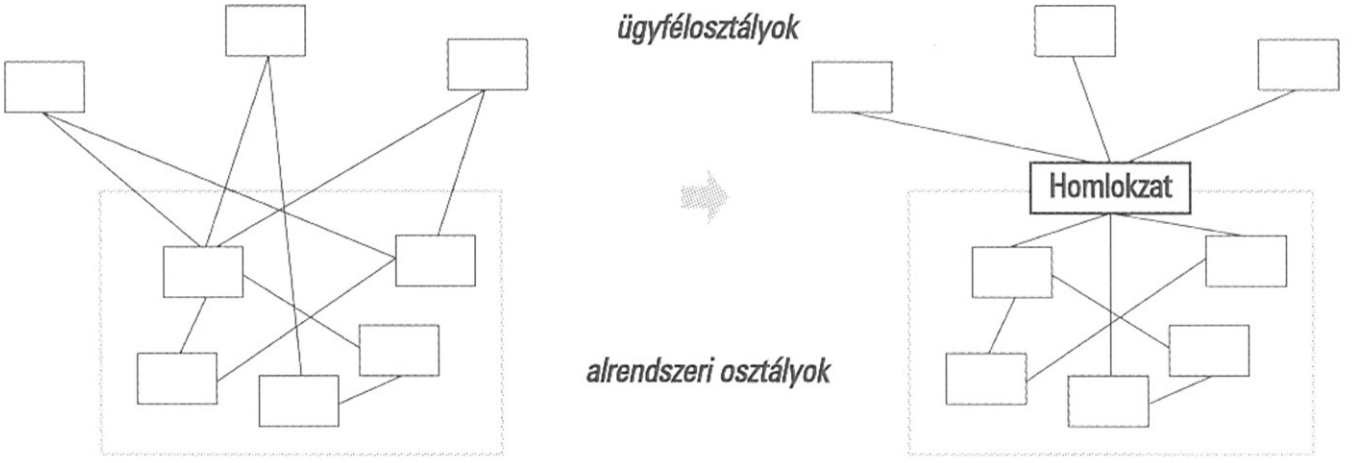
Facade, Arculat, Látszat

Cél

Egy alrendszerben felületek egy halmazához egységes felületet biztosítani. A módszerrel magasabb szintű felületet határozunk meg, amelynek révén az adott alrendszer könnyebben használhatóvá válik.

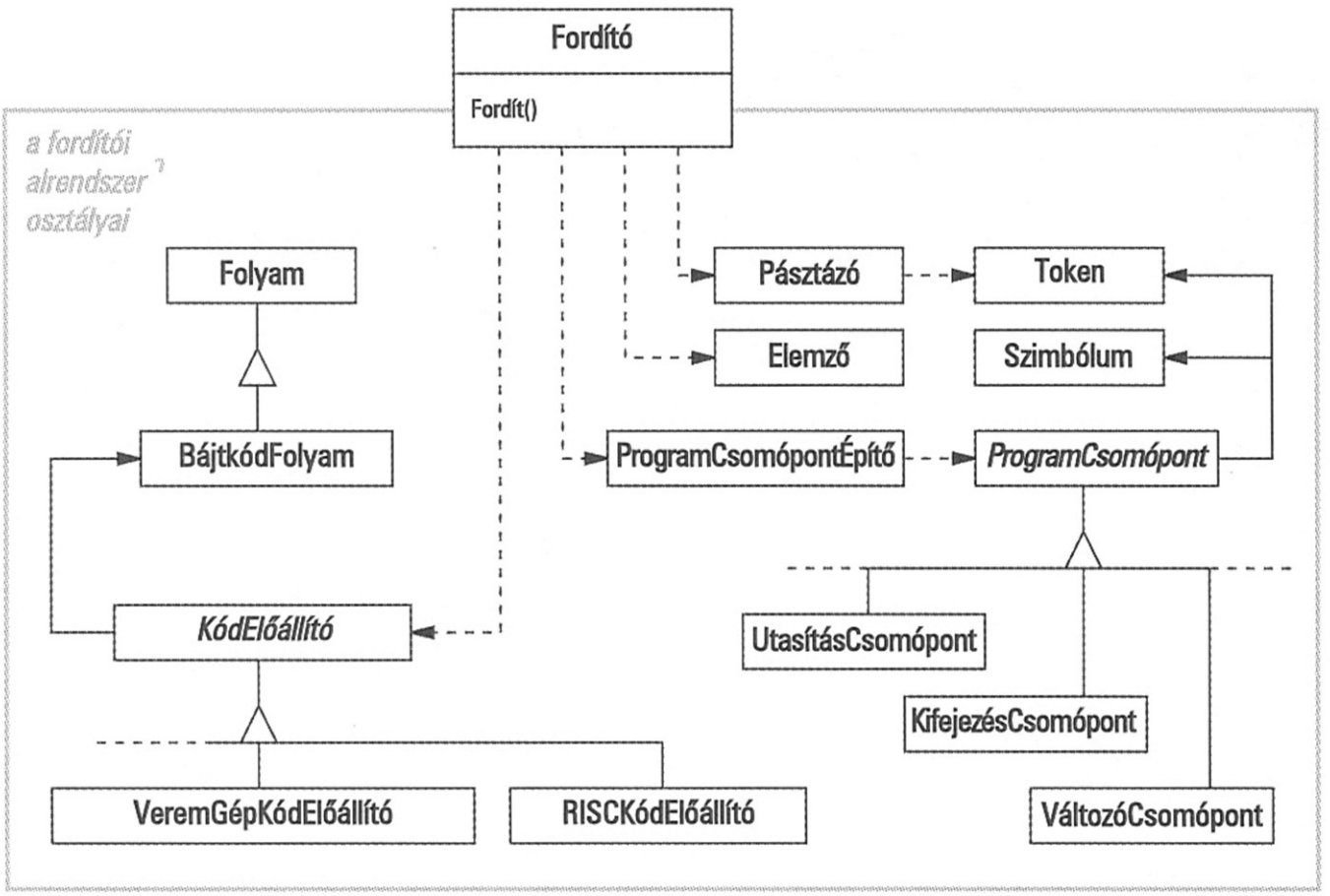
Feladat

Ha egy rendszert alrendszerekre bontunk, csökkenthetjük annak bonyolultságát. Fontos tervezési célkitűzés, hogy az alrendszerek közötti függőségeket és kommunikációt a lehető legkevesebbre csökkentsük. Ennek elérésére az egyik megoldás, ha bevezetünk egy **homlokzat** objektumot, amely egyetlen egyszerűsített felületet ad az adott alrendszer általánosabb szolgáltatásai számára.



Vegyünk például egy programozási környezetet, amely hozzáférést biztosít az alkalmazások számára a fordítói alrendszerhez. Ez az alrendszer olyan osztályokat tartalmaz, mint a Pásztázó (Scanner), az Elemző (Parser), a ProgramCsomópont (ProgramNode), a BájtkódFolyam (BytecodeStream), vagy a ProgramCsomópontÉpítő (ProgramNodeBuilder), amelyek megvalósítják a fordítót. Egyes alkalmazásoknak szükségük lehet ezen osztályok közvetlen elérésére, de a fordító legtöbb ügyfele nem törődik az olyan részletekkel, mint az elemzés vagy a kód-előállítás, csak valamilyen kódot szeretnének lefordítani. Számukra az erőteljes, de alacsonyszintű felületek a fordítói alrendszerben csak bonyolítják a feladatot.

Ahhoz, hogy az ügyfeleket ezen osztályoktól elszigetelő magasabb szintű felületet biztosíthasson, a fordítói alrendszer egy Fordító (Compiler) nevű osztályt is tartalmaz, amely egységes felület határoz meg a fordító szolgáltatásaihoz. A Fordító osztály homlokzatként viselkedik: az ügyfelek számára egyetlen egyszerű felületet nyújt a fordítói alrendszerhez, emellett összeragasztja a fordító szolgáltatásait megvalósító osztályokat, anélkül, hogy teljesen elrejtené azokat. A fordító homlokzata a legtöbb programozó dolgát megkönnyíti, miközben nem rejti el azokat az alacsonyabb szintű szolgáltatásokat, amelyekre egyeseknek szükségük lehet.



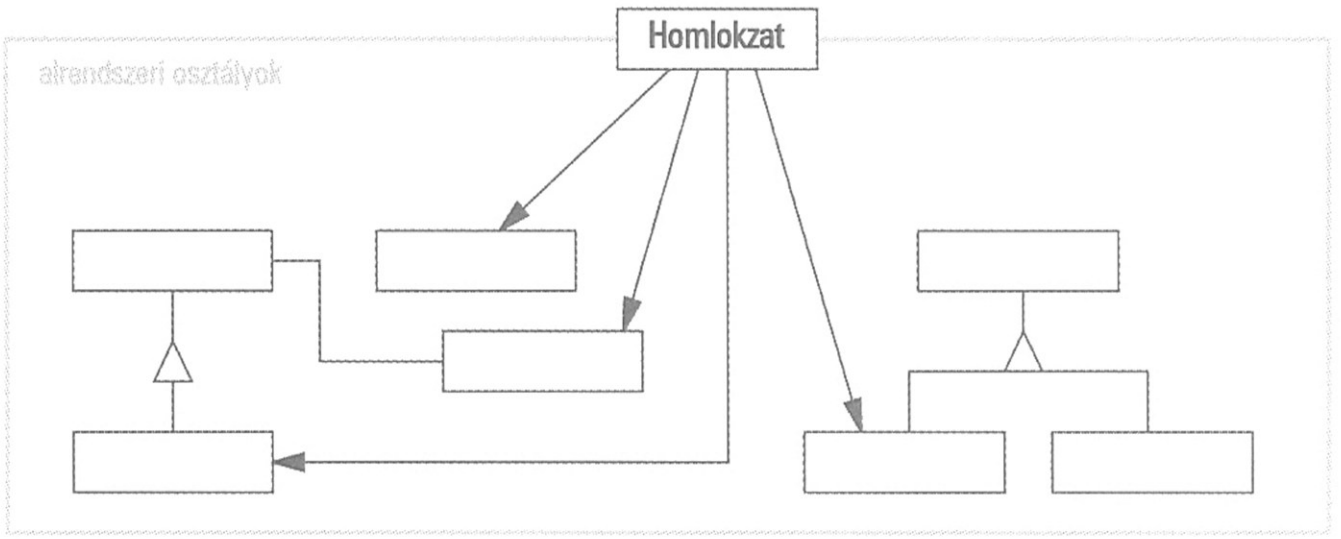
Alkalmazhatóság

A Homlokzat minta alkalmazása a következő esetekben célszerű:

- Egyszerű felületet szeretnénk nyújtani egy bonyolult alrendszerhez. Az alrendszerek a fejlesztés közben hajlamosak egyre összetettebbé válni, a legtöbb tervezési minta azonban több kisebb osztályt eredményez. Ezáltal az alrendszer jobban újrahasznosítható és könnyebben testreszabható lesz, de a testreszabást nem igénylő ügyfelek számára nehezebbé válik a használata. A homlokzat olyan egyszerű alapértelmezett nézetét adhatja az alrendszernek, ami a legtöbb ügyfél számára megfelel. Csak azoknak az ügyfeleknek kell betekinteniük a homlokzat mögé, amelyek nagyobb testreszabhatóságot igényelnek.

- Az elvont fogalmat megvalósító osztályok és az ügyfelek között számos függőség áll fenn. A homlokzat bevezetésével az alrendszer elválasztható az ügyfelektől és más alrendszerektől, így függetlenebbé és hordozhatóbbá válik.
- Az alrendszereket rétegezni szeretnénk. Egy homlokzattal belépési pontot határozhatunk meg minden alrendszeri szinthez. Ha az egyes alrendszerek egymástól függnek, e függőségek egyszerűsíthetők, ha az alrendszerek csak homlokzatukon keresztül társaloghatnak egymással.

Szerkezet



Résztvevők

- **Homlokzat** (Fordító)
  - Tudja, mely alrendszeri osztályok felelnek egy adott kérelemért.
  - Az ügyfélkérelmek kezelését a megfelelő alrendszeri objektumokra ruházza át.
- **alrendszeri osztályok** (Pásztázó, Elemző, ProgramCsomópont stb.)
  - Megvalósítják az alrendszer szolgáltatásait.
  - Elvégzik a Homlokzat objektum által rájuk bízott feladatot.
  - Nincs tudomásuk a homlokzatról, vagyis nem hivatkoznak arra.

Együttműködés

- Az ügyfelek úgy lépnek kapcsolatba az alrendszerrel, hogy kérelmeket küldenek a Homlokzatnak, ami aztán továbbítja azokat a megfelelő alrendszeri objektumhoz vagy objektumokhoz. Bár a tényleges munkát az alrendszeri objektumok végzik, a homlokzatnak is szüksége lehet bizonyos feladatok elvégzésére, hogy felületét az alrendszeri felületekhez igazítsa.
- A homlokzatot használó ügyfeleknek nem kell közvetlenül elérniük az alrendszeri objektumokat.

Következmények

A Homlokzat tervezési minta előnyei a következők:

1. Elszigeteli az ügyfeleket az alrendszeri összetevőktől, így csökkenti azon objektumok számát, amelyekkel az ügyfeleknek foglalkozniuk kell, és megkönnyíti az alrendszer használatát.
2. Laza csatolást hoz létre az alrendszer és ügyfelei között. Az alrendszerek összetevői gyakran szoros csatolásúak, a laza csatolás révén azonban anélkül változtathatók, hogy ez hatással lenne az alrendszer ügyfeleire. A homlokzatok segítenek a rendszer rétegzésében, és az objektumok közti függőségek szabályozásában, megszüntetve a bonyolult vagy körkörös függőségeket, ami igen lényeges, ha az alrendszert és az ügyfelet egymástól függetlenül valósítjuk meg.
   A fordítási függőségek csökkentése létfontosságú a nagy szoftverrendszerekben, hiszen ha az alrendszeri osztályok megváltozása nem von maga után nagy számú újrafordítást, időt takaríthatunk meg. A függőségek homlokzatokkal való csökkentése korlátozza az újrafordítás szükségességét, ha egy fontos alrendszerben apró változás történik. A homlokzat emellett a rendszer más felületre történő átültetését is megkönnyíti, mert használata mellett kevésbé valószínű, hogy egy alrendszer felépítése során az összes többi alrendszer felépítésére is szükség volna.
3. Nem akadályozza meg, hogy az alkalmazások – ha szükségük van rá – használják az alrendszeri osztályokat, így választhatunk a használat könnyebbsége és a nagyobb általánosság között.

Megvalósítás

A Homlokzat minta megvalósítása során a következőkre kell figyelni:

1. *Az ügyfél–alrendszer csatolás csökkentése.* Az ügyfelek és az alrendszer között fennálló csatolás tovább csökkenthető, ha a homlokzatot elvont osztállyá tesszük, amely az alrendszer különböző megvalósításaihoz konkrét alosztályokkal rendelkezik. Ekkor az ügyfelek az elvont Homlokzat osztály felületén keresztül érintkezhetnek az alrendszerrel. Ez az elvont csatolás megakadályozza, hogy az ügyfelek tudomással bírjanak arról, hogy az alrendszer melyik megvalósítását használják.
   Az alosztályok származtatása helyett azt is megtehetjük, hogy a Homlokzat objektumot különböző alrendszeri objektumokkal állítjuk be. Ekkor a homlokzat testreszabásához egyszerűen csak ki kell cserélnünk egy vagy több alrendszeri objektumát.
2. *Nyilvános vagy privát alrendszeri osztályok?* Az alrendszerek annyiban hasonlítanak az osztályokhoz, hogy nekik is van felületük, és egységbe zárnak valamit – az osztályok állapotokat és műveleteket, az alrendszerek osztályokat. Ahogy pedig az osztályoknak van nyilvános és privát felületük, az alrendszerek is rendelkezhetnek ilyenekkel.
   Az alrendszerek nyilvános felülete olyan osztályokból áll, amelyeket minden ügyfél elérhet, míg a privát felület csak az alrendszert bővítő objektumok számára hozzáfér-

hető. A Homlokzat osztály természetesen a nyilvános felület része, de nem az egyetlen, más alrendszeri osztályok is nyilvánosak. A fordítói alrendszer Elemző és Pásztázó osztályai például szintén a nyilvános felület részei.

Az alrendszeri osztályok nyilvánossá tétele hasznos lehet, mégis kevés objektumközpontú nyelv támogatja. A C++ és a Smalltalk hagyományosan globális névtérbe helyezték az osztályokat, de a C++ nyelvet szabványosító bizottság később a névterek hozzáadásával lehetővé tette, hogy csak a nyilvános alrendszeri osztályokat fedjük fel.

Példakód

Nos, akkor nézzük meg, hogyan láthatunk el homlokzattal egy fordítói alrendszert.

Az alrendszer meghatároz egy `BytecodeStream` (BájtkódFolyam) osztályt, amely egy `Bytecode` (Bájtkód) objektumokból álló adatfolyamot valósít meg. A `Bytecode` objektumok bájtkódokat zárnak egységbe, amelyek gépi utasításokat fogalmaznak meg. Az alrendszer ezenkívül meghatároz egy `Token` nevű osztályt is azon objektumok számára, amelyek a programozási nyelv alapelemeit (token) zárják egységbe.

A `Scanner` (Pásztázó) osztály egy karakterfolyamot vesz, és egyszerre egy elemet (tokent) vizsgálva tokenfolyammá alakítja azt.

```
class Scanner {
public:
   Scanner(istream&);
   virtual ~Scanner();

   virtual Token& Scan();
private:
   istream& _inputStream;
};
```

A `Parser` (Elemző) osztály egy `ProgramNodeBuilder` (ProgramCsomópontÉpítő) segítségével elemzőfát épít fel a `Scanner` tokenjeiből.

```
class Parser {
public:
   Parser();
   virtual ~Parser();

   virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

A `Parser` a `ProgramNodeBuilder` visszahívásával fokozatosan felépíti az elemzőfát. Ezek az osztályok az Építő mintát követve működnek együtt.

```
class ProgramNodeBuilder {
public:
   ProgramNodeBuilder();

   virtual ProgramNode* NewVariable(
         const char* variableName
   ) const;

   virtual ProgramNode* NewAssignment(
         ProgramNode* variable, ProgramNode* expression
   ) const;

   virtual ProgramNode* NewReturnStatement(
         ProgramNode* value
   ) const;

   virtual ProgramNode* NewCondition(
         ProgramNode* condition,
         ProgramNode* truePart, ProgramNode* falsePart
   ) const;
//...

   ProgramNode* GetRootNode();
private:
   ProgramNode* _node;
};
```

Az elemzőfát olyan `ProgramNode` (ProgramCsomópont) alosztályok példányai építik fel, mint a `StatementNode` (UtasításCsomópont), az `ExpressionNode` (KifejezésCsomópont) és így tovább. A `ProgramNode` hierarchia az Összetétel tervezési mintát követi. A `ProgramNode` a program-csomópont és esetleges gyermekei kezelésére határoz meg felületet.

```
class ProgramNode {
public:
   // program-csomópont kezelése
   virtual void GetSourcePosition(int& line, int& index);
   //...

   // gyermekek kezelése
   virtual void Add(ProgramNode*);
   virtual void Remove(ProgramNode*);
   //...
```

```
    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

A `Traverse` (Bejár) művelet egy `CodeGenerator` (KódElőállító) objektumot vesz, amit a `ProgramNode` alosztályok arra használnak, hogy gépi kódot állítsanak elő egy `BytecodeStream` `Bytecode` objektumai formájában. A `CodeGenerator` osztály maga egy látogató (lásd a Látogató mintát az 5. fejezetben).

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    //...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& _output;
};
```

A `CodeGenerator` olyan alosztályokkal rendelkezik, mint a `StackMachineCode-Generator` (VeremGépKódElőállító) vagy a `RISCCodeGenerator` (RISCKódElőállító), amelyek különböző hardver-architektúrák számára állítanak elő gépi kódot.

A `ProgramNode` valamennyi alosztálya megvalósítja a `Traverse` műveletet, hogy meghívja azt gyermekobjektumaira. A gyermekobjektumok azután ugyanígy meghívják a műveletet saját gyermekeikre, és így tovább. Az `ExpressionNode` például a következőképpen határozza meg a `Traverse` műveletet:

```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
            i.CurrentItem()->Traverse(cg);
    }
}
```

Az eddig tárgyalt osztályok építik fel a fordítói alrendszert. Most bevezetünk egy `Compiler` (Fordító) nevű homlokzatosztályt, amely összefogja a rendszer részeit. A `Compiler` egyszerű felületet biztosít a forrás lefordítására, és az adott gépfelépítésnek megfelelő kód előállítására.

```
class Compiler {
public:
   Compiler();

   virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
   istream& input, BytecodeStream& output
) {
   Scanner scanner(input);
   ProgramNodeBuilder builder;
   Parser parser;

   parser.Parse(scanner, builder);

   RISCCodeGenerator generator(output);
   ProgramNode* parseTree = builder.GetRootNode();
   parseTree->Traverse(generator);
}
```

Ez a megvalósítás mereven bekódolja a használandó kód-előállító típusát, így a programozóknak nem kell megadniuk a cél-architektúrát. Ez a megoldás akkor ésszerű, ha csak egyetlen célgéptípus lehetséges. Ha viszont nem ez a helyzet, a `Compiler` konstruktorát úgy kell megváltoztatnunk, hogy egy `CodeGenerator` paramétere legyen, így a programozók a `Compiler` példányosításakor meghatározhatják a használandó kód-előállítót. A fordító homlokzata más résztvevőket is paraméterré tehet, például a `Scanner`-t és a `ProgramNodeBuilder`-t, ami növeli a rugalmasságot, de eltávolodik az eredeti céltól, ami általában a felület egyszerűsítése.

Ismert felhasználások

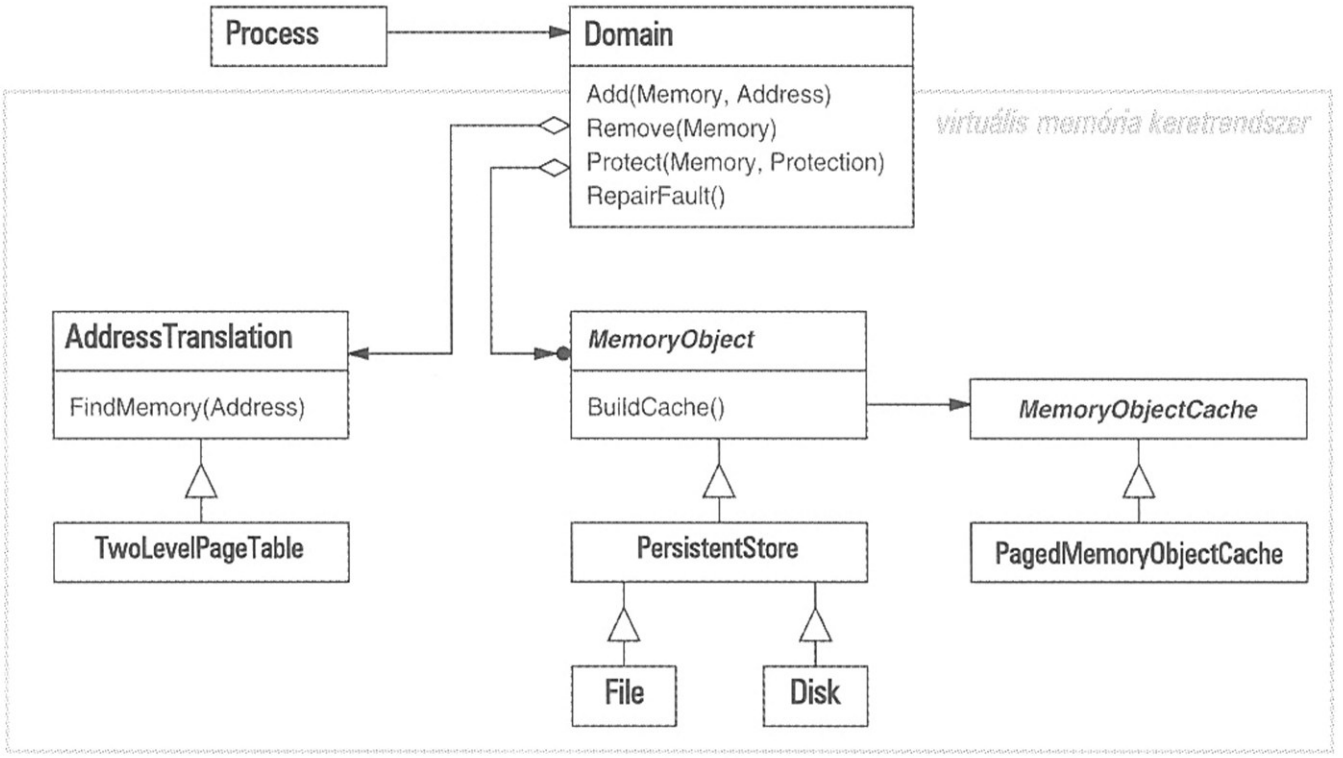
A Példakód részben szereplő fordító az ObjectWorks\Smalltalk fordítórendszerén [Par90] alapult.

Az ET++ alkalmazás-keretrendszerben [WGM88] az alkalmazások beépített böngészővel vizsgálhatják objektumaikat futás közben. Ezek a böngészők önálló alrendszerekben kapnak helyet, ami egy „ProgrammingEnvironment" (programozási környezet) nevű homlokzatosztályt tartalmaz. Ez a homlokzat a böngészők elérésére szolgáló műveleteket (például InspectObject, InspectClass) határozza meg.

Az ET++ alkalmazások „hamisíthatják" is a beépített böngészőtámogatást. Ilyen esetben a ProgrammingEnvironment a kérelmeket null-műveletekként valósítja meg, vagyis azok nem csinálnak semmit. Csak az ETProgrammingEnvironment alosztály ad olyan megvalósí-

tást a kérelmekhez, amely meghatározza a megfelelő böngészőket megjelenítő műveleteket. Az alkalmazásnak nincs tudomása arról, hogy jelen van-e egy elérhető böngészőkörnyezet, vagyis az alkalmazás és a böngésző alrendszer között elvont csatolás áll fenn.

A Choices operációs rendszer [CIRM93] arra használ homlokzatokat, hogy több keretrendszert egyesítsen. A Choices kulcsfogalmai a folyamat (process), a tárolás (storage) és a címtér (address space). Mindegyikhez tartozik egy keretrendszerként megvalósított alrendszer, amely támogatja a Choices különböző hardverfelületekre történő átültetését. Az alrendszerek közül kettőnek van „képviselője" (vagyis homlokzata); ezek neve FileSystemInterface (FájlRendszerFelület, tárolás), illetve Domain (Tartomány, címterek).



A virtuális memória keretrendszernek például a Domain (Tartomány) a homlokzata. A tartományok egy-egy címteret képviselnek, és a virtuális címek és eltolások memória-objektumokra vagy fájlokra való leképezését biztosítják. A Domain fő műveletei arra szolgálnak, hogy egy memória-objektumot elhelyezzünk egy adott címen, eltávolítsunk egy ilyen objektumot, illetve kezeljünk egy laphibát.

Amint a fenti diagram is mutatja, a virtuális memória alrendszer belül a következő összetevőkből áll:

- MemoryObject (MemóriaObjektum, egy adattárat képvisel);
- MemoryObjectCache (MemóriaObjektumTár, a memória-objektumok adatait átmenetileg a fizikai memóriában tárolja; valójában egy stratégia, amely az átmeneti tárolás módját zárja egységbe);
- AddressTranslation (CímFordítás, a címfordító hardvert zárja egységbe).

Amikor egy laphiba miatti megszakítás következik be, a RepairFault (JavítHiba) művelet hívódik meg. A Domain megkeresi a hibát okozó címen levő memória-objektumot, és az ahhoz rendelt átmeneti tárra bízza a RepairFault végrehajtását. A tartományok összetevőik megváltoztatásával testreszabhatók.

Kapcsolódó minták

A Homlokzat mintával együtt használható az Elvont Gyár, így felületet biztosíthatunk az alrendszeri objektumok alrendszertől független létrehozására. Az Elvont Gyár a rendszerfüggő osztályok elrejtésében is helyettesítheti a Homlokzat mintát.

A Közvetítő minta annyiban hasonlít a Homlokzatra, hogy szintén meglevő osztályok szolgáltatásait vonatkoztatja el. A Közvetítő célja azonban a társobjektumok kapcsolattartásának elvonttá tétele, gyakran az olyan szolgáltatások központosításával, amelyek egyik objektumhoz sem tartoznak. A közvetítő kollégái ismerik a közvetítőt, és az egymás közötti közvetlen kommunikáció helyett vele társalognak. A homlokzat ezzel szemben csupán elvonttá teszi az alrendszeri objektumok felületét, hogy megkönnyítse használatukat; új szolgáltatásokat nem határoz meg, az alrendszeri osztályok pedig nem tudnak róla.

Általában csak egyetlen homlokzatobjektumra van szükség, így a homlokzatok gyakran Egykék.

Pehelysúlyú

Szerkezeti objektumminta

Egyéb nevek

Flyweight, Könnyűsúlyú

Cél

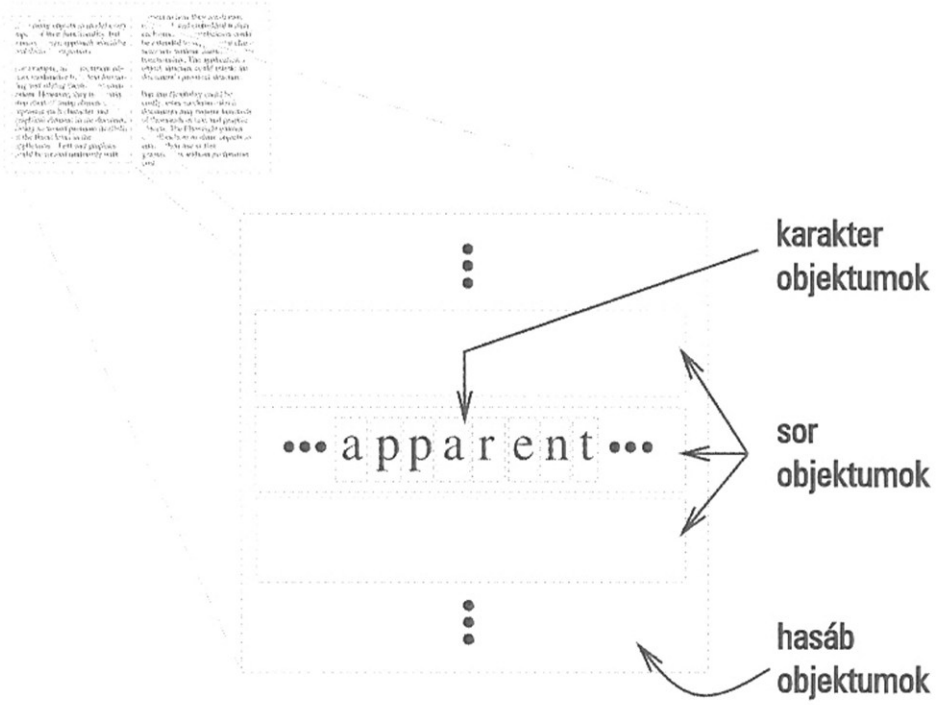
Megosztás révén támogatni a nagy finomságú objektumok tömegeinek hatékony felhasználását.

Feladat

Egyes alkalmazások nyerhetnek azon, ha mindenre külön objektumot használnak, de egy naiv megvalósítás igen költséges lehet.

A legtöbb dokumentum- vagy szövegszerkesztő például bizonyos fokig modularizált szövegformázó és –szerkesztő képességekkel rendelkezik. Az objektumközpontú szövegszer-

kesztők jellemzően objektumokkal ábrázolják az olyan beágyazott elemeket, mint a táblázatok és ábrák. A dokumentum egyes karaktereit azonban már nem önálló objektumok képviselik, még ha ez a program legfinomabb szintjén is jelentene rugalmasságot. Ha így lenne, a karaktereket és beágyazott elemeket egységesen kezelhetnénk, kirajzolási és formázási módjuktól függően. A program anélkül lenne bővíthető új karakterkészletek támogatásával, hogy a többi szolgáltatásra ez hatással lenne. Az alkalmazás objektumszerkezete pontosan tükrözhetné a dokumentum „fizikai" szerkezetét. Az alábbi diagram azt mutatja, hogyan jelölhet karaktereket objektumokkal egy szövegszerkesztő:



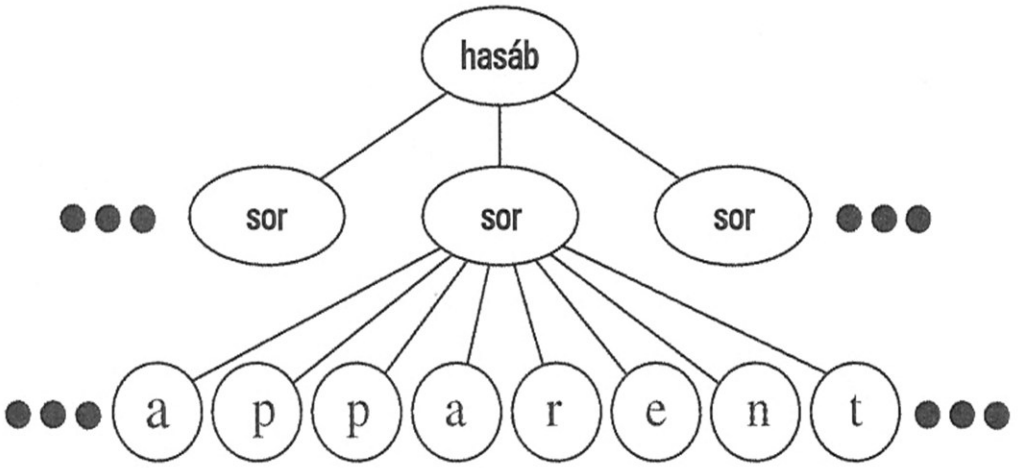
A módszer hátulütője a magas költség. Még a szerényebb méretű dokumentumok is karakter objektumok százezreit igényelnék, ami rengeteg memóriát emésztene fel, a futási sebesség pedig elfogadhatatlan szintre csökkenne. A Pehelysúlyú minta azonban megmutatja, hogyan oszthatunk meg objektumokat finomabb szinten való használatra anélkül, hogy a költségek az egekbe szöknének.

A **pehelysúlyú** objektum olyan megosztott objektum, amely egyidejűleg több környezetben használható. Mindegyik környezetben önálló objektumként viselkedik, vagyis nem különböztethető meg egy nem megosztott objektum példányától, és nem élhet feltételezésekkel működési környezetéről. A minta kulcsa a **belső** és **külső** állapot(információk) megkülönböztetése. A belső állapot a pehelysúlyú objektumban tárolódik, és olyan információkból áll, amelyek függetlenek a pehelysúlyú objektum környezetétől, így megoszthatók. A külső állapot ezzel szemben a környezettől függően változik, ezért megosztása nem lehetséges. Az ügyfélobjektumok feladata, hogy külső állapotot adjanak át a pehelysúlyú objektumnak, amikor az igényli.

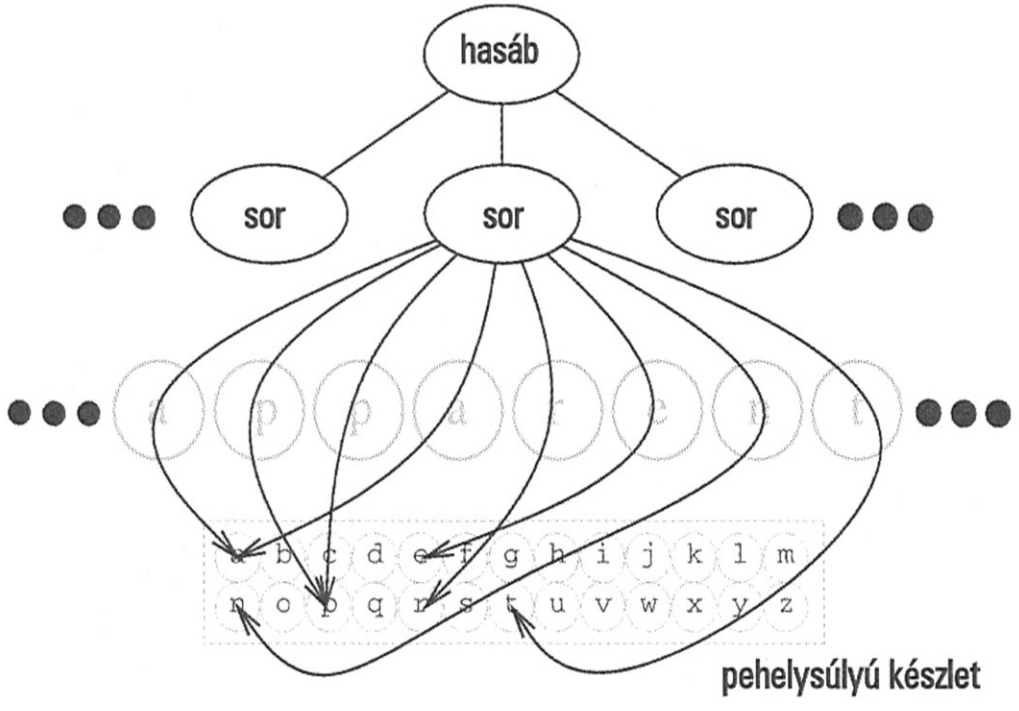
A pehelysúlyú objektumok olyan fogalmakat vagy egyedeket modelleznek, amelyek száma normális esetben túl nagy ahhoz, hogy objektumokkal ábrázolhatók legyenek. Egy szövegszerkesztőben például az ábécé minden betűjéhez létrehozhatunk egy-egy pehelysúlyú ob-

jektumot. Ezek csak egy karakterkódot tárolnak; a dokumentumban elfoglalt helyet és a betűstílust a szöveg-elrendező algoritmusok és a karakter megjelenési helyén érvényben levő formázási parancsok alapján határozzuk meg. A karakterkód belső állapotinformáció, míg a többi információ külső.

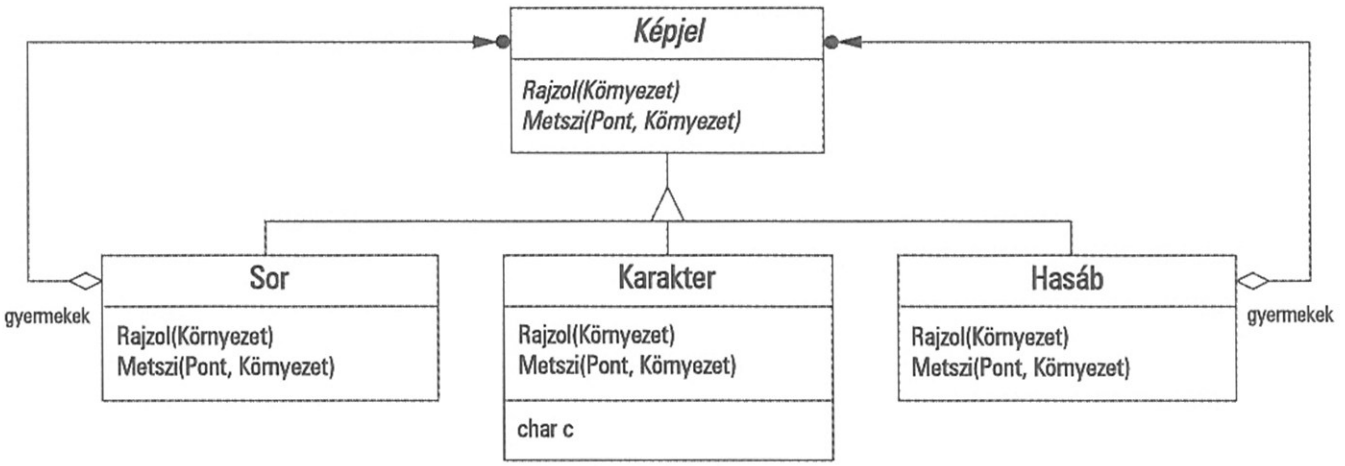
Logikailag a dokumentum egy adott karakterének minden előfordulásához létezik egy objektum:



Fizikailag azonban karakterenként egy megosztott pehelysúlyú objektummal rendelkezünk, amely különböző környezetekben jelenik meg a dokumentumszerkezetben. Egy adott karakter objektum valamennyi előfordulása ugyanarra a példányra hivatkozik a pehelysúlyú objektumok megosztott gyűjtőtárában:



A fenti objektumok osztályszerkezetét a következő ábrán mutatjuk be. A Képjel (Glyph) a grafikus objektumok elvont osztálya, amelyek közül néhány pehelysúlyú lehet; a külső állapoton alapuló műveletek paraméterként kapják meg. A Rajzol (Draw) és a Metszi (Intersects) műveleteknek például tudniuk kell, milyen környezetben van a képjel, mielőtt elvégezhetnék munkájukat.



Az „a" betűt képviselő pehelysúlyú objektum csak a megfelelő karakterkódot tárolja, a helyet vagy a betűtípust nem. Az objektum kirajzolásához szükséges környezetfüggő információkat az ügyfelek adják át. Egy Sor (Row) képjel például tudja, hogy gyermekeinek hol kell kirajzolniuk magukat, hogy egy sorban helyezkedjenek el, így a rajzolási kérelemben átadhatja nekik a helyüket.

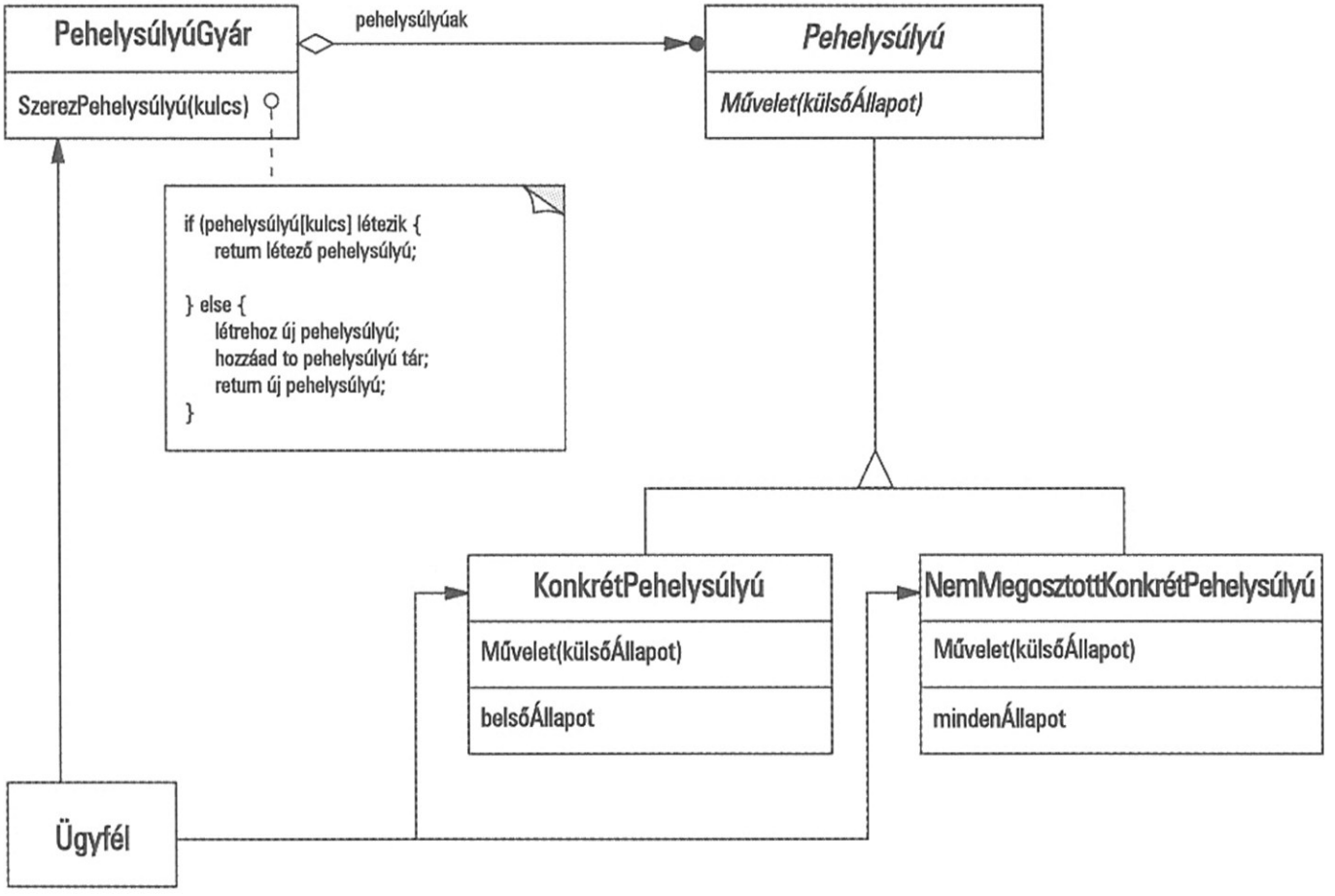
Mivel a különböző karakter objektumok száma jóval kisebb, mint a dokumentumban levő karaktereké, lényegesen kevesebb objektumunk lesz, mint amennyit egy naiv megvalósításban használnánk. Egy egyetlen betűtípussal és színnel írt dokumentum 100 karakter objektum sorrendje alapján előállítható (ez durván az ASCII karakterkészlet mérete), függetlenül a dokumentum hosszától, és mivel a legtöbb dokumentum nem használ tíznél több betűtípus–szín kombinációt, az említett szám a gyakorlatban nem nő jelentősen. Így válik az objektum alapú megközelítés önálló karakterek esetében is hasznosíthatóvá.

Alkalmazhatóság

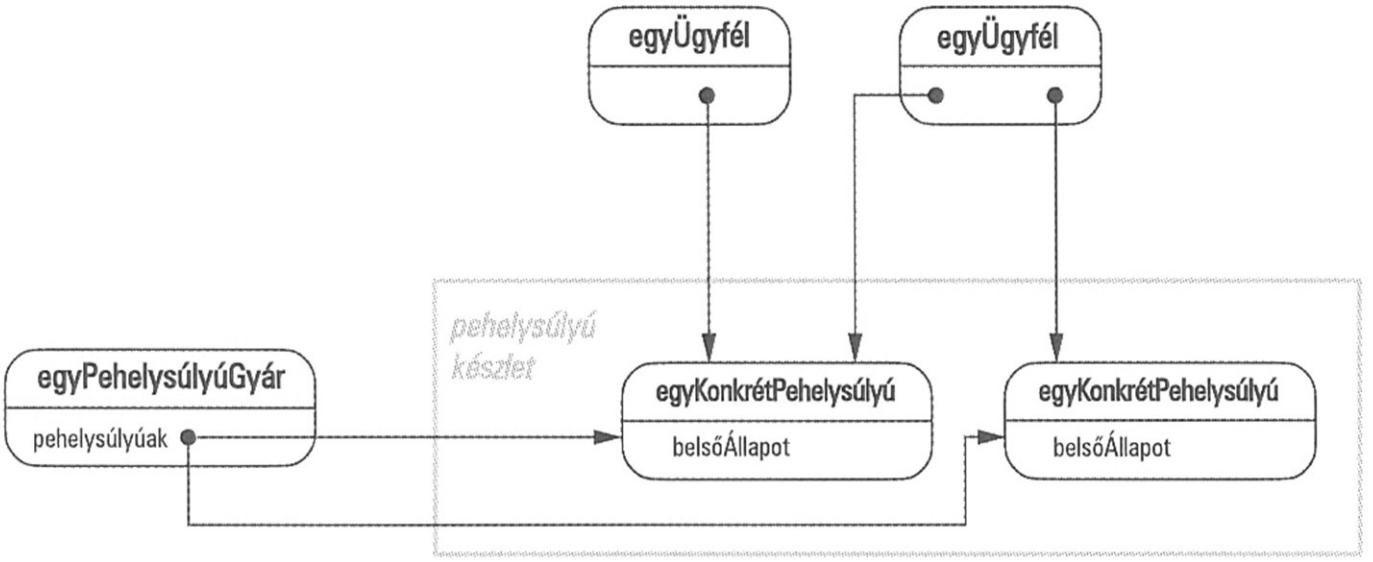
A Pehelysúlyú minta hatékonysága nagyban függ attól, hol és hogyan használjuk. Csak akkor alkalmazzuk, ha az alábbi állítások *mindegyike* igaz:

- Az alkalmazás nagy számú objektumot használ.
- Az objektumok nagy száma miatt magas a tárköltség.
- A legtöbb objektum-tulajdonság (állapot) külsővé tehető.
- A külső állapot eltávolítása után objektumok egész csoportjai helyettesíthetők viszonylag kevés megosztott objektummal.
- Az alkalmazás nem az objektumok azonosságán alapul. Miután a pehelysúlyú objektumok megoszthatók, egy azonosságvizsgálat fogalmilag különböző objektumok esetében is „igaz" eredményt adhat.

Szerkezet



A következő objektumdiagram a pehelysúlyú objektumok megosztásának módját mutatja:



Résztvevők

- **Pehelysúlyú** (Képjel)
  - Felületet vezet be, amelyen keresztül a pehelysúlyú objektumok külső állapotinformációkat fogadhatnak, és azok alapján műveleteket végezhetnek.
- **KonkrétPehelysúlyú** (Karakter)
  - Megvalósítja a Pehelysúlyú (Flyweight) felületet és tárat biztosít az esetleges belső állapotinformációk számára. A KonkrétPehelysúlyú (ConcreteFlyweight) objektumoknak megoszthatóknak kell lenniük, a bennük tárolt állapotnak pedig belsőnek kell lennie, vagyis függetlennek a KonkrétPehelysúlyú objektum környezetétől.
- **NemMegosztottKonkrétPehelysúlyú** (Sor, Hasáb)
  - Nem minden Pehelysúlyú alosztálynak kell megoszthatónak lennie. A Pehelysúlyú felület csak *lehetővé teszi* a megosztást, nem kényszerít rá. A NemMegosztottKonkrétPehelysúlyú (UnsharedConcreteFlyweight) objektumok (ahogy a Sor és Hasáb – Row, Column – osztályok is) gyakran rendelkeznek KonkrétPehelysúlyú gyermekobjektumokkal a pehelysúlyú objektumszerkezet valamely szintjén.
- **PehelysúlyúGyár**
  - Pehelysúlyú objektumokat hoz létre és kezel.
  - Gondoskodik a pehelysúlyú objektumok megfelelő megosztásáról. Amikor egy ügyfél pehelysúlyú objektumot kér, a PehelysúlyúGyár (FlyweightFactory) átad egy meglevő példányt, vagy létrehoz egyet, ha még egy sem létezik.
- **Ügyfél**
  - Hivatkozást tart fenn egy vagy több pehelysúlyú objektumra.
  - Kiszámítja vagy tárolja a pehelysúlyú objektum(ok) külső állapotát.

Együttműködés

- A pehelysúlyú objektumok által igényelt állapotinformációk belsők vagy külsők lehetnek. A belső állapotot a KonkrétPehelysúlyú objektum tárolja, a külső állapot tárolásáról, illetve kiszámításáról az Ügyfél (Client) objektumok gondoskodnak. Az ügyfelek akkor adják át a külső állapotot a pehelysúlyú objektumnak, amikor meghívják annak műveleteit.
- Az ügyfeleknek nem szabad közvetlenül példányosítaniuk a KonkrétPehelysúlyú objektumokat. Kizárólag a PehelysúlyúGyár objektumtól szerezhetik be azokat, hogy megosztásuk megfelelő legyen.

Következmények

A pehelysúlyú objektumok a külső állapotinformációk átvitele, megkeresése, illetve kiszámítása miatt lassíthatják a program futását, különösen ha a külső állapot korábban belső állapotként tárolódott. Ezt a költséget azonban ellensúlyozza a tárhely-megtakarítás, ami egyre nagyobb lesz, ahogy több pehelysúlyú objektumot osztunk meg.

A tárhely-megtakarítás több tényezőtől függ:

- a példányok számának a megosztásból következő csökkenésétől,
- az objektumonkénti belső állapotinformációk mennyiségétől, valamint
- attól, hogy a külső állapotot kiszámítjuk vagy tároljuk.

Minél több pehelysúlyú objektumot osztunk meg, annál több helyet takarítunk meg, és a megtakarítás a megosztott állapotinformációk mennyiségének növelésével tovább növelhető. A legtöbbet akkor nyerünk, ha az objektumok jelentős mennyiségű belső és külső állapotinformációt használnak fel, a külső állapotot pedig számítjuk, és nem tároljuk. A szükséges tárhely így két módon csökken: a megosztás a belső állapot tárolási költségét csökkenti, míg a külső állapotét „elcseréljük" a kiszámításához szükséges időre.

A Pehelysúlyú mintát gyakran együtt használják az Összetétel mintával, hogy egy hierarchikus szerkezetet egy olyan gráffal ábrázoljanak, amelynek megosztott levél-csomópontjai vannak. A megosztás egyik következménye, hogy a pehelysúlyú levél-csomópontok nem tárolhatnak mutatót szülőjükre; helyette a külső állapot részeként kapják meg a szülőmutatót, ami lényegesen befolyásolja, hogy a hierarchia objektumai hogyan kommunikálnak egymással.

Megvalósítás

A Pehelysúlyú minta megvalósítása során a következőkre kell figyelnünk:

1. *A külső állapot eltávolítása.* A minta alkalmazhatóságát nagy részben az határozza meg, mennyire egyszerű a külső állapotinformációk azonosítása és eltávolítása a megosztott objektumokból. A külső állapot eltávolítása ugyanis nem segít a tárköltség csökkentésében, ha ugyanolyan sokféle külső állapotinformáció létezik, mint objektum (a megosztás előtt). Ideális esetben a külső állapot egy önálló objektumszerkezetből számítható ki, amelynek jóval kisebb a tárigénye.
   Szövegszerkesztőnkben például a tipográfiai információk „térképét" külön szerkezetben tárolhatjuk, ahelyett, hogy a betűtípust és -stílust tárolnánk minden egyes karakter objektumhoz. A „térkép" nyomon követi az azonos tipográfiai jellemzőkkel írt karaktereket; amikor egy karakter kirajzolja önmagát, ezeket a jellemzőket a rajzolási bejárás „mellékhatásaként" kapja meg. Mivel a dokumentumok általában csak néhány betűtípust és -stílust használnak, ezen információk külső tárolása az egyes karakter objektumok számára jóval hatékonyabb, mint a belső tárolás.
2. *A megosztott objektumok kezelése.* Mivel az objektumokat megosztjuk, az ügyfeleknek nem szabad közvetlenül példányosítaniuk azokat. Egy adott pehelysúlyú objektum megkeresését a PehelysúlyúGyár teszi lehetővé az ügyfelek számára. A PehelysúlyúGyár objektumok gyakran egy társításos tárat (asszociatív tárat) használnak, hogy az ügyfelek megkereshessék a számukra érdekes pehelysúlyú objektu-

mokat. A szövegszerkesztő pehelysúlyúobjektum-gyára például egy karakterkódokkal indexelt táblázatot tarthat fenn, amelyből a kezelő a kód alapján visszaadja a megfelelő pehelysúlyú objektumot, létrehozva azt, ha még nem létezik.

A megoszthatóság maga után von valamiféle hivatkozás-számlálást vagy szemétgyűjtést is, hogy felszabadíthassuk a pehelysúlyú objektumok által elfoglalt tárat, amikor már nincs rájuk szükség. Mindazonáltal egyikre sincs szükség, ha a pehelysúlyú objektumok száma kicsi és rögzített (ilyenek például az ASCII karakterkészlet pehelysúlyú objektumai) – ilyenkor érdemes az objektumokat mindig a kezünk ügyében tartani.

Példakód

Térjünk vissza a dokumentumformázó példához, amelyben most egy `Glyph` (Képjel) nevű alaposztályt határozunk meg a pehelysúlyú grafikus objektumok számára. Logikailag a képjelek összetételek (lásd az Összetétel mintát), amelyeknek grafikai jellemzőik vannak, és képesek kirajzolni önmagukat. Mi most csak a betűtípus jellemzőre összpontosítunk, de ugyanez a megközelítés alkalmazható bármely más grafikai jellemzőre, amellyel egy képjel rendelkezhet.

```
class Glyph {
public:
   virtual ~Glyph();

   virtual void Draw(Window*, GlyphContext&);

   virtual void SetFont(Font*, GlyphContext&);
   virtual Font* GetFont(GlyphContext&);

   virtual void First(GlyphContext&);
   virtual void Next(GlyphContext&);
   virtual bool IsDone(GlyphContext&);
   virtual Glyph* Current(GlyphContext&);

   virtual void Insert(Glyph*, GlyphContext&);
   virtual void Remove(GlyphContext&);
protected:
   Glyph();
};
```

A `Character` (Karakter) alosztály csak egy karakterkódot tárol:

```
class Character : public Glyph {
public:
   Character(char);
```

```
    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

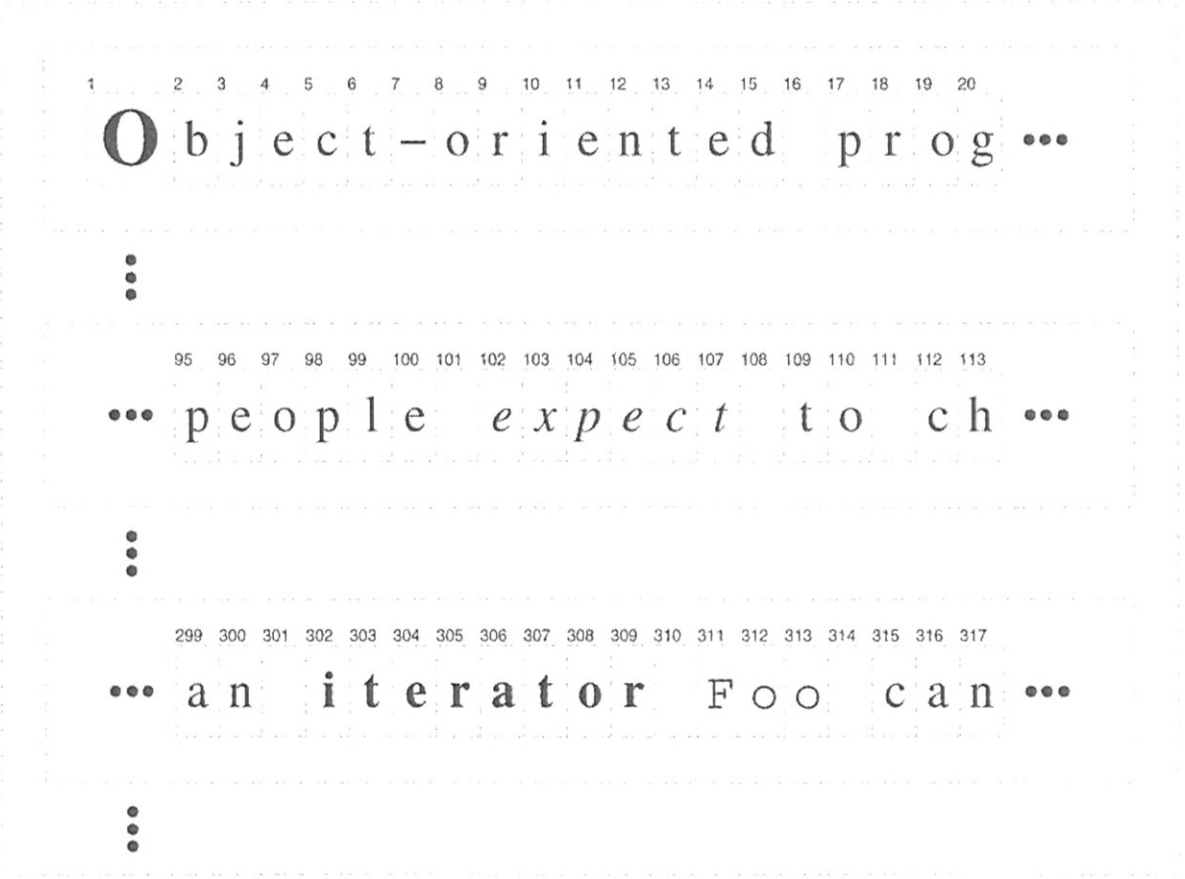
Ahhoz, hogy elkerülhessük, hogy minden egyes képjel betűtípus jellemzőjének helyet foglaljunk, a jellemzőt külsőleg, egy `GlyphContext` (KépjelKörnyezet) objektumban tároljuk. A `GlyphContext` a külső állapotinformációk raktáraként működik. Tömören egymáshoz rendeli a képjeleket és betűtípusukat (illetve bármilyen más lehetséges grafikai jellemzőjüket) a különböző környezetekben. Minden művelet, amelynek szüksége van arra, hogy ismerje a képjel betűtípusát egy adott környezetben, paraméterként egy `GlyphContext` példányt kap, amelytől elkéri a környezetben érvényes betűtípust. A környezet a képjelnek a képjelszerkezetben elfoglalt helyétől függ, ezért a `Glyph` gyermekbejáró és -kezelő műveleteinek minden használat után frissíteniük kell a `GlyphContext`-et.

```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```

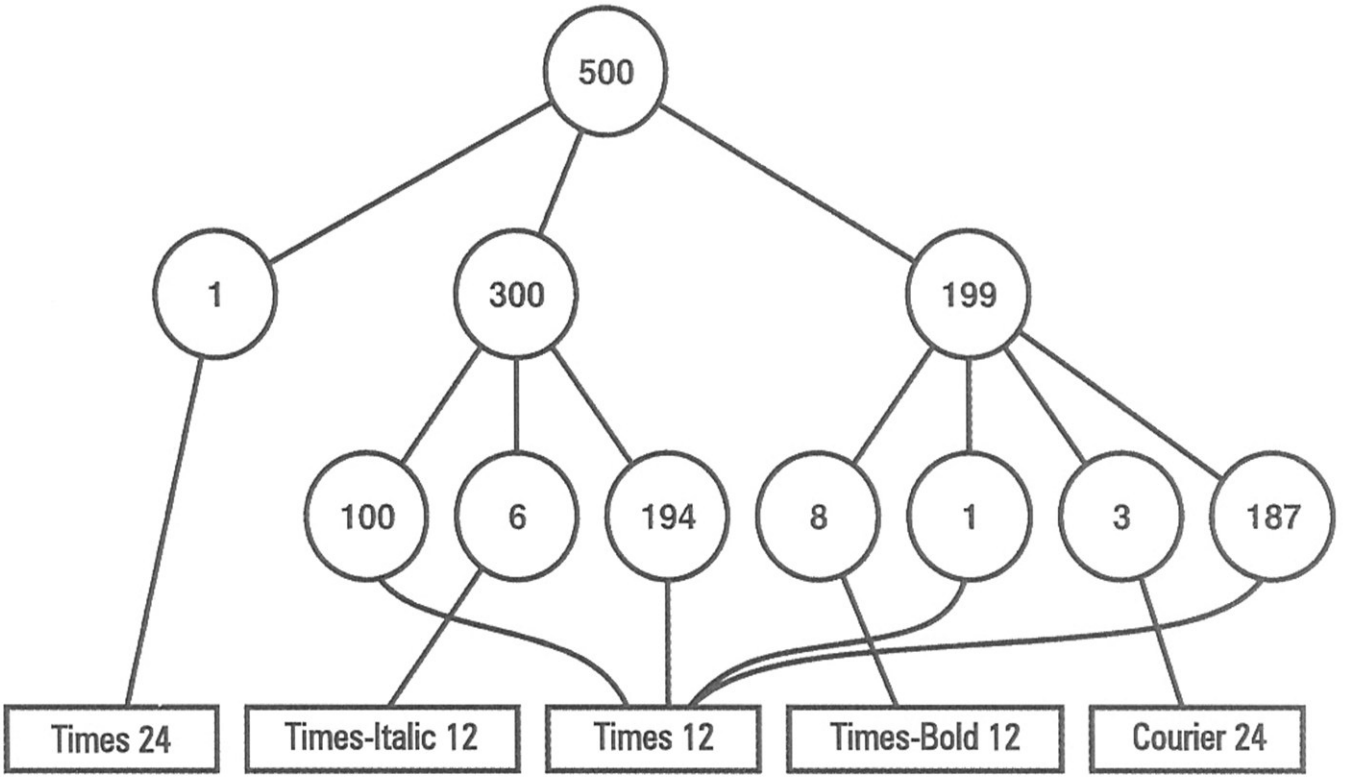
A `GlyphContext`-nek mindig tudnia kell, hol jár a képjelszerkezetben. Ahogy a bejárás halad, a `GlyphContext::Next` növeli az `_index`-et. A `Glyph` gyermekekkel rendelkező alosztályai (például a `Row` vagy a `Column`) úgy kell, hogy megvalósítsák a `Next` (Következő) műveletet, hogy az a bejárás minden pontján meghívja a `GlyphContext::Next`-et.

A `GlyphContext::GetFont` (KépjelKörnyezet::SzerezBetűtípus) az indexet egy `BTree` (B-fa) szerkezet kulcsaként használja, amely a képjel–betűtípus hozzárendeléseket tárolja. A fa minden csomópontját annak a karakterláncnak a hossza címkézi, amelyhez az adott csomópont betűtípus-információt ad. A fa levelei egy-egy betűtípusra mutatnak, míg a belső csomópontok részláncokra tördelik a karakterláncokat. Minden gyermekhez egy-egy részlánc tartozik.

Vegyük az alábbi részletet egy képjel-összetételből:



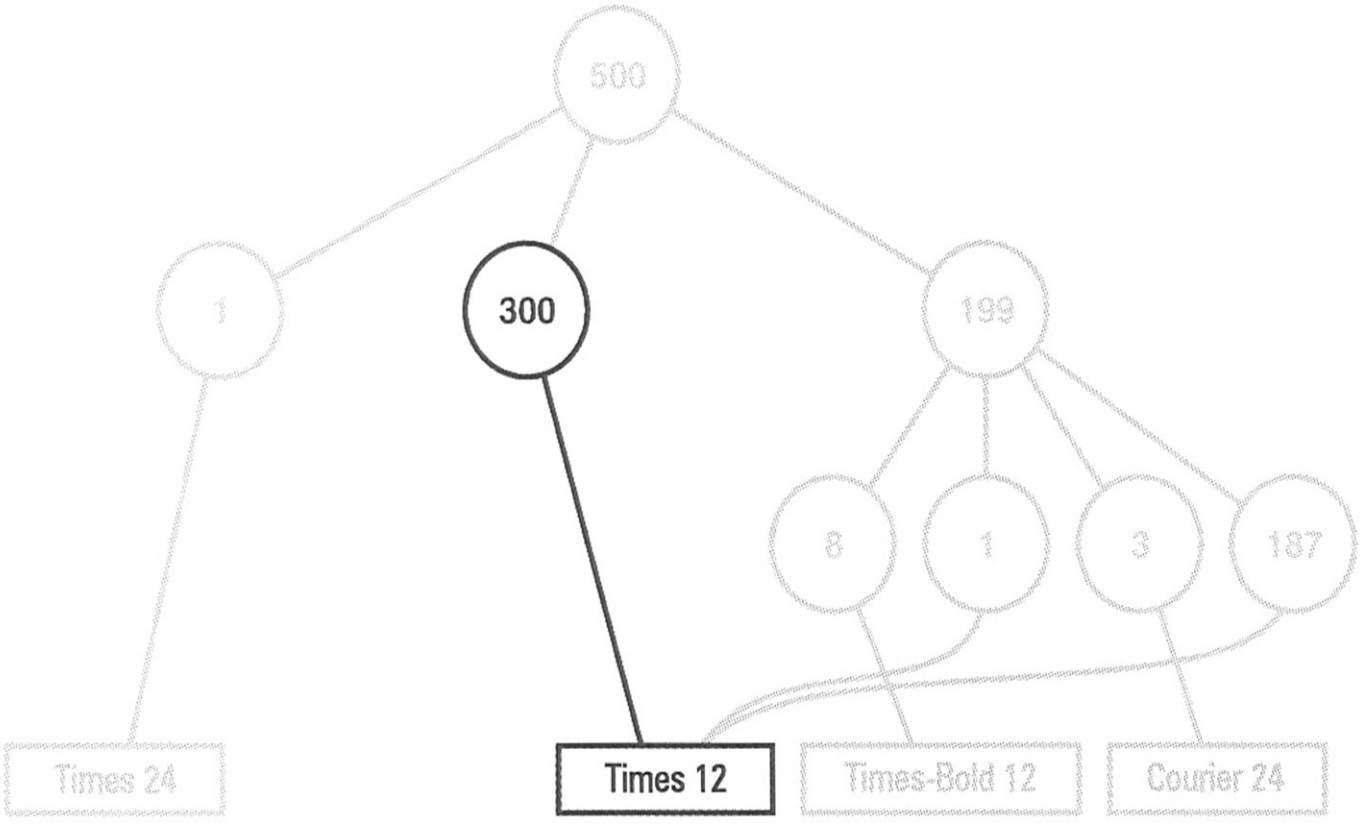
A `BTree` szerkezet valahogy így festhet:



A belső csomópontok képjel-indextartományokat határoznak meg. Ha a betűtípus megváltozik, a szerkezethez képjeleket adunk, vagy képjeleket veszünk el onnan, a `BTree` frissül. Tegyük fel például, hogy a bejárás során a 102-es sorszámnál (indexnél) tartunk. Ekkor az alábbi kód az „expect" szó minden karakterét a környező szöveg betűtípusára (`times12`, ami a `Font` példánya a 12 pontos Times Roman betűkhöz) állítja:

```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
//...

gc.SetFont(times12, 6);
```

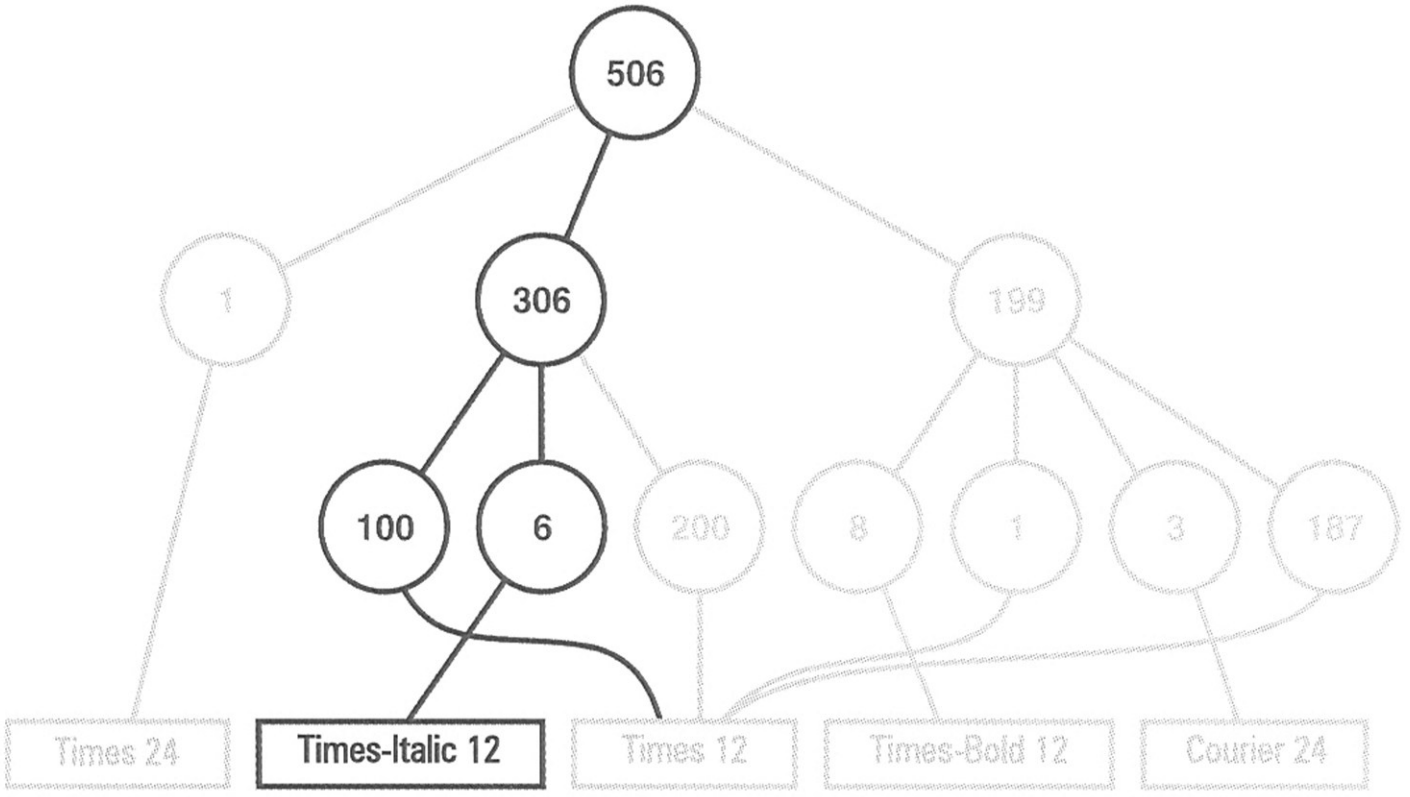
Az új `BTree` szerkezet a következőképpen néz ki (a változást kiemeltük):



Tegyük fel, hogy az „expect" elé beszúrjuk a „don't" szót (egy szóközzel együtt), 12 pontos Times-Italic (dőlt) betűvel. Az alábbi kód értesíti a `gc`-t az eseményről, feltételezve, hogy még mindig a 102-es indexen tartózkodik:

```
gc.Insert(6);
gc.SetFont(timesItalic12, 6);
```

A `BTree` szerkezet ekkor így módosul:



Amikor a `GlyphContext`-től elkérik az aktuális elem betűtípusát, végighalad a B-fán, amíg meg nem találja az aktuális indexnek megfelelő betűtípust. Miután a betűtípus-váltások száma viszonylag csekély, a fa a képjelszerkezet méretéhez képest kicsi marad, ami alacsonyan tartja a tárköltséget, anélkül, hogy aránytalanul nőne a keresési idő.[3]

Az utolsó objektum, amire szükségünk van, egy PehelysúlyúGyár, amely létrehozza a képjeleket és gondoskodik megfelelő megosztásukról. A `GlyphFactory` (KépjelGyár) osztály `Character` és más típusú képjeleket példányosít. Csak a `Character` objektumokat osztjuk meg; az összetett képjelekből jóval kevesebb van, és fontosabb állapotinformációik (például a gyermekekre vonatkozók) amúgy is belsők.

```
const int NCHARCODES = 128;

class GlyphFactory {
public:
   GlyphFactory();
   virtual ~GlyphFactory();

   virtual Character* CreateCharacter(char);
   virtual Row* CreateRow();
   virtual Column* CreateColumn();
   //...
```

[3] A keresési idő ebben a sémában a betűtípus-váltások gyakoriságával arányos. A teljesítmény akkor a legrosszabb, ha minden karakterre jut egy betűtípus-váltás, de ez a gyakorlatban szokatlan lenne.

```
private:
   Character* _character[NCHARCODES];
};
```

A `_character` tömb karakterkóddal indexelt `Character` képjeleket címző mutatókat tartalmaz. A tömb a konstruktorban nulla kezdőértéket kap.

```
GlyphFactory::GlyphFactory () {
   for (int = 0; i < NCHARCODES; ++i) {
         _character[i] = 0;
   }
}
```

A `CreateCharacter` (LétrehozKarakter) megkeres egy karaktert a tömbben levő képjelben, és visszaadja a megfelelő képjelet, ha létezik. Ha nem létezik, létrehozza, a tömbbe helyezi, majd ezután adja vissza:

```
Character* GlyphFactory::CreateCharacter (char c) {
   if (!_character[c]) {
         _character[c] = new Character(c);
   }

   return _character[c];
}
```

A többi művelet egyszerűen egy új objektumot példányosít minden híváskor, mivel a nem karakter képjeleket nem osztjuk meg:

```
Row* GlyphFactory::CreateRow () {
   return new Row;
}

Column* GlyphFactory::CreateColumn () {
   return new Column;
}
```

Ki is hagyhatnánk ezeket a műveleteket, és a meg nem osztott képjelek példányosítását közvetlenül az ügyfélre bízhatnánk, de ha később úgy döntünk, hogy mégis megoszthatóvá tesszük őket, módosítanunk kell létrehozó kódjukat az ügyfélben.

Ismert felhasználások

A pehelysúlyú objektumok ötletét először az InterViews 3.0 [CL90] írta le és aknázta ki tervezési módszerként. Fejlesztői bizonyítékként egy erőteljes szövegszerkesztőt (Doc) is elkészítettek [CL92]. A Doc a dokumentumok valamennyi karakterét képjel (Glyph) objektu-

mokkal ábrázolja. A szerkesztő minden karakter számára, amelyet egy adott (a grafikai jellemzőket meghatározó) stílussal írtak, külön Glyph példányt épít, így a karakterek belső állapota a karakterkódból és a stílusinformációból (egy stílustáblázat adott indexéből) áll.[4] Ez azt jelenti, hogy csak a karakter helye külső információ, aminek révén a Doc működése gyors. A dokumentumokat a Document (Dokumentum) osztály jelképezi, amely egyben a PehelysúlyúGyár szerepét is betölti. A Doc vizsgálata azt mutatta, hogy a pehelysúlyú karakterek megosztása igen hatékony: egy átlagos, 180 000 karakterből álló dokumentum mindössze 480 karakter objektum számára igényel tárhelyet.

Az ET++ [WGM88] a megjelenítési szabványoktól való függetlenséget támogatja pehelysúlyú objektumokkal.[5] A megjelenítési szabványok a felhasználói felület elemeinek (gördítősávok, gombok, menük – egységes nevükön vezérlők) megjelenését, illetve azok (árnyékolással, térhatással való) díszítését szabályozzák. A vezérlők ezeket a feladatokat (elrendezés, kirajzolás) egy önálló Layout (Elrendezés) objektumra ruházzák át, amelynek megváltoztatásával akár futásidőben módosítható a megjelenítési mód.

Minden vezérlőosztályhoz egy-egy Layout osztály tartozik (ScrollbarLayout, MenubarLayout stb.). A megoldással nyilvánvalóan az a gond, hogy a felhasználói felület objektumainak száma megkétszereződik. Ezt a terhelést elkerülendő a Layout objektumok pehelysúlyúak. Ez természetesen adódik, hiszen ezen objektumok feladata többnyire valamilyen viselkedés meghatározása, az elrendezéshez és rajzoláshoz szükséges csekély mennyiségű külső állapotinformáció pedig könnyen átadható nekik.

A Layout objektumokat Look (Kinézet) objektumok hozzák létre és kezelik. A Look osztály egy elvont gyár, amely a megfelelő Layout objektumot olyan műveletekkel állítja elő, mint a GetButtonLayout (SzerezGombElrendezés), GetMenuBarLayout (SzerezMenüSorElrendezés) és így tovább. Minden megjelenítési szabványhoz tartozik egy Look alosztály (MotifLook, OpenLook stb.), amelyek a megfelelő Layout objektumokat biztosítják.

A Layout objektumok egyébiránt lényegüket tekintve stratégiák (lásd a Stratégia mintát), vagyis olyan stratégia objektumok, amelyeket pehelysúlyúként valósítottak meg.

Kapcsolódó minták

A Pehelysúlyú mintát gyakran használják együtt az Összetétel mintával, hogy egy megosztott levél-csomópontokkal rendelkező irányított körmentes gráf formájában valósítsanak meg egy hierarchikus logikai szerkezetet.

Az Állapot és a Stratégia minta objektumait általában pehelysúlyúként a legjobb megvalósítani.

[4] A Példakód részben szereplő kódban a stílusinformáció külső, csak a karakterkód belső állapotinformáció.

[5] A megjelenítési szabványoktól való függetlenség biztosítására egy másik megközelítést az Elvont Gyár mintánál találhatunk.

Helyettes

Szerkezeti objektumminta

Cél

Adott objektumot képviselőn vagy helyőrzőn keresztül irányítani, hogy szorosabban felügyelhessük működését.

Egyéb nevek

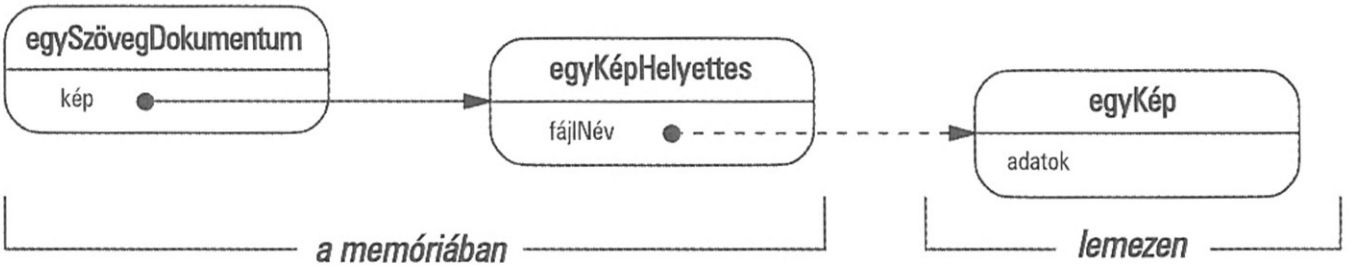
Proxy, Surrogate (Helyettesítő)

Feladat

Az objektumokhoz való hozzáférés szabályozásának egyik célja, hogy létrehozásuk és előkészítésük teljes költségének „megfizetését" elhalasszuk addig, amíg ténylegesen sor nem kerül használatukra. Vegyünk egy szövegszerkesztő programot, amely képes grafikus objektumokat beágyazni egy dokumentumba. Egyes grafikus objektumok – például a nagyméretű raszterképek – létrehozása igen költséges lehet. A dokumentum megnyitásának azonban gyorsnak kell lennie, ezért el kell kerülnünk azt, hogy a költséges objektumokat a dokumentum megnyitásakor, egyszerre hozzuk létre. Erre egyébként sincs szükség, hiszen egyidejűleg nem mindegyik objektum látható a dokumentumban.

Mindezekből az következik, hogy a költséges objektumokat *igény szerint* kell létrehoznunk, vagyis ebben az esetben akkor, amikor az adott kép láthatóvá válik. De addig mit tegyünk a helyére a dokumentumban, és hogyan rejtsük el azt a tényt, hogy a kép igény szerint jön létre, anélkül, hogy túlbonyolítanánk a szerkesztő megvalósítását? Csak hogy egy példát említsünk, a létrehozás optimalizálásának nem szabad hatással lennie a megjelenítési és formázási kódra.

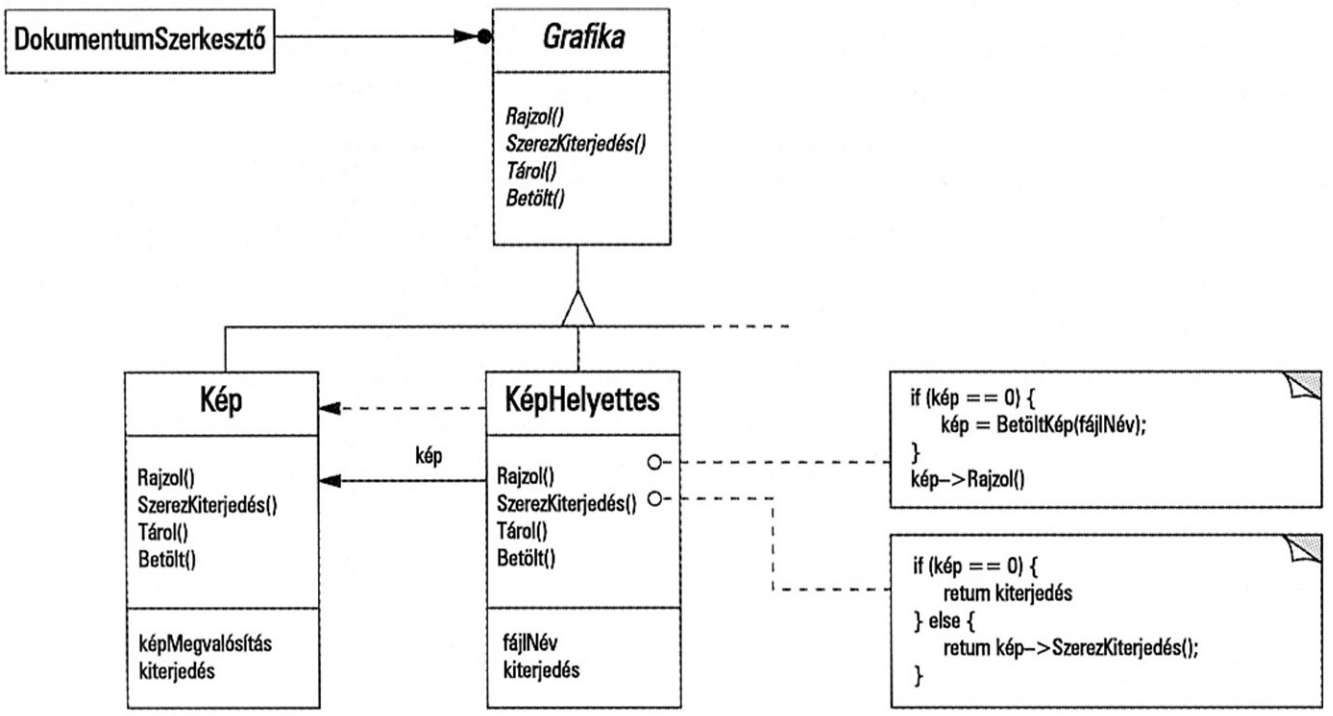
A megoldás egy új objektum, a **képhelyettes** használata, amely a tényleges kép helyén áll. A helyettes ugyanúgy viselkedik, mint maga a kép, és gondoskodik annak példányosításáról, ha szükség van rá.



A képhelyettes csak akkor hozza létre a képet, amikor a szövegszerkesztő Rajzol (Draw) műveletének meghívásával önmaga kirajzolására szólítja fel. A helyettes ezután közvetlenül a képhez továbbítja a kérelmeket, ezért hivatkoznia kell a képre, miután létrehozta.

Tegyük fel, hogy a képek önálló fájlokban tárolódnak. Ebben az esetben a tényleges objektumra a fájlnévvel hivatkozhatunk. A helyettes emellett a kép **kiterjedését** (extent), vagyis a szélességét és magasságát is tárolja. A kiterjedés tárolása lehetővé teszi a helyettes számára, hogy kezelje a formázótól a méret beállítására vonatkozó kérelmeket, anélkül, hogy valóban példányosítaná a képet.

Az alábbi osztálydiagram részletesebben illusztrálja a fenti példát:



A dokumentumszerkesztő a beágyazott képeket az elvont Grafika (Graphic) osztály által meghatározott felületen keresztül éri el. Az igény szerint létrehozott képek osztálya a KépHelyettes (ImageProxy). A KépHelyettes a lemezen található képre a fájlnévvel hivatkozik. A fájlnevet argumentumként adjuk át a KépHelyettes konstruktorának.

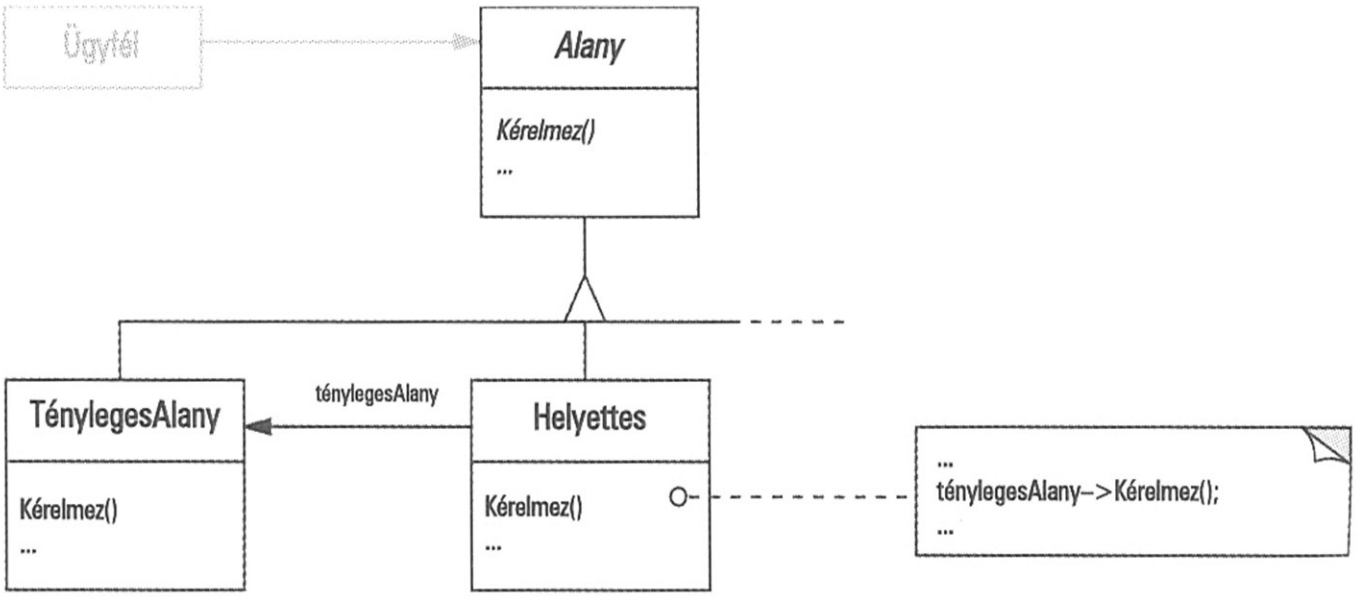
A KépHelyettes a kép befoglaló dobozát is tárolja, illetve egy hivatkozást a tényleges Kép (Image) példányra. A hivatkozás mindaddig nem érvényes, amíg a helyettes nem példányosítja a képet. A Rajzol (Draw) művelet gondoskodik róla, hogy a kép példányosítására sor kerüljön, mielőtt a kérelmek továbbítódnának hozzá. A SzerezKiterjedés (GetExtent) csak akkor továbbít kérelmet a képnek, ha már létezik belőle példány, egyébként a KépHelyettes a tárolt kiterjedést adja vissza.

Alkalmazhatóság

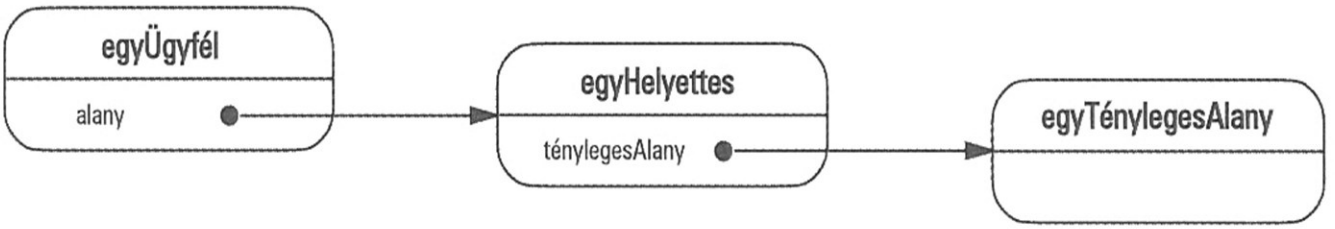
A Helyettes tervezési minta akkor használható, amikor egy egyszerű mutatónál rugalmasabb vagy kifinomultabb hivatkozás szükséges egy objektumhoz. A mintát többek között a következő helyzetekben célszerű alkalmazni:

1. **Távoli helyettes** – helyi képviselőt biztosít egy másik címtérben található objektum számára. A NEXTSTEP [Add94] erre a célra az NXProxy osztályt használja. Coplien [Cop92] az ilyen helyettest „nagykövetnek" (Ambassador) nevezi.
2. **Virtuális helyettes** – igény szerint létrehozza a költséges objektumokat. A Feladat részben leírt KépHelyettes is ilyen helyettes.
3. **Védelmi helyettes** – szabályozza a hozzáférést az eredeti objektumhoz. A védelmi helyettesek akkor hasznosak, ha egyes objektumoknak különböző hozzáférési jogosultságokkal kell rendelkezniük. A Choices operációs rendszerben [CIRM93] található KernelProxy-k (RendszermagHelyettes-ek) például védett hozzáférést biztosítanak az operációs rendszer objektumai számára.
4. **Okos helyettes** – olyan, mint egy sima mutató, csak az objektum elérésekor további műveleteket végez. Használatára néhány jellemző példa:
   - a tényleges objektumra való hivatkozások számlálása, hogy az objektum helye automatikusan felszabadítható legyen, ha már nem hivatkozik rá semmi (**okos mutatónak** is hívják [Ede92]);
   - egy maradandó objektum betöltése a memóriába az első rá vonatkozó hivatkozásnál;
   - annak ellenőrzése, hogy hozzáférés előtt sor került-e a tényleges objektum zárolására, amivel megakadályozható, hogy közben más objektum módosítsa.

Szerkezet



Íme egy helyettes-szerkezet lehetséges objektumdiagramja futásidőben:



Résztvevők

- **Helyettes** (KépHelyettes)
  - Hivatkozást tart fenn, amelynek segítségével a helyettes elérheti a tényleges alanyt. A helyettes egy Alanyra (Subject) is hivatkozhat, ha a TénylegesAlany (RealSubject) és az Alany felülete azonos.
  - Az Alany felületével azonos felületet biztosít, hogy a helyettes kicserélhető legyen a tényleges alannyal.
  - Szabályozza a hozzáférést a tényleges alanyhoz, és felelhet annak létrehozásáért és törléséért is.
  - Az egyéb felelősségek a helyettes típusától függnek:
    - » A *távoli helyettesek* feladata a kérelmek és argumentumaik kódolása, illetve a kódolt kérelem elküldése a másik címtérben található tényleges alanynak.
    - » A *virtuális helyettesek* ideiglenesen kiegészítő információkat tárolhatnak a tényleges alanyról, hogy elérését elhalaszthassák. A Feladat részben szereplő KépHelyettes például a kép kiterjedését tárolja.
    - » A *védelmi helyettesek* ellenőrzik, hogy a hívó rendelkezik-e a kérelem teljesítéséhez szükséges hozzáférési engedéllyel.
- **Alany** (Grafika)
  - Meghatározza a TénylegesAlany és a Helyettes (Proxy) közös felületét, hogy a Helyettes mindazokon a helyeken használható legyen, ahol TénylegesAlanyra van szükség.
- **TénylegesAlany** (Kép)
  - Meghatározza a helyettes által képviselt tényleges objektumot.

Együttműködés

- A Helyettes szükség esetén kérelmeket továbbít a TénylegesAlanynak, a helyettes típusától függően.

Következmények

A Helyettes minta közvetettséget biztosít az objektumok elérésében. Ennek a közvetettségnek a helyettes típusától függően számos előnye van:

1. A távoli helyettes elrejti, hogy az objektum egy másik címtérben található.
2. A virtuális helyettes optimalizálja a viselkedést, például azzal, hogy igény szerint hozza létre az objektumokat.
3. A védelmi és okos helyettesek további műveleteket tesznek lehetővé az objektumok elérésekor.

A Helyettes mintával emellett még valamit elrejthetünk az ügyfelek elől, mégpedig a „**másolás íráskor**" (copy-on-write) megoldást, ami az igény szerinti létrehozáshoz kapcsolódik. A nagy és bonyolult objektumok másolása költséges művelet lehet, pedig e költség szükségtelen, ha a másolat soha nem módosul. Ha helyettes használatával elhalasztjuk a másolást, gondoskodhatunk róla, hogy a másolás költségét csak akkor fizetjük meg, ha az objektumot módosítjuk.

Ahhoz, hogy a másolás íráskor működjön, az alanyra vonatkozó hivatkozások számlálására van szükség. A helyettes másolása csupán a hivatkozásszámláló értékét növeli. Az alany tényleges másolását csak akkor hajtja végre a helyettes, ha az ügyfél olyan műveletet kérelmez, ami módosítja azt. Ekkor a helyettesnek csökkentenie is kell a hivatkozásszámláló értékét. Amikor az érték nullára csökken, az alany törlődik.

A másolás íráskor jelentősen csökkenti a nehézsúlyú alanyok másolásának költségét.

Megvalósítás

A Helyettes minta révén a következő nyelvi szolgáltatások aknázhatók ki:

1. *A C++ tagelérő műveletének túlterhelése.* A C++ támogatja a tagok elérésére szolgáló `operator->` művelet túlterhelését. E megoldás révén további műveleteket végezhetünk, amikor egy objektum hivatkozását feloldjuk (dereferencia), ez pedig segítségünkre lehet egyes helyettestípusok megvalósításában; a helyettes ugyanúgy viselkedik, mint egy mutató.
   A következő példa azt mutatja be, hogyan használható a fenti eljárás egy `ImagePtr` (KépMutató) nevű virtuális helyettes megvalósítására.

```
class Image;
extern Image* LoadAnImageFile(const char*);
        // külső függvény

class ImagePtr {
public:
        ImagePtr(const char* imageFile);
        virtual ~ImagePtr();
```

```
        virtual Image* operator->();
        virtual Image& operator*();
private:
        Image* LoadImage();
private:
        Image* _image;
        const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
        _imageFile = theImageFile;
        _image = 0;
}

Image* ImagePtr::LoadImage () {
        if (_image == 0) {
                image = LoadAnImageFile(_imageFile);
        }
        return _image;
}
```

A túlterhelt -> és * műveletek (operátorok) a `LoadImage` (BetöltKép) segítségével adják vissza az `_image`-et a hívóknak (illetve töltik be, ha szükséges).

```
Image* ImagePtr::operator-> () {
        return LoadImage();
}

Image& ImagePtr::operator* () {
        return *LoadImage();
}
```

E megközelítés révén anélkül hívhatjuk meg az `Image` műveleteit `ImagePtr` objektumokon keresztül, hogy a műveleteket az `ImagePtr` felület részévé kellene tennünk:

```
ImagePtr image = ImagePtr("egyKépFájlNév");
image->Draw(Point(50, 100));
        // (image.operator->())->Draw(Point(50, 100))
```

Megfigyelhetjük, hogy az `image` helyettes mutatóként viselkedik, pedig nem egy `Image`-re hivatkozó mutatóként vezetjük be. Ebből következően nem használható pontosan úgy, mint egy valódi `Image` mutató, vagyis az ügyfeleknek különbözőképpen kell kezelniük az `Image` és `ImagePtr` objektumokat.

A tagelérő művelet túlterhelése nem felel meg mindenféle helyettesnél. Egyes helyetteseknek pontosan tudniuk kell, *melyik* művelet hívására került sor, így az ilyen túlterhelés ebben az esetben nem működik.

Vegyük a Feladat részben szereplő virtuális helyettes példáját. A képet egy adott időpontban (a Draw művelet meghívásakor), és nem a képre való minden hivatkozásnál kell betölteni. A tagelérő művelet túlterhelése ezt a megkülönböztetést nem teszi lehetővé. Tehát itt egyenként, magunknak kell megvalósítanunk a helyettes minden olyan műveletét, ami kérelmet továbbít az alanynak.

Az említett műveletek általában nagyon hasonlítanak egymásra, amint azt majd a Példakód részben is láthatjuk. Jellemzően mindegyik művelet ellenőrzi, hogy a kérelem érvényes-e, hogy az eredeti objektum létezik-e, és így tovább, mielőtt a kérelmet az alanyhoz továbbítaná. Ezt a kódot újra és újra megírni meglehetősen fárasztó, ezért többnyire előfeldolgozó segítségével automatikusan állítják elő.

2. *A doesNotUnderstand használata a Smalltalk-ban.* A Smalltalk egy horgot biztosít, amellyel támogathatjuk a kérelmek automatikus továbbítását. Amikor egy ügyfél üzenetet küld egy olyan fogadónak, amelynek nincs megfelelő metódusa, a nyelvben a `doesNotUndestand: aMessage` hívására kerül sor. A Helyettes (Proxy) osztály úgy írhatja felül a `doesNotUnderstand`-et, hogy az üzenet továbbítódjon az alanyhoz.

   Ahhoz, hogy biztosítsuk, a kérelem továbbítódik az alanyhoz, nem pedig csendesen elnyeli a helyettes, meghatározhatunk egy olyan Helyettes osztályt, amely *egyetlen* üzenetet sem ért meg. A Smalltalk ezt úgy támogatja, hogy a Helyettest ősosztály nélküli osztályként határozza meg.[6]

   A `doesNotUnderstand:` legnagyobb hátránya az, hogy a legtöbb Smalltalk rendszer néhány olyan különleges üzenettel rendelkezik, amelyeket közvetlenül a virtuális gép kezel, így nem kerül sor a szokásos metódus-kikeresésre. Az egyetlen, amelyet általában Object-ben valósítanak meg (és így érintheti a helyetteseket) az azonosságvizsgáló == művelet.

   Ha a helyettes megvalósítására a `doesNotUnderstand:`-et használjuk, meg kell kerülnünk ezt a problémát, hiszen a helyettesek azonossága nem jelenti a tényleges alanyok azonosságát. Emellett az is hátrány, hogy a `doesNotUnderstand:`-et hibakezelésre tervezték, nem helyettesek építésére, ezért általában elég lassú.

3. *A helyettesnek nem mindig kell ismernie a tényleges alany típusát.* Ha egy Helyettes osztály kizárólag egy elvont felületen keresztül léphet kapcsolatba az alannyal, nincs szükség rá, hogy minden TénylegesAlany osztályhoz külön Helyettes osztályt készítsünk – a helyettes minden TénylegesAlany osztályt egységesen kezelhet. Ha azonban a Helyettesek példányosítják a TénylegesAlanyokat (mint ahogy a virtuális helyettesek teszik), ismerniük kell a konkrét osztályt.

---

[6] A NEXTSTEP [Add94] elosztott objektumainak megvalósítása (pontosabban az NXProxy osztály) is ezt a megoldást alkalmazza. Itt a `forward` (továbbít) felülírására kerül sor, ami a NEXTSTEP hasonló célú horga.

A megvalósítással kapcsolatban szót kell ejtenünk arról is, hogyan hivatkozhatunk az alanyra annak példányosítása előtt. Egyes helyetteseknek attól függetlenül kell hivatkozniuk a hozzájuk tartozó alanyra, hogy az a memóriában vagy a lemezen található-e, vagyis valamilyen címtérfüggetlen objektumazonosítót kell használniuk. A Feladat részben erre a célra a fájlnevet alkalmaztuk.

Példakód

A következő kódban kétféle helyettest valósítunk meg: a Feladat részben leírt virtuális helyettest, és egy olyan helyettest, ami a `doesNotUnderstand:` horgot használja.[7]

1. *Virtuális helyettes.* A `Graphic` (Grafika) osztály határozza meg a grafikus osztályok felületét:

```
class Graphic {
public:
        virtual ~Graphic();

        virtual void Draw(const Point& at) = 0;
        virtual void HandleMouse(Event& event) = 0;

        virtual const Point& GetExtent() = 0;

        virtual void Load(istream& from) = 0;
        virtual void Save(ostream& to) = 0;
protected:
        Graphic();
};
```

   Az `Image` (Kép) osztály a képfájlok megjelenítéséhez valósítja meg a `Graphic` felületet. Az `Image` felülbírálja a `HandleMouse` (KezelEgér) műveletet, hogy a felhasználók interaktívan átméretezhessék a képet.

```
class Image : public Graphic {
public:
        Image(const char* file);   //képet tölt be egy fájlból
        virtual ~Image();

        virtual void Draw(const Point& at);
        virtual void HandleMouse(Event& event);

        virtual const Point& GetExtent();

        virtual void Load(istream& from);
        virtual void Save(ostream& to);
private:
        //...
};
```

[7] A Bejáró mintánál a következő fejezetben egy másfajta helyettest is bemutatunk.

Az ImageProxy (KépHelyettes) felülete megegyezik az Image-ével:

```
class ImageProxy : public Graphic {
public:
        ImageProxy(const char* imageFile);
        virtual ~ImageProxy();

        virtual void Draw(const Point& at);
        virtual void HandleMouse(Event& event);

        virtual const Point& GetExtent();

        virtual void Load(istream& from);
        virtual void Save(ostream& to);
protected:
        Image* GetImage();
private:
        Image* _image;
        Point _extent;
        char* _fileName;
};
```

A konstruktor helyi másolatot ment a képet tároló fájl nevéről, és kezdőértéket ad az _extent (_kiterjedés) és _image (_kép) tagoknak:

```
ImageProxy::ImageProxy (const char* fileName) {
        _fileName = strdup(fileName);
        _extent = Point::Zero; //a kiterjedést még nem ismerjük
        _image = 0;
}

Image* ImageProxy::GetImage() {
        if (_image == 0) {
                _image = new Image(fileName);
        }
        return _image;
}
```

A GetExtent (SzerezKiterjedés) megvalósítása, ha lehetséges, az ideiglenesen tárolt kiterjedést adja vissza, egyébként a kép betöltődik a fájlból. A Draw (Rajzol) tölti be a képet, a HandleMouse pedig a tényleges képhez továbbítja az eseményt.

```
const Point& ImageProxy::GetExtent () {
        if (_extent == Point::Zero) {
                _extent = GetImage()->GetExtent();
        }
        return _extent;
}

void ImageProxy::Draw (const Point& at) {
        GetImage()->Draw(at);
}
```

```
void ImageProxy::HandleMouse (Event& event) {
        GetImage()->HandleMouse(event);
}
```

A `Save` (Ment) művelet egy folyamba menti a tárolt képkiterjedést és fájlnevet. A `Load` (Betölt) elkéri ezeket a információkat, és előkészíti a megfelelő tagokat.

```
void ImageProxy::Save (ostream& to) {
        to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
        from >> _extent >> _fileName;
}
```

Végül, tegyük fel, hogy van egy `TextDocument` (SzövegDokumentum) nevű osztályunk, amely tartalmazhat `Graphic` objektumokat:

```
class TextDocument {
public:
        TextDocument();

        void Insert(Graphic*);
        //...
};
```

Egy `ImageProxy`-t így szúrhatunk be egy szövegdokumentumba:

```
TextDocument* text = new TextDocument;
//...
text->Insert(new ImageProxy("egyKépFájlNév"));
```

2. *Helyettesek, amelyek a doesNotUnderstand-et használják.* A Smalltalkban olyan osztályok meghatározásával készíthetünk általános helyetteseket, amelyek ősosztálya a nil[8], valamint a `doesNotUnderstand:` metódus meghatározásával képes az üzenetek kezelésére.
   Az alábbi metódus feltételezi, hogy a helyettesnek van egy `realSubject` metódusa, ami a tényleges alanyt adja vissza. Az `ImageProxy` esetében ez a metódus ellenőrizné, hogy létrejött-e az `Image`, szükség esetén létrehozná, és végül visszaadná. A metódus a `perform:withArguments:` segítségével juttatja célba az üzenetet.

```
doesNotUnderstand: aMessage
        ^ self realSubject
                perform: aMessage selector
                withArguments: aMessage arguments
```

   A `doesNotUnderstand:` argumentuma a `Message` (Üzenet) egy példánya, ami a helyettes által nem értelmezhető üzenetet jelképezi. Így a helyettes minden üzenetre úgy válaszol, hogy ellenőrzi, létezik-e az alany, mielőtt az üzenetet továbbítaná hozzá.

[8] Szinte minden osztály végső ősosztálya az Object, ezért a kifejezés egyenértékű azzal, mintha azt mondanánk: „olyan osztályok meghatározásával, amelyeknek az Object nem őse".

A `doesNotUnderstand:` egyik előnye, hogy különféle feldolgozó műveleteket végezhet. Például egy védelmi helyettest hozhatunk létre, ha meghatározzuk az elfogadható üzenetek halmazát (`legalMessages`), majd az alábbi metódussal látjuk el a helyettest:

```
doesNotUnderstand: aMessage
        ^ (legalMessages includes: aMessage selector)
              ifTrue: [self realSubject
                    perform: aMessage selector
                    withArguments: aMessage arguments]
              ifFalse: [self error: 'Érvénytelen művelet']
```

A metódus ellenőrzi, hogy az üzenet érvényes-e, mielőtt továbbítaná azt a tényleges alanyhoz. Ha az üzenet nem elfogadható, `error:` üzenetet küld a helyettesnek, ami a hibák végtelen ciklusát indíthatja el, hacsak a helyettes meg nem határozza az `error:`-t. Az `error:` meghatározását az esetleges metódusokkal együtt az Object osztályból kell ide másolni.

Ismert felhasználások

A Feladat részben szereplő virtuális helyettes az ET++ szövegépítő-blokk osztályaiból származik.

A NEXTSTEP [Add94] a megosztható objektumok helyi képviselőiként használ helyetteseket (az NXProxy osztály példányait). A kiszolgálók helyetteseket hoznak létre a távoli objektumok számára, amikor az ügyfelek igénylik azokat. Amikor üzenet érkezik, a helyettes argumentumaival együtt kódolja, majd a kódolt üzenetet továbbítja a távoli alanyhoz. Az alany ugyanígy kódolja az esetleges eredményeket, és visszaküldi az NXProxy objektumnak.

McCullogh [McC87] a távoli objektumok helyettesekkel való elérését tárgyalja a Smalltalkban. Pascoe [Pas86] azt írja le, hogyan adhatunk „mellékhatásokat" a metódushívásokhoz és a hozzáférés-szabályozáshoz úgynevezett „egységbe zárókkal" (Encapsulator).

Kapcsolódó minták

Illesztő: Az illesztők más felületet biztosítanak az illesztett objektumok számára, míg a helyettesek felülete ugyanaz, mint alanyaiké. Mindazonáltal az elérés védelmére használt helyettesek visszautasíthatnak egy olyan műveletet, amit az alany végrehajtana, így felületük az alany felületének részhalmaza is lehet.

Díszítő: Bár a díszítők megvalósítása hasonló lehet a helyettesekéhez, a díszítők célja más. A díszítők új felelősségeket adnak egy objektumhoz, míg a helyettesek az objektumhoz való hozzáférést szabályozzák.

A helyettesek megvalósításának hasonlósága a díszítőkéhez különböző lehet. Egy védelmi helyettes pontosan ugyanúgy is megvalósítható, mint egy díszítő, míg egy távoli helyettes nem hivatkozik közvetlenül az alanyára, csak olyan közvetett hivatkozásokon keresztül, mint amilyen a számítógép azonosítója és a célgépen érvényes helyi cím. A virtuális helyettesek kezdetben a fájlnévhez hasonló közvetlen hivatkozásokat alkalmaznak, de végül egy közvetett hivatkozást szereznek meg és használnak.

A szerkezeti mintákról

Olvasás közben bizonyára észrevettük a hasonlóságokat a szerkezeti minták között, különösen résztvevőiket és azok együttműködését illetően. A hasonlóságok valószínűleg abból erednek, hogy mindegyik minta a kód és az objektumok szervezésére szolgáló nyelvi szolgáltatások azonos részhalmazára támaszkodik: az osztály alapú minták esetében egyszeres és többszörös öröklést, az objektummintáknál objektum-összetételt használunk. A hasonlóságok azonban elfedik az egyes minták céljai közti különbségeket, ezért ebben a részben összehasonlítjuk a szerkezeti minták egyes csoportjait, hogy előnyeikről tisztább képet kapjunk.

Illesztő vagy Híd?

Az Illesztő és a Híd minta osztozik néhány közös tulajdonságon: mindkettő a rugalmasságot támogatja azáltal, hogy közvetett elérést nyújt egy objektumhoz, illetve mindkettőben szerepel az objektumétól eltérő felületen keresztüli kérelemtovábbítás.

A két minta közti legfontosabb különbséget céljuk jelenti. Az Illesztő minta két meglevő, eltérő felület összeegyeztetésére összpontosít. A felületek megvalósításával, vagy egymástól független fejlesztésével nem foglalkozik, vagyis arra szolgál, hogy két önállóan tervezett osztály együttműködését biztosítsa, anélkül, hogy bármelyik megvalósítását át kellene dolgoznunk. A Híd minta ezzel szemben egy elvont fogalmat és annak (esetleg számos) megvalósítását köti össze. A megvalósító osztályok cserélgethetők, mégis stabil felületet nyújt az ügyfeleknek, ezenkívül könnyen adhatók hozzá új megvalósítások a rendszer fejlesztése során.

Az említett különbségek eredményeképpen az Illesztő és a Híd mintát gyakran a szoftver életciklusának különböző pontjain alkalmazzák. Az illesztőkre általában akkor mutatkozik igény, amikor felfedezzük, hogy két össze nem egyeztethető osztálynak együtt kellene működnie (általában a kód megkettőzésének elkerülése végett), és ez a csatolás előre nem látható. A hidak alkalmazásánál viszont előre tudjuk, hogy az adott elvont fogalomnak számos megvalósítása kell legyen, és a kettőnek egymástól függetlenül fejleszthetőnek kell lennie. Az Illesztő minta tervezés *után* biztosítja az együttműködést, míg a Híd minta a tervezés *előtt*. Ez nem jelenti azt, hogy az Illesztő alacsonyabb rendű a Hídnál, csupán azt mutatja, hogy céljuk különböző.

A homlokzatokra (lásd a Homlokzat mintát) gondolhatunk úgy is, mint objektumok halmazának illesztőire, de ez az értelmezés figyelmen kívül hagyja azt a tényt, hogy a homlokzatok *új* felületet határoznak meg, míg az illesztők egy régi felület újrahasznosításával működnek. Vessük jól észbe, hogy az illesztők mindig két *létező* felület együttműködését biztosítják, nem pedig újat hoznak létre.

Összetétel, Díszítő vagy Helyettes?

Az Összetétel és a Díszítő minta szerkezeti diagramja hasonló, ami azt tükrözi, hogy mindkettő önhívó összetételre épül, és így rendez el korlátlan számú objektumot. Ez a hasonlóság azt sugallhatja számunkra, hogy egy díszítő objektum nem más, mint egy csökevényes összetétel, de ennek semmi köze a Díszítő minta lényegéhez. A hasonlóság valójában megáll az önhívó összetételnél – a két minta célja ismét csak egészen más.

A Díszítőt arra tervezték, hogy alosztályok származtatása nélkül adhassunk felelősségeket objektumokhoz. Így elkerülhető az alosztályok számának robbanásszerű növekedése, ami bekövetkezne, ha statikusan próbálnánk a felelősségek minden lehetséges kombinációját lefedni. Az Összetétel minta más célt szolgál: az osztályok szervezésére összpontosít, hogy rokon objektumokat egységesen, illetve több objektumot egyként kezelhessünk. Középpontjában tehát nem a díszítés, hanem az ábrázolás áll.

Az említett célok különböznek, de kiegészítik egymást, amiből az is következik, hogy a két mintát gyakran együtt használják. Mindkettő olyan felépítést eredményez, melynek révén egyszerűen objektumok egymáshoz csatlakoztatásával, új osztályok meghatározása nélkül építhetünk programokat. Lesz egy elvont osztályunk, amelynek egyes alosztályai összetételek, míg mások díszítők lesznek, más alosztályok pedig a rendszer alapvető építőköveit valósítják meg. Ebben az esetben a díszítők és összetételek közös felületet kapnak. A Díszítő minta szempontjából az összetételek a KonkrétElem szerepét töltik be, míg az Összetétel minta felől tekintve a díszítők Levelek. A két mintát természetesen nem *muszáj* együtt alkalmazni, céljuk pedig – mint láttuk – teljesen különböző.

Egy másik tervezési minta, amelynek szerkezete hasonló a Díszítőéhez, a Helyettes. Mindkét minta azt írja le, hogyan biztosíthatunk közvetett elérést egy objektumhoz, és mind a helyettes, mind a díszítő objektum megvalósítása egy másik objektumra hivatkozik, amelyhez kérelmeket továbbítanak. A cél azonban ebben az esetben is különböző.

A Díszítőhöz hasonlóan a Helyettes minta is összeállít egy objektumot, és azonos felületet nyújt az ügyfeleknek. Abban viszont már különbözik, hogy nem foglalkozik a tulajdonságok dinamikus csatolásával és leválasztásával, és nem is önhívó összetételre tervezték. Célja egy helyettes biztosítása egy alany számára, amikor annak közvetlen elérése nemkívánatos vagy kényelmetlen, például mert egy távoli gépen található, elérése engedélyhez kötött, vagy maradandó (perzisztens) objektum.

A Helyettes minta kulcsa az alany, amelyhez a helyettes elérést biztosít, vagy megtagadja azt. A Díszítő mintában az elemek csak a szolgáltatások egy részét nyújtják, a többiről a díszítők gondoskodnak. A Díszítő minta használatára akkor kerülhet sor, amikor egy objektum teljes szolgáltatásköre nem határozható meg fordításkor, vagy legalábbis nem egyszerűen. Ez a meghatározatlanság teszi az önhívó összetételt a Díszítő minta lényegi elemévé. Ez a Helyettes minta esetében nem áll fenn, mert itt a középpontban egyetlen kapcsolat – a helyettes és alanya kapcsolata – áll, ami statikusan is kifejezhető.

Az említett különbségek jelentősek, mert mindegyik az objektumközpontú rendszerek egy-egy visszatérő problémájára ad megoldást. Ez azonban nem jelenti azt, hogy ezek a tervezési minták nem használhatók együtt. Elképzelhető például egy olyan „helyettesdíszítő", ami egy helyettest egészít ki új feladatokkal, illetve egy olyan „díszítőhelyettes", ami egy távoli objektumot díszít. Az ilyen keverékek haszna *elképzelhető* (bár gyakorlati példát most egyet sem tudnánk felhozni), de ami biztos, hogy felbonthatók *valóban* hasznos mintákra.

5

Viselkedési minták

A viselkedési minták középpontjában az algoritmusok állnak, illetve a felelősségi körök hozzárendelése az objektumokhoz. E minták nem csupán osztályok vagy objektumok rendszerét írják le, hanem a közöttük folyó kommunikációt is, így a futásidőben nehezen nyomon követhető, bonyolult vezérlési folyamatot modellezik, mégpedig úgy, hogy éppen a vezérlésről terelik el a figyelmünket, hogy az objektumok között fennálló kapcsolatokra összpontosíthassunk.

A viselkedési osztályminták a kívánt viselkedést öröklés segítségével rendelik az egyes osztályokhoz. Ez a fejezet két ilyen mintát tartalmaz, közülük a Sablonfüggvény (Template Method) az egyszerűbb és az ismertebb. E módszer egy algoritmus elvont meghatározására szolgál, amit lépésről lépésre ír le. Minden egyes lépés egy elvont műveletet vagy egy alapműveletet indít el. Az algoritmust az alosztályok töltik meg tartalommal, amikor meghatározzák az elvont műveleteket. A másik itt bemutatandó viselkedési osztályminta az Értelmező (Interpreter), amely egy osztályhierarchia formájában ír le egy nyelvtant, amelyhez egy értelmezőt valósít meg, az osztályok példányaival dolgozó műveletként.

A viselkedési objektumminták öröklés helyett objektum-összetételt alkalmaznak. Egyes minták azt írják le, hogyan működik együtt társobjektumok egy csoportja, hogy végrehajthassanak egy olyan műveletet, amit egyetlen objektum önmagában nem lenne képes végrehajtani. Itt az a lényeges kérdés, hogyan értesülnek egymásról az objektumok. Hivatkozhatnának kifejezetten is egymásra, de ekkor csatolásuk mértéke nemkívánatos módon megnövekedne; szélsőséges esetben minden objektum ismerne minden objektumot. A Közvetítő (Mediator) minta ezt úgy kerüli el, hogy a társobjektumok közé egy közvetítő objektumot iktat be, amely biztosítja a laza csatoláshoz szükséges közvetettséget.

A Felelősséglánc (Chain of Responsibility) minta még ennél is lazább csatolást hoz létre, azáltal, hogy megengedi, hogy egy objektumnak jelölt objektumok láncán keresztül rejtve küldjünk kérelmet. A kérelmet a futásidejű körülményektől függően bármelyik jelölt telje-

sítheti. A jelöltek száma nem korlátos, és azt is megválaszthatjuk, mely jelöltek vegyenek részt futásidőben a felelősséglláncban.

A Megfigyelő (Observer) minta objektumok közötti függőségeket határoz meg és tart fenn. A Megfigyelő klasszikus példája a Smalltalk modell–nézet–vezérlő (Model/View/Controller) felosztása, amelyben a modell minden nézete értesítést kap, ha a modell állapota megváltozik.

Más viselkedési objektumminták alapja a viselkedések objektumba zárása, illetve a kérelmek teljesítésének ezen objektumokra való átruházása. A Stratégia (Strategy) minta egy algoritmust tokoz be egy objektumba, megkönnyítve ezáltal az algoritmus meghatározását és megváltoztatását. A Parancs (Command) minta egy kérelmet zár egy objektumba, hogy az paraméterként átadható, előzménylistában tárolható, vagy más módon kezelhető legyen. Az Állapot (State) minta egy objektum állapotaiból hoz létre önálló objektumot, így az objektum módosíthatja viselkedését, amikor állapotobjektuma megváltozik. A Látogató (Visitor) minta az olyan viselkedéseket zárja egységbe, amelyek másképp több osztályban lennének elosztva, a Bejáró (Iterator) pedig az összesítő (aggregát) objektumok elemeinek elérésére és bejárására ad elvont leírást.

Felelősséglánc

Viselkedési objektumminta

Egyéb nevek

Chain of Responsibility, Válaszlánc

Cél

A minta arra szolgál, hogy elkerüljük a kérelem küldőjének a fogadóhoz való kötését. Ezt úgy érjük el, hogy több objektumnak is jogot adunk a kérelem kezelésére. A fogadó objektumokat láncba állítjuk, amelyen a kérelem addig halad, amíg el nem ér egy objektumot, ami képes a kezelésére.

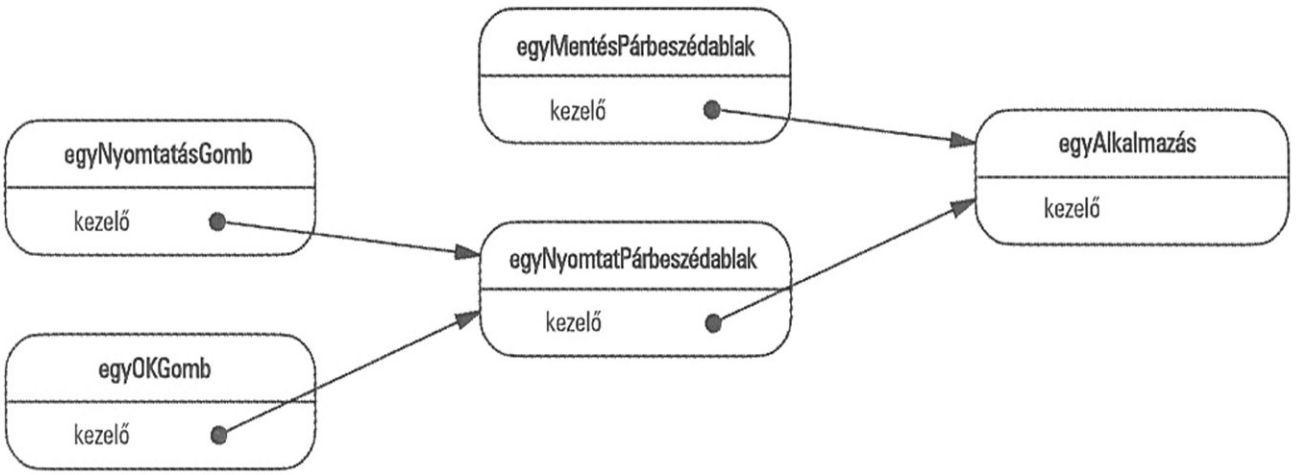
Feladat

Adott egy grafikus felhasználói felület; ehhez szeretnénk környezetfüggő súgót készíteni, vagyis azt elérni, hogy ha a felhasználó a felület valamelyik elemére kattint, az adott elemhez segítséget kaphasson. A megjelenő súgószöveg a választott felületi elemtől és annak környezetétől függ, például egy párbeszédablak egy gombjához más információ tartozhat, mint a főablak egy ugyanolyan gombjához. Ha az adott elemhez nem tartozik súgószöveg, a súgórendszernek az elem környezetéről (például a párbeszédablakról, mint egészről) kell valamilyen általánosabb információt megjelenítenie.

Természetesen adódik, hogy az információkat az általánosság foka szerint rendezzük, a legkonkrétabbtól a legáltalánosabbig. Az is világos, hogy a súgó megjelenítésére irányuló kérelmet több felhasználói felületi objektum is kezelheti; az, hogy az adott pillanatban melyik, a környezettől és az elérhető információ konkrétságától függ.

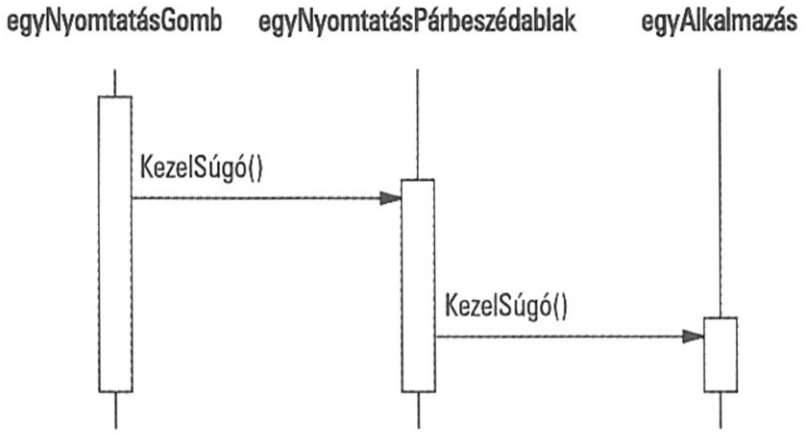
A gond az, hogy az információt *ténylegesen* szolgáltató objektumot a kérelmet *kibocsátó* objektum (például a gomb) nem feltétlenül ismeri. Vagyis egy olyan módszerre van szükség, amellyel a kérelmező objektumot függetleníthetjük az információ szolgáltatására képes objektumoktól. A Felelősséglánc minta ezt a módszert határozza meg.

Az alapgondolat az, hogy a küldőt és a fogadót azáltal választjuk szét, hogy több objektumnak is lehetőséget adunk a kérelem teljesítésére. A kérelem végighalad ezen objektumok láncán, amíg egy olyanhoz nem ér, amelyik ténylegesen végrehajtja.



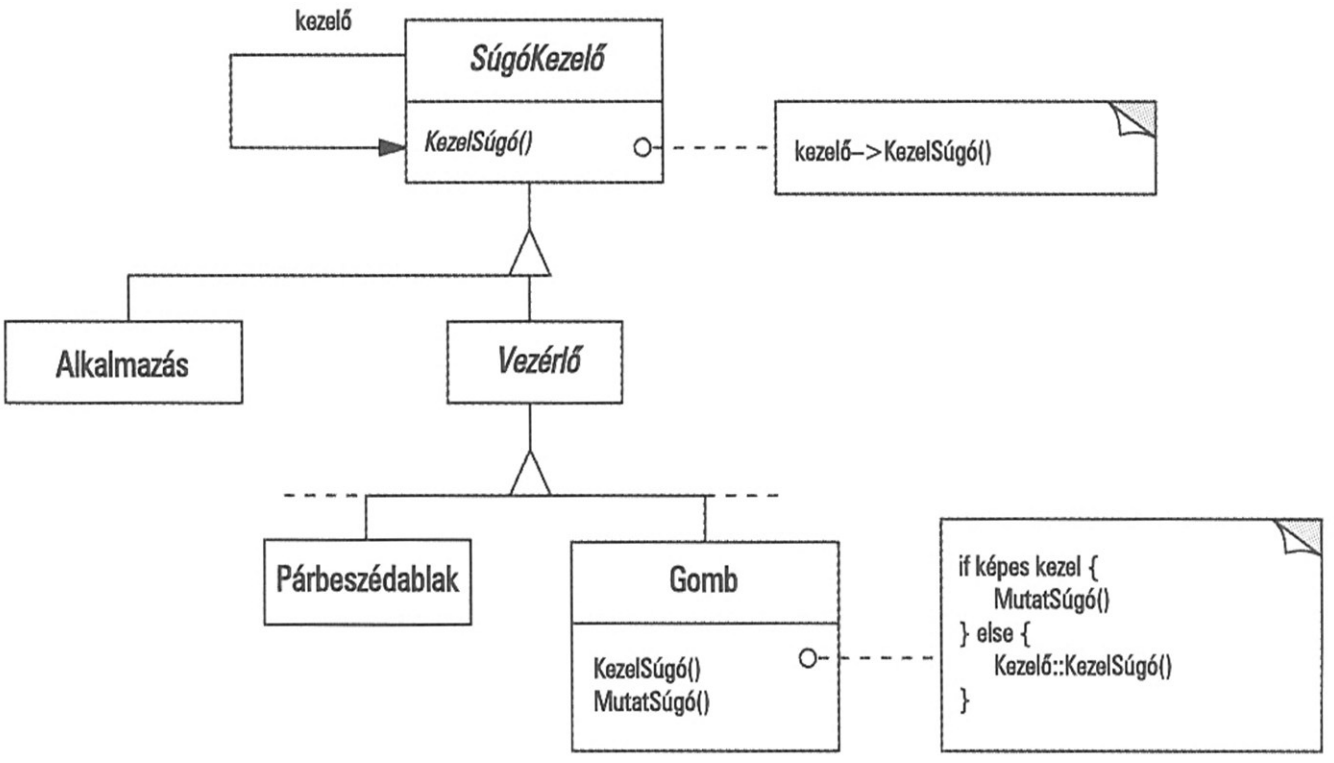
A lánc első objektuma megkapja a kérelmet és vagy teljesíti, vagy továbbítja a lánc következő jelöltjének, amely hasonlóképpen cselekszik. A kérelmező objektum nem tudja, melyik objektum fogja kezelni a kérést, ezért azt mondjuk, a kérelem **rejtett fogadóval** (implicit receiver) rendelkezik.

Tegyük fel, hogy a felhasználó kattintással egy Nyomtatás (Print) feliratú gombhoz kér segítséget. A gombot egy NyomtatásPárbeszédablak (PrintDialog) példány tartalmazza, amely ismeri az őt tartalmazó alkalmazásobjektumot (lásd az előző ábrát). Az alábbi együttműködési diagram azt mutatja, hogyan továbbítódik a kérelem a felelősséglánccon keresztül:



Ebben az esetben sem az egyNyomtatásGomb (aPrintButton), sem az egyNyomtatásPárbeszédablak (aPrintDialog) nem kezeli a kérelmet; az az egyAlkalmazás (anApplication) objektumnál áll meg, amely vagy válaszol rá, vagy figyelmen kívül hagyja. A kérelmet kibocsátó ügyfél nem hivatkozik közvetlenül a végrehajtó objektumra.

Ahhoz, hogy biztosíthassuk a fogadók rejtettségét és továbbíthassuk a kérelmet a láncon át, a lánc objektumai közös felülettel rendelkeznek, ami a kérelmek kezelését és a lánc **következő elemének** (követő, successor) elérését írja le. A súgórendszer például meghatározhat egy SúgóKezelő (HelpHandler) nevű osztályt, a megfelelő KezelSúgó (HandleHelp) művelettel együtt. A SúgóKezelő lehet a jelölt objektumosztályok szülőosztálya, de meghatározható mixin („bekeveredő") osztályként is. Ekkor a súgókérelmeket kezelni kívánó osztályok szülővé tehetik a SúgóKezelőt:



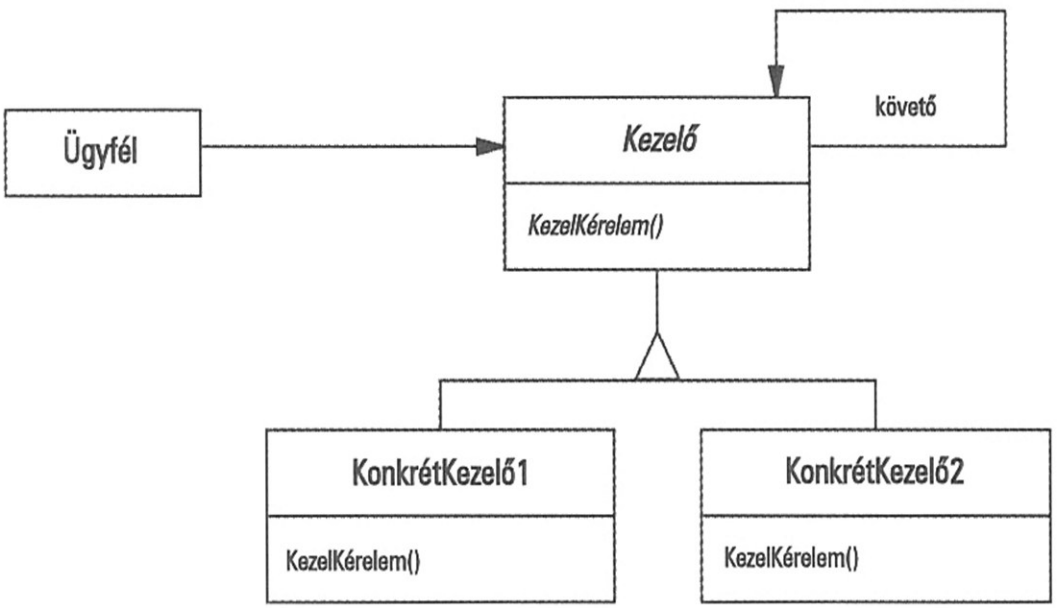
A Gomb (Button), Párbeszédablak (Dialog) és Alkalmazás (Application) osztályok a súgókérelmeket a SúgóKezelő (HelpHandler) műveleteinek segítségével teljesítik. A KezelSúgó (HandleHelp) művelet alapértelmezés szerint a kérelmet a következő jelölthöz továbbítja; az alosztályok e művelet felülbírálásával jeleníthetnek meg súgót megfelelő környezet esetén, más esetben pedig az alapértelmezett megvalósítás szerint továbbítják a kérelmet.

Alkalmazhatóság

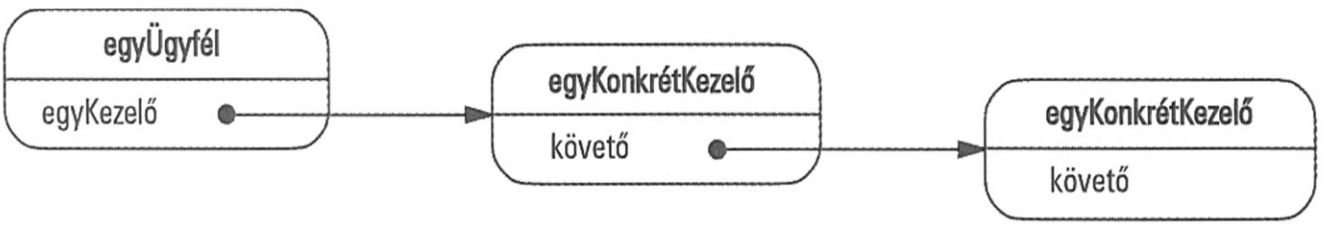
A Felelősséglánc minta a következő esetekben alkalmazható:

- Egy kérelmet egynél több objektum kezelhet, a kezelő pedig *előre* nem ismert, azt automatikusan kell kijelölni.
- Anélkül szeretnénk kérelmet intézni több objektum valamelyikéhez, hogy konkrétan meghatároznánk a fogadót.
- A kérelem teljesítésére képes objektumok halmazát dinamikusan kell meghatározni.

Szerkezet



Az objektumszerkezet valahogy így festhet:



Résztvevők

- **Kezelő** (SúgóKezelő)
  - Felületet határoz meg a kérelmek kezeléséhez.
  - Megvalósítja a következő jelölthöz való kapcsolódást. (Nem kötelező.)
- **KonkrétKezelő** (NyomtatásGomb, NyomtatásPárbeszédablak)
  - Kezeli a felelősségi körébe tartozó kérelmeket.
  - Képes elérni a következő jelöltet.
  - Ha a KonkrétKezelő (ConcreteHandler) képes teljesíteni a kérelmet, akkor teljesíti, egyébként pedig továbbítja a rákövetkező jelöltnek.
- **Ügyfél**
  - Kérelmet intéz a lánc egy KonkrétKezelő objektumához.

Együttműködés

- Amikor az ügyfél kérelmet bocsát ki, a kérelem végighalad a láncon, amíg egy KonkrétKezelő objektum felelősséget nem vállal a kezeléséért.

Következmények

A Felelősséglánc minta előnyei és hátrányai a következők:

1. *Lazább csatolás.* A minta szükségtelenné teszi, hogy a hívók tudják, melyik objektum milyen kérelmeket kezel, csak azt kell tudniuk, hogy a kérelem kezelése „megfelelően" meg fog történni. Sem a küldő, sem a fogadó nem tud egymásról; a lánc objektumainak a lánc szerkezetét sem kell ismerniük.
   Mindezek eredményeként a Felelősséglánc egyszerűsíti az objektumok közötti kapcsolatokat: az objektumok nem hivatkoznak minden fogadójelöltre, csak a rájuk következőre.
2. *Nagyobb rugalmasság az objektumok felelősségi körének kijelölésében.* A Felelősséglánc minta nagyobb szabadságot ad az objektumok feladatainak kiosztásában: egy-egy kérelem kezeléséhez futásidőben is kiegészíthetjük vagy más módon módosíthatjuk a láncot, de ezt párosíthatjuk azzal is, hogy bizonyos kérelmek teljesítésére szakosodott alosztályokat hozunk létre statikusan.
3. *A kérelem teljesítése nem garantált.* Miután a kérelmeknek nincs konkrét fogadójuk, kezelésükre nincs semmilyen *garancia* – a kérelem végighaladhat a láncon, anélkül, hogy egyetlen objektum is válaszolna rá. Ez akkor is megtörténhet, ha a lánc beállítása nem megfelelő.

Megvalósítás

A Felelősséglánc minta megvalósításával kapcsolatban a következő dolgok lényegesek:

1. *A hivatkozási lánc megvalósítása.* A lánc kétféleképpen hozható létre:
   (a) Új hivatkozások meghatározásával. (Ez általában a Kezelőben történik, de a KonkrétKezelők is megtehetik.)
   (b) A meglevő hivatkozások felhasználásával.
   Az eddigi példákban új hivatkozásokat alkalmaztunk, de a hivatkozási lánc létrehozásához sok esetben használhatjuk a meglevő objektumhivatkozásokat is. Ilyenek lehetnek például a rész–egész hierarchiák szülőhivatkozásai; a grafikus felületi elemek általában tartalmaznak ilyeneket. Az Összetétel minta ismertetésénél részletesebben is kitérünk a szülőhivatkozásokra.
   A meglevő hivatkozások használata akkor célszerű, ha illeszkednek a létrehozandó lánchoz. Amennyiben megfelelnek, megkímélnek minket attól, hogy újakat kelljen létrehoznunk, és helyet takarítanak meg. Ha azonban a szerkezet nem tükrözi az alkalmazás igényelte felelősségláncot, a szükséges hivatkozásokat létre kell hoznunk.
2. *Az elemek összekötése.* Ha nincsenek már létező hivatkozások a lánc létrehozásához, magunknak kell azokat elkészítenünk. Ebben az esetben a Kezelő (Handler) nem csak felületet biztosít a kérelmek kezeléséhez, hanem a KezelKérelem (HandleRequest) alapértelmezett megvalósításában hivatkozást is a következő jelöltre, amelyhez a kérelmet továbbítja. Ha valamelyik KonkrétKezelő alosztályra nem tartozik az adott kérelem, a továbbító műveletet nem kell felülbírálnia, hiszen az alapértelmezett megvalósítás feltétel nélkül továbbít.
   Íme a SúgóKezelő (HelpHandler) alaposztály, amely a hivatkozást tartalmazza:

```
class HelpHandler {
public:
        HelpHandler (HelpHandler* s) : _successor(s) {}
        virtual void HandleHelp();
private:
        HelpHandler* _successor;
};

void HelpHandler::HandleHelp() {
        if (_successor) {
                _successor->HandleHelp();
        }
}
```

3. *A kérelmek ábrázolása.* A kérelmek ábrázolására több lehetőségünk is van. A legegyszerűbb, ha a kérelmet a kódba „drótozott" művelethívásként adjuk meg, mint ahogy a KezelSúgó (HandleHelp) esetében is tettük. Ez kényelmes és biztonságos, igaz, csak azok a kérelmek továbbíthatók így, amelyeket a Kezelő osztály meghatároz.

Egy másik lehetőség, hogy egyetlen kezelő függvényt írunk, ami paraméterként kapja meg a kérelem kódját (például egy egész számot vagy karakterláncot). Ezzel a kérelmek halmaza korlátlanul bővíthetővé válik; az egyetlen megkötés, hogy a küldőnek és a fogadónak meg kell egyeznie a kérelem kódolásában.

Ez a megközelítés rugalmasabb, de feltételes utasításokat igényel, amelyekkel a kérelem a kódolásától függően továbbítható. Ezen kívül a paraméterek átadására nincs típusbiztos mód, vagyis magunknak kell be- és kicsomagolnunk azokat, ami nyilvánvalóan kevésbé biztonságos, mintha közvetlenül hívnánk meg egy műveletet.

A paraméter-átadási probléma megoldására önálló *kérelemobjektumokat* használhatunk, amelyek összefogják a kérelem-paramétereket. A kérelmeket mondjuk egy `Request` (Kérelem) nevű osztály képviselheti, új kérelemtípusokat pedig úgy vehetünk fel, hogy alosztályokat származtatunk ebből az osztályból. Ezek az alosztályok aztán különböző típusú paramétereket határozhatnak meg. A kezelőknek ismerniük kell a kérelem típusát (vagyis hogy melyik `Request` alosztályt használják), hogy hozzáférhessenek a paraméterekhez.

A kérelem azonosításához a `Request` meghatározhat egy elérő függvényt, ami az adott osztály azonosítóját adja vissza. Egy másik megoldás, hogy – amennyiben a megvalósításhoz használt nyelv támogatja – a fogadó a futásidejű típusinformációkat használja fel.

Íme egy továbbító függvény vázlata, amely a kérelmek azonosításához kérelemobjektumokat használ. A kérelem típusát a `Request` alaposztályban megadott `GetKind` (SzerezTípus) művelet határozza meg.

```
void Handler::HandleRequest (Request* theRequest) {
        switch (theRequest->GetKind()) {
        case Help:
                //az argumentum átalakítása a megfelelő típusra
                HandleHelp((HelpRequest*) theRequest);
                break;

        case Print:
                HandlePrint((PrintRequest*) theRequest);
                //...
                break;

        default:
                //...
                break;
        }
}
```

Az alosztályok a továbbítást a `HandleRequest` (KezelKérelem) felülbírálásával bővíthetik, így csak azokat a kérelmeket kezelik, amelyekre szakosodtak, a többit a szülőosztályhoz továbbítják. Ez a `HandleRequest` műveletnek hatékony bővítése (nem is felülírása). Lássunk egy példát, hogyan bővítheti egy `ExtendedHandler` (BővítettKezelő) nevű alosztály a `Handler` osztály `HandleRequest`-változatát:

```
class ExtendedHandler : public Handler {
public:
        virtual void HandleRequest(Request* the Request);
        //...
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
        switch (theRequest->GetKind()) {
        case Preview:
                //a Preview - Nyomtatási kép - kérelem kezelése
                break;
        default:
                //a többi kérelmet a Handler-re bízzuk
                Handler::HandleRequest(theRequest);
        }
}
```

4. *Automatikus továbbítás (Smalltalk).* A Smalltalkban a kérelmek továbbítására a `doesNotUnderstand` használható, melynek megvalósítása elfogja a megfelelő metódussal nem rendelkező üzeneteket. A `doesNotUnderstand` felülbírálásával az üzenet az objektumot követő objektumhoz továbbítható, így a továbbítást nem szükséges magunknak megvalósítanunk; az osztályok csak a hozzájuk tartozó kérelmeket kezelik, a többit a `doesNotUnderstand` segítségével más osztályokhoz továbbítják.

Példakód

A következő példa azt illusztrálja, hogyan kezeli a felelősséglánc a súgórendszerhez intézett kérelmeket. Maga a súgókérelem itt konkrétan kifejezett művelet. A kérelmeket a lánc grafikus felületi elemei között a grafikus elemek hierarchiájában jelenlevő szülőhivatkozások segítségével továbbítjuk, az egyéb elemek pedig a Kezelő (Handler) osztályban meghatározott hivatkozás révén kapják meg.

A `HelpHandler` (SúgóKezelő) osztály a súgókérelmek kezelésének felületét határozza meg. Tartalmazza az alapértelmezett (üres) súgótémakört (help topic), illetve egy hivatkozást a lánc következő súgókezelő elemére. A kulcsművelet a `HandleHelp` (KezelSúgó), amelyet az alosztályok felülírnak. A `HasHelp` (VanSúgó) művelet célja, hogy kényelmesebbé tegye annak ellenőrzését, hogy létezik-e az adott elemhez hozzárendelt súgótémakör.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler {
public:
   HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
   virtual bool HasHelp();
   virtual void SetHandler(HelpHandler*, Topic);
   virtual void HandleHelp();
```

```
private:
   HelpHandler* _successor;
   Topic _topic;
};

HelpHandler::HelpHandler (
   HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp() {
   return topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp() {
   if (_successor != 0) {
         _successor->HandleHelp();
   }
}
```

Valamennyi grafikus elem a `Widget` (Vezérlő) elvont osztály alosztálya, ami viszont a `HelpHandler` alosztálya, hiszen a felhasználói felület valamennyi eleméhez tartozhat súgó. (Természetesen mixin alapú megvalósítást is használhattunk volna.)

```
class Widget : public HelpHandler {
protected:
   Widget(Widget* parent, Topic t = NO_HELP_TOPIC);
private:
   Widget* _parent;
};

Widget::Widget (Widget* w, Topic t) : HelpHandler(w, t) {
   _parent = w;
}
```

Példánkban a lánc első kezelőeleme egy gomb (button). A `Button` osztály a `Widget` alosztálya, konstruktora pedig két paramétert vár: egy hivatkozást az őt tartalmazó grafikus elemre, illetve a súgótémakört.

```
class Button : public Widget {
public:
   Button(Widget* d, Topic t = NO_HELP_TOPIC);

   virtual void HandleHelp();
   // A Button által felülírt Widget műveletek...
};
```

A `Button` `HandleHelp`-változata először ellenőrzi, hogy létezik-e súgótémakör a gombokhoz. Ha a fejlesztő nem adott meg ilyet, a kérelem a `HelpHandler`-ben meghatározott `HandleHelp` művelet segítségével továbbítódik a következő jelölthöz. Amennyiben *létezik* súgótémakör, a gomb megjeleníti azt, és a keresés befejeződik.

```
Button::Button (Widget* h, Topic t) : Widget(h, t) { }

void Button::HandleHelp () {
   if (HasHelp()) {
         // súgó a gombhoz
    } else {
         HelpHandler::HandleHelp();
   }
}
```

A `Dialog` (Párbeszédablak) hasonló megoldást ad, de ekkor a következő jelölt nem egy másik grafikus elem, hanem *bármilyen* súgókezelő lehet. A bemutatott programban a jelölt az `Application` (Alkalmazás) osztály egy példánya lesz.

```
class Dialog : public Widget {
public:
   Dialog(HelpHandler* h, Topic t = NO_HELP_TOPIC);
   virtual void HandleHelp();

   // A Dialog által felülírt Widget műveletek...
   //...
};

Dialog::Dialog (HelpHandler* h, Topic t) : Widget(0) {
   SetHandler(h, t);
}

void Dialog::HandleHelp () {
   if (HasHelp()) {
         // súgó a párbeszédablakhoz
    } else {
         HelpHandler::HandleHelp();
    }
}
```

A lánc végén az `Application` egy példánya áll. Az alkalmazás nem grafikus elem (Widget), így az `Application` közvetlenül a `HelpHandler` osztályból származik. Amikor egy súgókérelem erre a szintre ér, az alkalmazás saját magáról szolgáltathat általános információkat, vagy felajánlhatja a súgótémakörök listáját:

```
class Application : public HelpHandler {
public:
   Application(Topic t) : HelpHandler(0, t) { }

   virtual void HandleHelp();
   // az alkalmazáshoz tartozó műveletek...
};

void Application::HandleHelp () {
   // a súgótémakörök listájának megjelenítése
}
```

Az alábbi kód létrehozza és összeköti az említett objektumokat. Párbeszédablakunk egy nyomtatási párbeszédablak, ezért az objektumokhoz a nyomtatással kapcsolatos témakörök tartoznak.

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application* application = new Application(APPLICATION_TOPIC);
Dialog* dialog = new Dialog(application, PRINT_TOPIC);
Button* button = new Button(dialog, PAPER_ORIENTATION_TOPIC);
```

A súgó megjelenítésére irányuló kérelmet a `HandleHelp`-et a lánc valamelyik objektumára meghívva adhatjuk ki. Ha a keresést a gombnál szeretnénk kezdeni, a gombra kell meghívni a műveletet:

```
button->HandleHelp();
```

Ebben az esetben a gomb azonnal kezelni fogja a kérelmet. Figyeljük meg, hogy a `Dialog`-ra bármelyik `HelpHandler` osztály következhet, sőt az, hogy a sorban melyik objektum következik, dinamikusan módosítható, így nem számít, melyik párbeszédablakot nyitottuk meg, mindenképpen a megfelelő környezetfüggő súgót indíthatjuk el.

Ismert felhasználások

A felhasználói események kezeléséhez számos osztálykönyvtár alkalmazza a Felelősséglánc mintát. A Kezelő osztály neve ugyan különbözhet, de az alapgondolat ugyanaz: amikor a felhasználó kattint az egérrel vagy lenyom egy billentyűt, egy esemény váltódik ki, ami végigfut a láncon. A MacApp [App89] és az ET++ [WGM88] az „eseménykezelő" (EventHandler), a Symantec TCL könyvtára [Sym93b] az „ügyintéző" (Bureaucrat), a NeXT AppKit-je [Add94] pedig a „válaszoló" (Responder) nevet használja.

A grafikus szerkesztőprogramok Unidraw keretrendszere parancs (Command) objektumokat határoz meg, amelyek a Component (Elem) és ComponentView (ElemNézet) objektumok felé irányuló kérelmeket zárják egységbe. A parancsok ilyen értelemben kérelmek, hiszen az összetevők vagy nézetek egy művelet végrehajtásához ugyanúgy a Megvalósítás részben bemutatott „kérelmek mint objektumok" elv alapján értelmezik a kapott parancsot. Az elemek (Component) és nézeteik (ComponentView) hierarchikus rendbe állíthatók, melyben a parancs értelmezését az elemek a szülőjüknek továbbíthatják, ami aztán továbbítja azt a saját szülőjének, és így tovább – vagyis előáll a felelősséglánc.

Az ET++ a grafikus frissítéshez használja a Felelősséglánc mintát. Amikor egy grafikus objektumnak frissítenie kell egy része megjelenítését, az InvalidateRect műveletet hívja meg, amit a grafikus objektumok maguk nem képesek kezelni, mert nem tudnak eleget saját kör-

nyezetükről. Előfordulhat például, hogy az objektum a kooordinátarendszerét átalakító Scroller (Gördítősáv) vagy Zoomer (Nagyító) objektumba ágyazódik, vagyis az objektum görgethető, illetve nagyítható, így egy része néha nem látható. Emiatt az InvalidateRect alapértelmezett megvalósítása a frissítési kérelmet továbbadja a befoglaló tároló objektumnak. A továbbítási lánc utolsó tagja a Window (Ablak) osztály egy példánya. Amikor a kérelem a Window-hoz ér, a frissítendő terület koordinátáinak megfelelő átalakítására már biztosan sor került. Az InvalidateRect műveletet a Window úgy kezeli, hogy értesítést küld az ablakkezelő rendszer felületének és frissítést kér.

Kapcsolódó minták

A Felelősséglánc mintát gyakran használják együtt az Összetétel mintával, ahol az elemek szülői jelentik a lánc következő elemét.

Parancs

Viselkedési objektumminta

Cél

A kérelmeket objektumokba zárjuk, aminek célja, hogy az ügyfeleknek paraméterként különböző kérelmeket adjunk át, ezeket sorba állítsuk vagy naplózzuk, illetve támogassuk a műveletek visszavonását.

Egyéb nevek

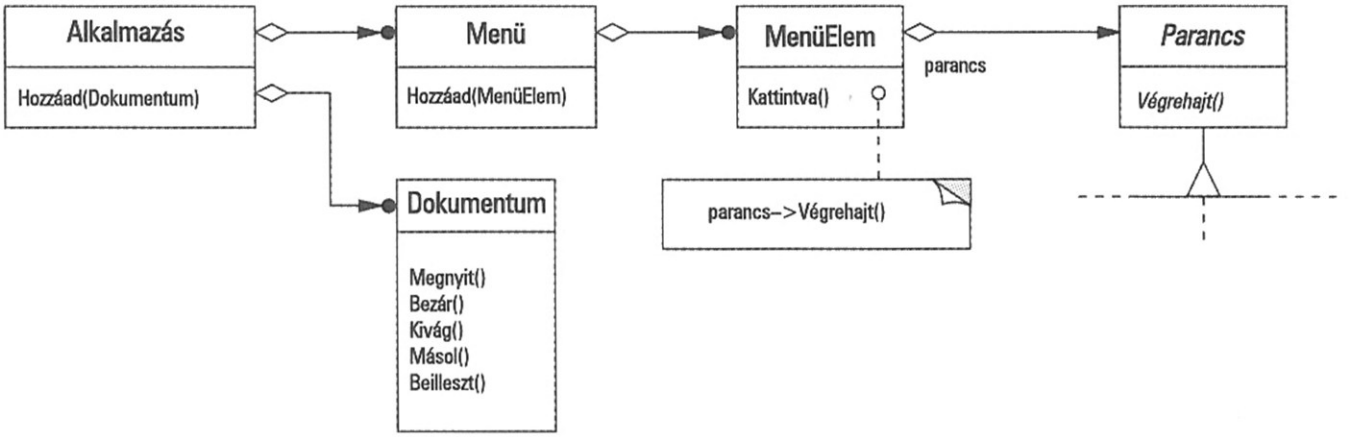
Command, Művelet, Akció (Action), Tranzakció (Transaction)

Feladat

Időnként szükséges lehet objektumokhoz anélkül intézni kérelmeket, hogy a kérelmezett műveletről vagy a kérelem fogadójáról bármilyen ismeretünk lenne. A felhasználói felületi eszköz- vagy elemkészletek például olyan objektumokat tartalmaznak, mint a gombok és menük, amelyek a felhasználó tevékenységétől függő műveleteket hajtanak végre (vagyis kérelmeket teljesítenek). Maga az elemkészlet azonban nem adhat konkrét megvalósítást a kérelmekhez magukban a gombokban és menükben, hiszen csak az adott elemkészletet használó alkalmazások tudják, melyik objektumnak mit kell csinálnia. Az elemkészlet fejlesztője nem ismerheti előre a kérelem fogadóját, illetve az általa végrehajtandó műveleteket.

A Parancs tervezési minta lehetővé teszi az elemkészletek objektumainak, hogy meg nem határozott alkalmazás-objektumokhoz intézzenek kérelmeket, mégpedig úgy, hogy magukat a kérelmeket is objektumokká alakítják. Ezek az objektumok azután ugyanúgy tárolha-

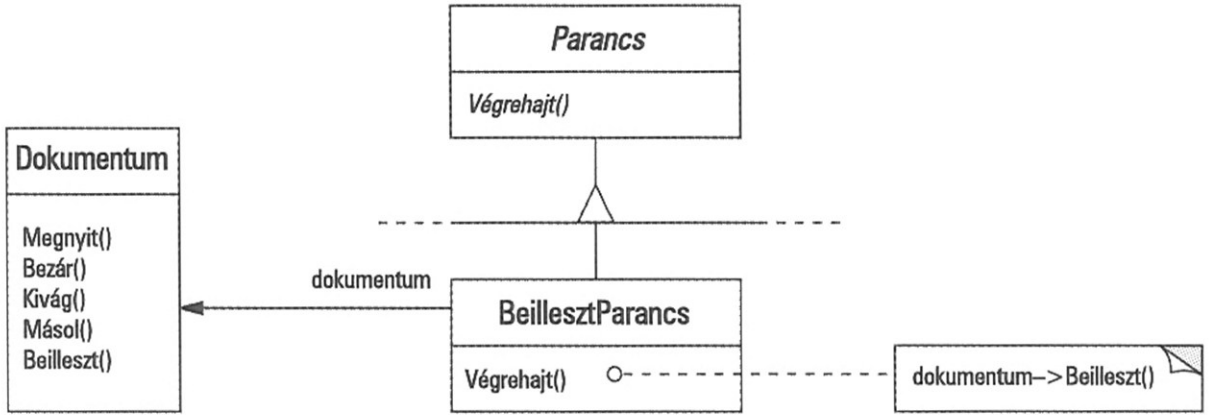
tók és továbbadhatók, mint más objektumok. A minta kulcsa az elvont Parancs (Command) osztály, amely a műveletek végrehajtásához biztosít felületet. Legegyszerűbb formájában a felület egy elvont Végrehajt (Execute) műveletet tartalmaz. A konkrét Parancs alosztályok a fogadó példányváltozóként való tárolásával és a Végrehajt-nak a kérelmet meghívó megvalósításával egy fogadó–művelet párt határoznak meg. A kérelem teljesítéséhez szükséges képességeket a fogadó tartalmazza.



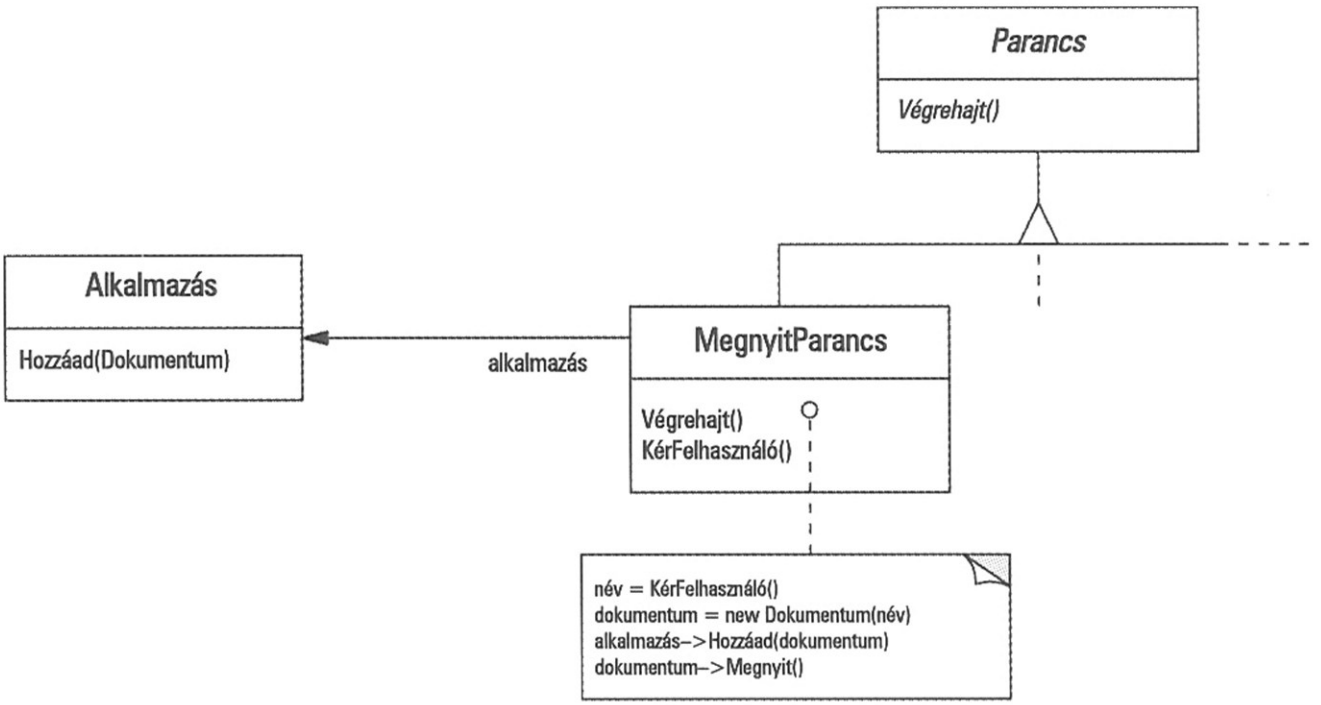
A Parancs objektumokkal a menük könnyen elkészíthetők. A menük (Menü, Menu) minden eleme a MenüElem (MenuItem) osztály egy példánya. A menüket és elemeiket a felhasználói felület egyéb részeivel együtt az Alkalmazás (Application) osztály hozza létre. Az Alkalmazás osztály gondoskodik a felhasználó által megnyitott Dokumentum (Document) objektumok nyilvántartásáról is.

Az alkalmazás minden menülemet valamelyik Parancs alosztály egy példányával állít elő. Amikor a felhasználó kiválasztja az egyik elemet, a MenüElem a hozzá tartozó parancsra meghívja a Végrehajt (Execute) műveletet, az pedig végrehajtja a parancsot. Maguk a menüelemek nem tudják, hogy a Parancs melyik alosztálya kapcsolódik hozzájuk. A Parancs alosztályok tárolják a kérelem fogadóját és meghívnak rá egy vagy több műveletet.

A BeillesztParancs (PasteCommand) például szöveg beillesztését teszi lehetővé a vágólapról a dokumentumba. A parancs fogadója az a Dokumentum objektum, amelyet példányosításkor megkap, a Végrehajt művelet pedig a Beilleszt (Paste) parancsot a fogadó dokumentumra hívja meg.

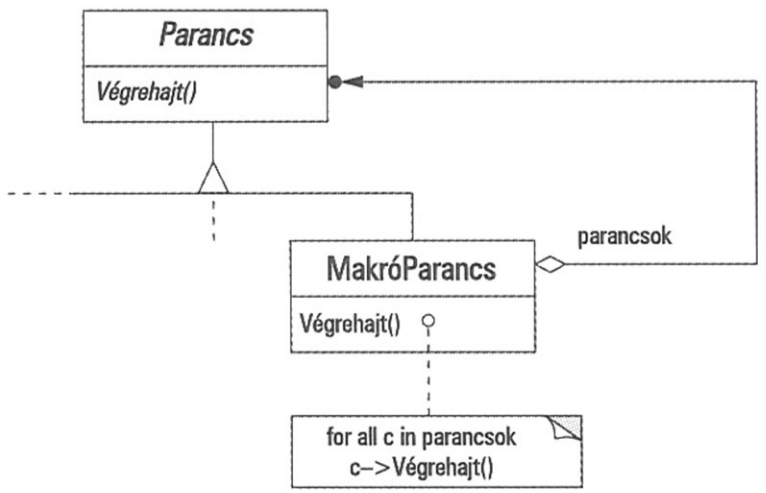


A MegnyitParancs (OpenCommand) Végrehajt művelete ettől eltér: a felhasználótól egy dokumentumnevet kér, létrehozza az ennek megfelelő Dokumentum objektumot, azt hozzáadja a fogadó alkalmazáshoz, majd megnyitja a dokumentumot.



Előfordulhat, hogy egy menüelemnek parancsok *sorozatát* kell végrehajtania. A dokumentumoldalt normál méretben középre helyező MenüElem például egy DokumentumKözépreParancs (CenterDocumentCommand) és egy NormálMéretParancs (NormalSizeCommand) objektumból épülhet fel. Miután a parancsok ilyetén összefűzése elég gyakori, egy MakróParancs (MacroCommand) nevű osztályt létrehozva lehetővé tehetjük, hogy a menüelemek korlátlan számú parancsot hajthassanak végre.

A MakróParancs a Parancs konkrét alosztálya, amely egyszerűen parancsok sorozatát hajtja végre. Nincs kifejezetten megadott fogadója, mert az általa sorban végrehajtott parancsok meghatározzák saját fogadójukat.



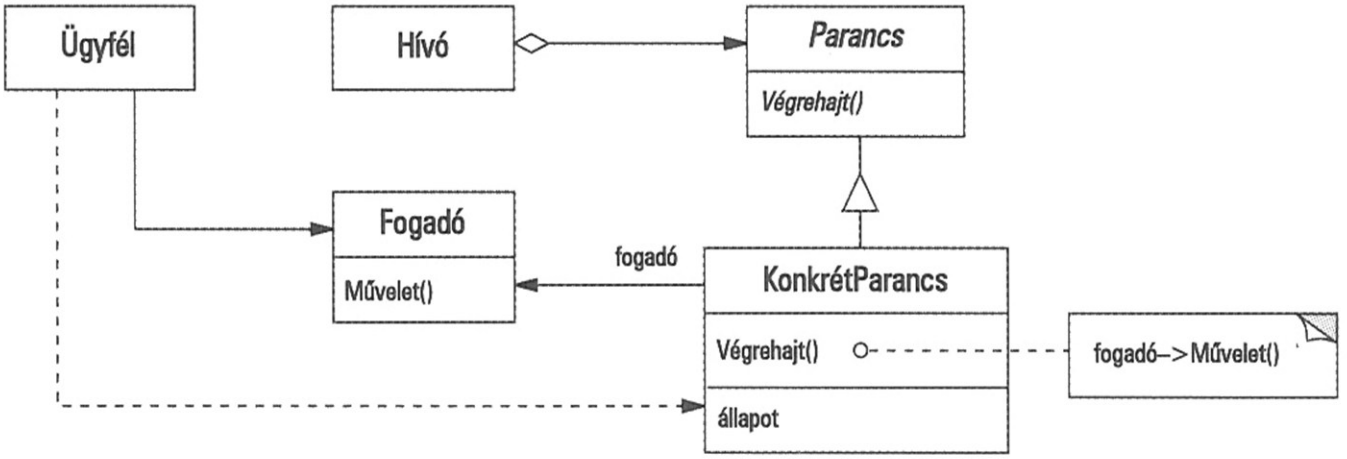
A fenti példák mindegyikében megfigyelhetjük, hogyan választja el a Parancs minta a műveletet kezdeményező objektumot a művelet végrehajtásának képességével rendelkező objektumtól. Ez igen nagy rugalmasságot ad a felhasználói felület megtervezésében, hiszen így egy alkalmazásban egy szolgáltatáshoz menüt és parancsgombot is kapcsolhatunk, ha azok ugyanazon Parancs alosztály egy példányán osztoznak. A parancsok dinamikusan ki is cserélhetők, ami környezetfüggő helyi menük megvalósításánál jöhet jól, a parancsok nagyobb egységekben való egyesítésével pedig a parancssorozatok összeállítását támogathatjuk. Mindez azért lehetséges, mert a kérelmet kibocsátó objektumnak csak azt kell tudnia, hogyan adja ki a kérelmet; annak végrehajtási módjáról nem kell ismeretekkel rendelkeznie.

Alkalmazhatóság

A Parancs minta használata a következő esetekben célszerű:

- Objektumoknak paraméterként egy végrehajtandó műveletet szeretnénk átadni, mint ahogy feljebb, a MenüElem objektumok esetében is tettük. Az ilyen paraméterezés módja az eljárásközpontú (procedurális) nyelvekben a **visszahívható** (callback) függvények használata. A visszahívható függvény olyan függvény, amit egy adott ponton bevezetünk, de később hívjuk meg. A parancsok a visszahívható függvények objektumközpontú megfelelői.
- Kérelmeket különböző időpontokban szeretnénk meghatározni, sorba állítani, illetve végrehajtani. A Parancs objektumok élettartama független lehet az eredeti kérelemtől. Ha a kérelem fogadóját címtér-független módon ábrázoljuk, a kérelemhez tartozó parancsobjektumot egy másik folyamathoz irányíthatjuk és ott teljesíthetjük a kérést.
- Támogatni szeretnénk a művelet-visszavonást. A Parancs Végrehajt művelete hatásának megfordításához állapotokat tárolhat. A Parancs felületnek rendelkeznie kell egy Visszavon (Unexecute) művelettel, ami megfordítja egy korábbi Végrehajt hívás hatását. A végrehajtott parancsokat előzménylistában tároljuk. A korlátlan szintű visszavonást és ismételt végrehajtást úgy érjük el, hogy a Végrehajt, illetve a Visszavon hívásával előre-hátra bejárjuk ezt a listát.
- Támogatni szeretnénk a változások naplózását, hogy rendszerösszeomlás esetén helyreállíthassuk a korábbi állapotot. Ha a Parancs felületet betöltő és tároló műveletekkel egészítjük ki, következetes változásnaplót tarthatunk fenn. Az összeomlásból való helyreállítás abból áll, hogy a naplózott parancsokat visszatöltjük a lemezről, majd a Végrehajt művelettel újra végrehajtjuk azokat.
- Egy programot alapműveletekből felépített magas szintű műveletekre szeretnénk alapozni. Az ilyen szerkezet gyakori a **tranzakciókat** támogató információs rendszerekben. A tranzakciók adatok változtatásainak halmazát zárják egységbe. A Parancs minta segítségével modellezhetjük a tranzakciókat, a parancsok közös felülete révén pedig minden tranzakciót egyformán hívhatunk meg. A minta azt is egyszerűbbé teszi, hogy a rendszert új tranzakciókkal egészítsük ki.

Szerkezet



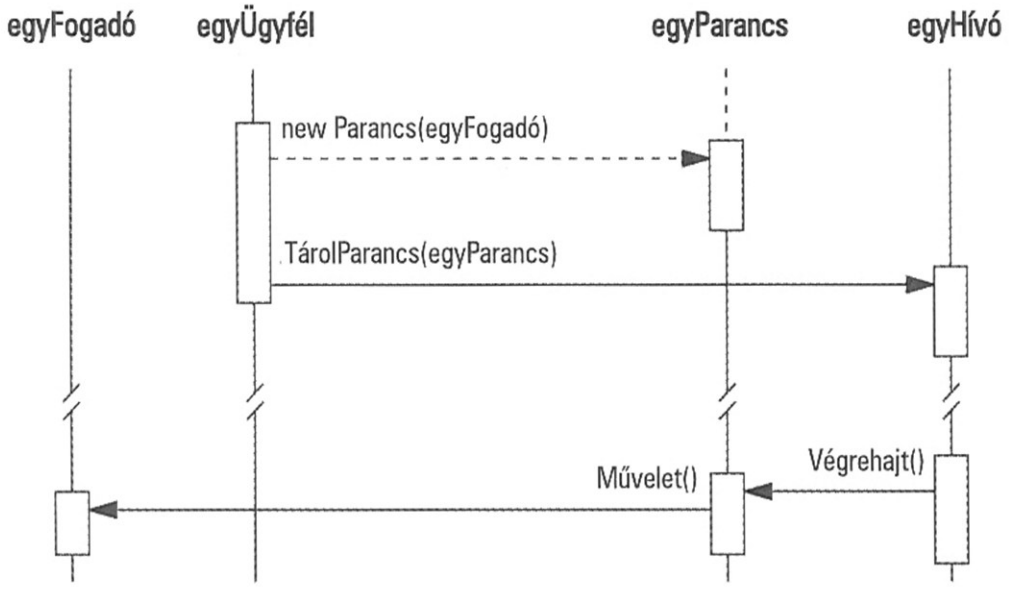
Résztvevők

- Parancs
  - Felületet biztosít egy művelet végrehajtásához.
- KonkrétParancs (BeillesztParancs, MegnyitParancs)
  - Összeköt egy Fogadó (Receiver) objektumot és egy műveletet.
  - A Fogadónak megfelelő művelet(ek) meghívásával megvalósítja a Végrehajt (Execute) műveletet.
- Ügyfél (Alkalmazás)
  - Létrehoz egy KonkrétParancs (ConcreteCommand) objektumot és beállítja annak fogadóját.
- Hívó (MenüElem)
  - Felkéri a parancsot a kérelem teljesítésére.
- Fogadó (Dokumentum, Alkalmazás)
  - Tudja, hogyan kell végrehajtani az adott kérelemhez kapcsolódó műveleteket. Bármelyik osztály betöltheti a Fogadó szerepét.

Együttműködés

- Az ügyfél létrehoz egy KonkrétParancs objektumot és meghatározza annak fogadóját.
- Valamelyik Hívó (Invoker) objektum elraktározza a KonkrétParancs objektumot.
- A Hívó a Végrehajt (Execute) műveletnek az adott parancsra való meghívásával kérelmet bocsát ki. Ha a parancsok visszavonhatók, a KonkrétParancs a Végrehajt meghívása előtt elraktározza a visszavonáshoz szükséges állapotot.
- A KonkrétParancs objektum a kérelem teljesítéséhez műveleteket hív meg a fogadóján.

Az alábbi diagram az említett objektumok közötti együttműködést mutatja, illusztrálva, hogyan választja el a Parancs a hívót a fogadótól, illetve az utóbbi által teljesített kérelemtől.



Következmények

A Parancs minta előnyei a következők:

1. A Parancs minta elválasztja a műveletet kezdeményező objektumot a művelet végrehajtási módját ismerő objektumtól.
2. A parancsok első osztályú objektumok, ugyanúgy kezelhetők és bővíthetők, mint bármely más objektum.
3. A parancsokból parancsösszetételek alakíthatók ki. Ennek egyik példája a korábban említett MakróParancs osztály. Az összetett parancsok általában az Összetétel minta példái.
4. Könnyű új parancsokat felvenni, mert a meglevő osztályokat nem kell módosítani.

Megvalósítás

A Parancs minta megvalósításával kapcsolatban a következő dolgok lényegesek:

1. *Mennyire legyenek „intelligensek" a parancsok?* Az egyes parancsok képességek széles körével rendelkezhetnek. Az egyik szélsőség az lehet, ha a parancs csupán összekapcsol egy fogadót a kérelem végrehajtásához szükséges műveletekkel, míg a másik az, ha maga valósít meg mindent, fogadó objektum bevonása nélkül. Az utóbbi akkor lehet hasznos, ha a meglevő osztályoktól független parancsokat szeretnénk meghatározni, ha nincs megfelelő fogadó objektum, vagy ha a parancs egyébként is ismeri a fogadóját. Egy másik alkalmazás-ablakot létrehozó parancs például az ablak létrehozásán kívül más objektumok készítésére is képes lehet. A két szélsőség között az olyan parancsok találhatók, amelyek éppen elegendő ismerettel rendelkeznek ahhoz, hogy dinamikusan megtalálják a fogadójukat.

2. *A visszavonás és újbóli végrehajtás támogatása.* A parancsobjektumok akkor képesek támogatni a visszavonást és újbóli végrehajtást, ha rendelkeznek valamilyen módszerrel (például Visszavon vagy Mégse – Unexecute, Undo – művelettel) a végrehajtás „megfordítására". A KonkrétParancs osztályoknak ehhez általában az állapotot is tárolniuk kell. Az állapot a következőket rögzíti:
   - a Fogadó objektumot, amely ténylegesen végrehajtja a kérelem teljesítéséhez szükséges műveleteket,
   - a fogadón végrehajtott művelet argumentumait,
   - a fogadó azon eredeti értékeit, amelyek a kérelem kezelése következtében megváltozhatnak. A fogadónak biztosítania kell azokat a műveleteket, amelyek segítségével a parancs a fogadót visszaállíthatja a korábbi állapotba.

   Ha csak egyszeri visszavonást akarunk támogatni, elég, ha az alkalmazás csak az utoljára végrehajtott parancsot tárolja. A többszintű visszavonáshoz, illetve ismétléshez viszont a programnak a végrehajtott parancsok **előzménylistájára** van szüksége, amelynek lehetséges hossza határozza meg, hány műveletet vonhatunk vissza. Az előzménylista parancssorozatokat tárol; ha a sorban visszafelé haladunk és „visszafordító" műveleteket hajtunk végre, töröljük a hatást, ha pedig előre, akkor újból végrehajtjuk a parancsokat.

   Előfordulhat, hogy a visszavonható parancsokról az előzménylistára történő felvétel előtt másolatot kell készíteni. Erre akkor lehet szükség, ha az eredeti – mondjuk egy menüelemtől érkező – kérelmet teljesítő parancsobjektummal később más műveleteket kívánunk végrehajtatni. Ha az objektum állapota a különböző hívásoknál más és más lehet, a másolás biztosítja, hogy a különböző változatokat meg tudjuk különböztetni.

   Egy kijelölt objektumokat törlő TörölParancs (DeleteCommand) objektumnak például minden végrehajtásnál különböző objektumokat kell tárolnia, ezért a TörölParancs-ról a végrehajtás után másolatot kell készíteni, és ezt a másolatot helyezzük majd az előzménylistára. Ha egy adott parancs állapota sohasem változik, a másolás nem szükséges; ebben az esetben elég, ha az előzménylistában egy hivatkozást helyezünk el a parancsra. A listára helyezés előtt másolandó parancsok prototípusként viselkednek. (Lásd a Prototípus tervezési mintát a 3. fejezetben.)
3. *A hibahalmozódás elkerülése a visszavonások során.* Amikor egy megbízható, a jelentést megőrző visszavonási–ismétlési rendszert próbálunk kialakítani, az állapothiba gondot okozhat. Ahogy parancsokat hajtunk végre, vonunk vissza és ismétlünk meg, a hibák halmozódhatnak, és az alkalmazás állapota végül eltérhet az eredeti értékektől. Ennek elkerülése érdekében további információkat kell tárolnunk a parancsobjektumokban, hogy az eredeti állapotba való visszaállítást biztosíthassuk. Ebben segíthet az Emlékeztető tervezési minta, amelynek révén a parancsok anélkül férhetnek hozzá ezekhez az információkhoz, hogy más objektumok belső szerkezetét felfednék.
4. *C++ sablonok használata.* A (1) vissza nem vonható, illetve (2) az argumentumokat nem igénylő parancsokhoz C++ sablonokat használhatunk, így elkerülhetjük, hogy minden művelethez és fogadóhoz újabb Parancs alosztályt kelljen létrehoznunk. A Példakód részben megmutatjuk, hogyan.

Példakód

Az alább bemutatott C++ kód a Feladat részben bevezetett Parancs (Command) osztályok megvalósítását vázolja fel; az `OpenCommand`, `PasteCommand` és `MacroCommand` (MegnyitParancs, BeillesztParancs és MakróParancs) meghatározására kerül sor. Először persze az elvont `Command` osztályt kell elkészítenünk:

```
class Command {
public:
   virtual ~Command();

   virtual void Execute() = 0;
protected:
   Command();
};
```

Az `OpenCommand` a felhasználó által megadott nevű dokumentumot nyitja meg. Konstruktorában egy `Application` (Alkalmazás) objektumot kell átadnunk neki. A megnyitandó dokumentum nevének megadására az `AskUser` (KérFelhasználó) eljárás kéri meg a felhasználót.

```
class OpenCommand : public Command {
public:
   OpenCommand(Application*);

   virtual void Execute();
protected:
   virtual const char* AskUser();
private:
   Application* _application;
   char* _response;
};

OpenCommand::OpenCommand (Application* a) {
   _application = a;
}

void OpenCommand::Execute () {
   const char* name = AskUser();

   if (name != 0) {
         Document* document = new Document(name);
         _application->Add(document);
         document->Open();
   }
}
```

A `PasteCommand` egy `Document` (Dokumentum) objektumot vár, ami a fogadója lesz. A fogadót paraméterként adjuk át a `PasteCommand` konstruktorának.

```
class PasteCommand : public Command {
public:
   PasteCommand(Document*);

   virtual void Execute();
private:
   Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
   _document = doc;
}

void PasteCommand::Execute () {
   _document->Paste();
}
```

Az egyszerű – vissza nem vonható és argumentumot nem váró – parancsokhoz osztálysablont használhatunk, így a parancs fogadóját paraméterként adhatjuk át. Az ilyen parancsok számára hozzuk létre a `SimpleCommand` (EgyszerűParancs) sablon alosztályt, amelynek paramétere a `Receiver` (Fogadó) típusa, feladata pedig a kapcsolat fenntartása a fogadó objektum és egy művelet között, amelyet egy tagfüggvényt címző mutatóként tárolunk.

```
template <class Receiver>
class SimpleCommand : public Command {
public:
   typedef void (Receiver::* Action)();

   SimpleCommand(Receiver* r, Action a) :
         _receiver(r), _action(a) { }

   virtual void Execute();
private:
   Action _action;
   Receiver* _receiver;
};
```

A konstruktor a fogadót és a műveletet a megfelelő példányváltózókban tárolja, az `Execute` (Végrehajt) pedig egyszerűen végrehajtja a műveletet a fogadón.

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
   (_receiver->*_action)();
}
```

Ahhoz, hogy létrehozzon egy parancsot, amely a `MyClass` (SajátOsztály) osztály egy példányára meghívja az `Action`-t (Művelet), az ügyfél a következő kódot tartalmazza:

```
MyClass* receiver = new MyClass;
//...
Command* aCommand =
   new SimpleCommand<MyClass>(receiver, &MyClass::Action);
//...
aCommand->Execute();
```

Ne feledjük, hogy ez a megoldás csak az egyszerű parancsok esetében működik. Az összetettebb parancsok, amelyek nem csupán fogadójukat tartják nyilván, hanem más argumentumokat, illetve a visszavonási állapotot is, a `Command` alosztályai kell legyenek.

A `MacroCommand` parancsok sorozatát kezeli, illetve műveleteket biztosít a parancssorozat bővítésére és szűkítésére. Fogadóra itt kifejezetten nincs szükség, hiszen az egyes „alparancsok" meghatározzák a saját fogadójukat.

```
class MacroCommand : public Command {
public:
   MacroCommand();
   virtual ~MacroCommand();

   virtual void Add(Command*);
   virtual void Remove(Command*);

   virtual void Execute();
private:
   List<Command*>* _cmds;
};
```

A `MacroCommand` működésének kulcsa az `Execute` tagfüggvény, amely bejárja a parancssorozat elemeit és egyesével végrehajtja rajtuk az `Execute` műveletet.

```
void MacroCommand::Execute () {
   ListIterator<Command*> i(_cmds);

   for (i.First(); !i.IsDone(); i.Next()) {
         Command* c = i.CurrentItem();
         c->Execute();
    }
}
```

Megjegyzendő, hogy amennyiben a `MacroCommand` megvalósítja az `Unexecute` (Visszavon) műveletet, az egyes parancselemek visszavonására az `Execute` megvalósításához képest fordított sorrendben van szükség.

Végül, a MacroCommand-nak rendelkeznie kell a parancselemek kezeléséhez, köztük az azok törléséhez szükséges műveletekkel.

```
void MacroCommand::Add (Command* c) {
   _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
   _cmds->Remove(c);
}
```

Ismert felhasználások

A Parancs minta valószínűleg Lieberman dolgozatában [Lie85] bukkant fel először. A visszavonható műveletek parancsokkal való megvalósításának ötletét az Apple [App89] tette népszerűvé. Az ET++ [WGM88], az InterViews [LCI+92] és a Unidraw [VL90] szintén tartalmaznak olyan osztályokat, amelyek a Parancs mintát követik. Az InterViews Action (Művelet) elvont osztálya parancsszolgáltatásokat biztosít, az ActionCallback (MűveletVisszahívás) sablon pedig, amelynek paramétere egy műveleti függvény, automatikusan képes példányosítani a parancs-alosztályokat.

A THINK osztálykönyvtár [Sym93b] ugyancsak parancsokat használ a visszavonható műveletek támogatására. A parancsokat a THINK „feladatoknak" (Task) nevezi. A feladatobjektumok egy Felelősségláncon haladnak végig.

A Unidraw parancsobjektumai abban az értelemben egyediek, hogy üzenetként viselkedhetnek. A Unidraw parancsok értelmezés céljából más objektumokhoz küldhetők, az értelmezés pedig a fogadó objektumtól függően változhat. A fogadó emellett át is ruházhatja az értelmezés feladatát egy másik objektumra, ami jellemzően a szülője egy nagyobb szerkezetben, például egy Felelősséglánccban. A Unidraw parancsok fogadói így tulajdonképpen „számított" és nem tárolt fogadók. Az értelmezés módja a futásidejű típusinformációktól függ.

Coplien leírja [Cop92], hogyan készíthetünk funktorokat, vagyis **függvényobjektumokat** a C++-ban. Használatukat a függvényhívó operátor (`operator()`) túlterhelésével részben elrejti. A Parancs minta azonban ettől eltér, hiszen középpontjában a fogadó és a függvény (vagyis a művelet) összekötése, és nem csupán egy függvény biztosítása áll.

Kapcsolódó minták

A MakróParancsok megvalósításához az Összetétel minta használható.

Az Emlékeztető minta segítségével a parancs által a hatásának visszavonásához igényelt állapotot tárolhatjuk.

A listára helyezés előtt másolandó parancsok Prototípusként viselkednek.

Értelmező

Viselkedési osztályminta

Egyéb nevek

Interpreter

Cél

Egy adott nyelv nyelvtanát ábrázoljuk, illetve ehhez az ábrázoláshoz értelmezőt biztosítunk, amely annak alapján képes a nyelv mondatait megérteni.

Feladat

Ha egy bizonyos típusú probléma rendszeresen felbukkan, célszerű lehet egy egyszerű nyelv mondataiként megfogalmazni a probléma előfordulási lehetőségeit, majd egy értelmezőt készíteni, amely e mondatok értelmezésével megoldja a problémát.

Gyakori feladat például, hogy bizonyos mintákra illeszkedő karakterláncokat keressünk. A karakterlánc-minták meghatározásának szabványos nyelvét a szabályos kifejezések (regular expressions) jelentik. Ha a kereső algoritmusok a keresendő karakterláncokat leíró szabályos kifejezéseket értelmezik, elkerülhetjük, hogy minden minta–karakterlánc párosításhoz külön algoritmust kelljen írnunk.

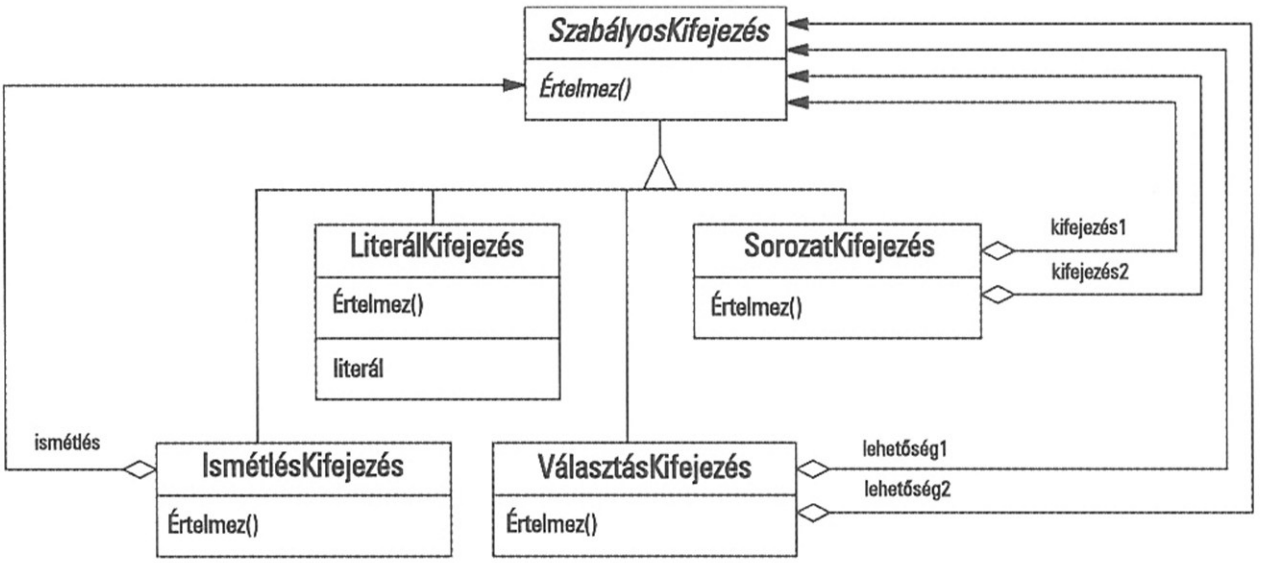
Az Értelmező tervezési minta annak módszerét adja meg, hogyan határozhatjuk meg egy egyszerű nyelv nyelvtanát, hogyan ábrázolhatjuk a nyelv mondatait, és hogyan értelmezhetjük ezeket a mondatokat. Az említett esetben a minta a szabályos kifejezések nyelvtanának leírását, adott szabályos kifejezések ábrázolását, illetve azok értelmezését segíti.

Tegyük fel, hogy az alábbi nyelvtan határozza meg a szabályos kifejezéseket:

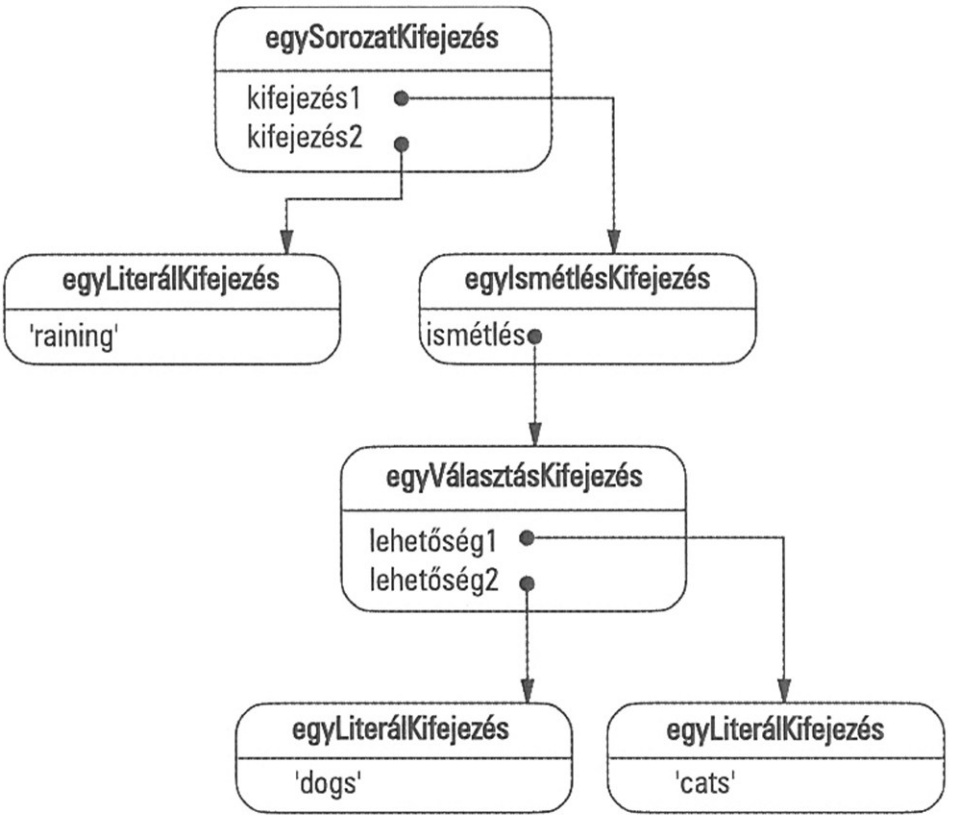
```
kifejezés ::= literál | választás | sorozat | ismétlés |
              '(' kifejezés ')'
választás ::= kifejezés '|' kifejezés
sorozat ::= kifejezés '&' kifejezés
ismétlés ::= kifejezés '*'
literál ::= 'a'|'b'|'c'|... {'a'|'b'|'c'|...}*
```

A kezdőszimbólum a `kifejezés` (expression) a zárószimbólum pedig az alapszavakat meghatározó `literál` (literal).

Az Értelmező mintában minden nyelvtani szabályt egy-egy osztállyal ábrázolunk. A szabály jobb oldalán álló szimbólumok ezen osztályok példányai. A fenti nyelvtant öt osztállyal ábrázolhatjuk: a SzabályosKifejezés (RegularExpression) elvont osztállyal, és annak négy alosztályával (LiterálKifejezés, VálasztásKifejezés, SorozatKifejezés, IsmétlésKifejezés – vagyis LiteralExpression, AlternationExpression, SequenceExpression és RepetitionExpression). Az utolsó három osztály alkifejezéseket tartalmazó változókat határoz meg.



A nyelvtan által meghatározott valamennyi szabályos kifejezést egy, az osztályok példányaiból felépülő elvont szintaxisfa ábrázolja.



A fenti szintaxisfa például a következő szabályos kifejezésnek felel meg:

```
raining & (dogs | cats) *
```

A szabályos kifejezésekhez úgy készíthetünk értelmezőt, ha a SzabályosKifejezés (RegularExpression) minden alosztályára meghatározzuk az Értelmez (Interpret) műveletet. Az Értelmez argumentumként azt a környezetet várja, amelyben a kifejezést értelmeznie kell. A környezet a bemenő karakterláncot jelenti, illetve azt, hogy a karakterlánc mekkora részét próbálták már a mintához illeszteni. A SzabályosKifejezés alosztályai az Értelmez műveletet úgy valósítják meg, hogy az adott környezet alapján a bemenő karakterlánc következő részét vizsgálják. Lássunk néhány példát:

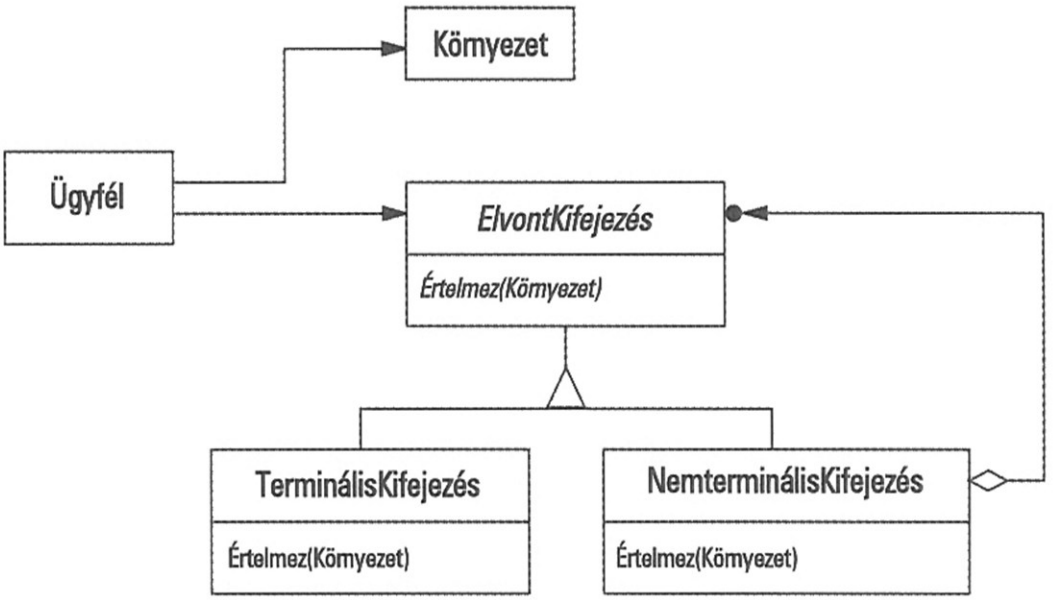
- A LiterálKifejezés azt ellenőrzi, hogy a bemenet illeszkedik-e az általa meghatározott literálra.
- A VálasztásKifejezés azt ellenőrzi, hogy a bemenet illeszkedik-e bármelyik választási lehetőségre.
- Az IsmétlésKifejezés azt ellenőrzi, hogy a bemenet tartalmaz-e ismétlődő kifejezést.

Alkalmazhatóság

Az Értelmező minta használata akkor célszerű, ha van értelmezhető nyelv, amelynek kifejezéseit elvont szintaxisfában ábrázolhatjuk. Az Értelmező minta a következő helyzetekben alkalmazható a legnagyobb sikerrel:

- A nyelvtan egyszerű. A bonyolult nyelvtanok osztályhierarchiája túl nagy és kezelhetetlen lehet; ezek esetében az elemző generátorok és hasonló eszközök jobb megoldást kínálnak, mert elvont szintaxisfa felépítése nélkül képesek értelmezni a kifejezéseket, amivel helyet és valószínűleg időt takaríthatunk meg.
- A hatékonyság nem létfontosságú szempont. A leghatékonyabb értelmezők általában *nem* közvetlenül elemzik a fákat, hanem először más formára alakítják azokat. A szabályos kifejezéseket például gyakran alakítják állapotautomatákká. Mindazonáltal a *fordító* ekkor is megvalósítható az Értelmező minta segítségével.

Szerkezet



Résztvevők

- **ElvontKifejezés** (SzabályosKifejezés)
  - Egy elvont Értelmez (Interpret) műveletet vezet be, ami az elvont szintaxisfa minden csomópontjára közös.
- **TerminálisKifejezés** (LiterálKifejezés)
  - A nyelvtan terminálszimbólumaihoz kapcsolódó Értelmez műveletet valósítja meg.
  - A mondatok minden egyes terminálszimbólumához egy-egy példány szükséges.
- **NemterminálisKifejezés** (VálasztásKifejezés, IsmétlésKifejezés, SorozatKifejezés)
  - A nyelvtan minden $R ::= R_1 R_2 \ldots R_n$ szabályához egy-egy ilyen osztály szükséges.
  - Az R_1-től az R_n-ig minden szimbólumhoz egy-egy ElvontKifejezés (AbstractExpression) típusú példányváltozót biztosít.
  - Megvalósítja a nyelvtan nemterminális szimbólumaihoz kapcsolódó Értelmez műveletet. Az Értelmez jellemzően ismételten önmagát hívja meg az R_1–R_n szimbólumokat jelképező változókra.
- **Környezet**
  - Az értelmező globális környezetéről tartalmaz információkat.
- **Ügyfél**
  - A nyelvtan által meghatározott nyelv egy adott mondatát ábrázoló elvont szintaxisfát épít fel (vagy kap meg). A fa a NemterminálisKifejezés (NonterminalExpression) és a TerminálisKifejezés (TerminalExpression) osztályok példányaiból áll.
  - Meghívja az Értelmez műveletet.

Együttműködés

- Az ügyfél felépíti (vagy megkapja) a mondatot, egy NemterminálisKifejezés és TerminálisKifejezés példányokból álló elvont szintaxisfa formájában, majd előkészíti a környezetet és meghívja az Értelmez műveletet.
- A NemterminálisKifejezés csomópontok mindegyike az alkifejezéseknek megfelelően határozza meg az Értelmez műveletet. A TerminálisKifejezések Értelmez művelete az alapesetet határozza meg.
- Az egyes csomópontokban levő Értelmez műveletek a Környezet (Context) segítségével tárolják és érik el az értelmező állapotát.

Következmények

Az Értelmező minta előnyei és hátrányai a következők:

1. *A nyelvtan könnyen módosítható és bővíthető.* Mivel a minta a nyelvtani szabályokat osztályokkal ábrázolja, a nyelvtant örökléssel módosíthatjuk vagy bővíthetjük. A meglevő kifejezések fokozatosan módosíthatók, az újakat pedig a régiek változataiként határozhatjuk meg.
2. *A nyelvtan megvalósítása is könnyű.* Az elvont szintaxisfa csomópontjait meghatározó osztályok megvalósítása hasonló; az osztályok könnyen megírhatók, sőt, létrehozásuk fordítóprogram- vagy elemzőgenerátor segítségével automatizálható is.
3. *A bonyolult nyelvtanok fenntartása nehéz.* Az Értelmező minta nyelvtani szabályonként legalább egy osztályt határoz meg. (Ha a szabályokat BNF formában adjuk meg, több osztályra is szükség lehet.) Ennélfogva a sok szabályból álló nyelvtanok kezelése nehéz. A probléma enyhítésére használhatunk más tervezési mintákat (lásd a Megvalósítás részt), de ha a nyelvtan nagyon bonyolult, az olyan megoldások, mint a fordító- vagy elemzőgenerátorok használata célszerűbb lehet.
4. *A kifejezések értelmezésére könnyű új módszereket hozzáadni.* Az Értelmező minta egyszerűbbé teszi, hogy a kifejezéseket új módon értékeljük ki. Ha egy adott kifejezés osztályára új műveletet határozunk meg, támogathatjuk például a típusellenőrzést. Ha a kifejezések értelmezési módját gyakran változtatjuk, érdemes a Látogató mintát alkalmazni, hogy elkerüljük a nyelvtani osztályok módosítását.

Megvalósítás

Az Értelmező és az Összetétel minta a megvalósítás módjában számos szempontból hasonlít. Az alábbiak viszont kifejezetten az Értelmező mintához kapcsolódnak:

1. *Elvont szintaxisfa építése.* Az Értelmező minta nem magyarázza el, *hogyan* hozhatjuk létre az elvont szintaxisfát, vagyis az elemzéssel (parsing) nem foglalkozik. A fa létrehozható táblavezérlésű elemzővel, „kézi" (általában rekurzív ereszkedő) elemzővel, illetve közvetlenül az ügyfél által.

2. *Az Értelmez művelet meghatározása.* A kifejezésosztályokban nem kell meghatároznunk az Értelmez műveletet. Ha gyakran készítünk új értelmezőt, érdemesebb a Látogató mintát alkalmazni és a műveletet külön „látogató" objektumba helyezni. Egy programozási nyelv nyelvtana például számos műveletet végez az elvont szintaxisfákon (típusellenőrzést, hatékonyságnövelést, kód-előállítást stb.). Ahhoz, hogy elkerüljük, hogy minden nyelvtani osztályhoz meg kelljen határoznunk ezeket a műveleteket, célszerűbb látogatót alkalmaznunk.
3. *A terminálszimbólumok megosztása a Pehelysúlyú mintával.* Azon nyelvtanok esetében, amelyek mondatai egy adott terminálszimbólum számos előfordulását tartalmazzák, érdemes a szimbólum egyetlen másolatát megosztani. A számítógépprogramok nyelvtanai jó példák erre – gondoljunk csak a kódban számos helyen felbukkanó változókra. A Feladat részben leírt példa mondataiban a `dog` terminálszimbólum (amit a LiterálKifejezés osztály ábrázol) fordulhat elő többször. A terminális csomópontok általában nem tárolnak információt az elvont szintaxisfában elfoglalt helyükről; az értelmezéshez szükséges környezetet a szülőcsomópontok adják át nekik. Így megkülönböztetünk megosztott (belső) és átadott (külső) állapotokat, és alkalmazhatjuk a Pehelysúlyú mintát. Például a LiterálKifejezés minden `dog` példánya megkapja a környezetet, ami az addig mintára illesztett karakterlánc-részt tartalmazza. Ezután minden ilyen LiterálKifejezés ugyanazt csinálja az Értelmez műveletben: ellenőrzi, hogy a bemenet következő része tartalmaz-e `dog`-ot, függetlenül attól, hol helyezkedik el a példány a fában.

Példakód

Íme két példa; az első egy teljes program Smalltalk nyelven, ami ellenőrzi, hogy egy karaktersorozat illeszkedik-e egy adott szabályos kifejezésre, a második egy logikai (Boolean) kifejezéseket értékelő C++ program.

A szabályos kifejezésre illesztő program azt vizsgálja, hogy egy adott karaktersorozatot meghatároznak-e a nyelv szabályos kifejezései. A szabályos kifejezések nyelvtana a következő (ezúttal angolul):

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression 'repeat'
literal ::= 'a'|'b'|'c'|... {'a'|'b'|'c'|...}*
```

Ez a nyelvtan a Feladat részben szereplő példa némileg módosított változata. Azért volt szükség a változtatásra, mert a Smalltalk-ban a `"*"` szimbólum nem lehet utótagos (postfix) művelet; így ezt a `repeat`-re cseréltük.

```
(('dog' | 'cat') repeat & 'weather')
```

A fenti szabályos kifejezés például a `"dog dog cat weather"` bemenő karakterláncnak felel meg.

Az illesztő megvalósításához a pár oldallal korábban bemutatott öt osztályt határozzuk meg. A `SequenceExpression` (SorozatKifejezés) osztály példányváltozói az `expression1` (kifejezés1) és az `expression2` (kifejezés2), amelyek az osztály gyermekeit jelölik az elvont szintaxisfában. Az `AlternationExpression` (VálasztásKifejezés) a választási lehetőségeket az `alternative1` (lehetőség1) és `alternative2` (lehetőség2) példányváltozókban tárolja, míg a `RepetitionExpression` (IsmétlésKifejezés) az ismétlendő kifejezést a `repetition` (ismétlés) változóban, a `LiteralExpression` (LiterálKifejezés) `components` (elemek) példányváltozója pedig objektumok (valószínűleg karakterek) listáját tartalmazza, amelyek azt a literális karakterláncot jelölik, aminek illeszkednie kell a bemenő karaktersorozatra.

A `match:` (illeszt) művelet a szabályos kifejezés értelmezője, amit az elvont szintaxisfát meghatározó valamennyi osztály megvalósít. Argumentuma az `inputState` (bemenet-Állapot), ami az illesztési folyamat aktuális állapotát jelöli a bemenő karakterlánc egy részének elolvasása után.

Ezt az állapotot bemeneti folyamok jellemzik, amelyek a szabályos kifejezés által elfogadható bemeneteket ábrázolják. (Ez nagyjából megfelel annak, mintha egy egyenértékű véges állapotú automata minden lehetséges állapotát rögzítenénk az adott pontig, a bemenő adatfolyam felismerése során.)

Az aktuális állapot a `repeat` (ismétel) művelet szempontjából a legfontosabb. Tegyük fel, hogy a szabályos kifejezés a következő:

```
'a' repeat
```

Ekkor az értelmező illeszkedőként értelmezi az `"a"`, `"aa"`, `"aaa"` stb. sorozatokat. Ha a szabályos kifejezés azonban ez:

```
'a' repeat & 'bc'
```

Az illeszkedő sorozatok máris az `"abc"`, `"aabc"`, `"aaabc"` és így tovább. Nézzünk egy másik változatot:

```
'a' repeat & 'abc'
```

Az `"aabc"` bemenet illesztése az `"'a' repeat"` alkifejezéshez ekkor két bemenő folyamot eredményez, amelyek közül az egyik a bemenet egy karakterét, míg a másik annak két karakterét vizsgálta már meg. Csak az egy karaktert elfogadó folyam fog illeszkedni a maradék `"abc"`-re.

Most nézzük a szabályos kifejezést meghatározó osztályok mindegyikére a `match:` meghatározását. A `SequenceExpression` meghatározása sorban minden alkifejezésére vizsgálja az illeszkedést, a bemenő folyamokat pedig általában eltávolítja az `inputState`-ből.

```
match: inputState
    ^ expression2 match: (expression1 match: inputState).
```

Az AlternationExpression-ök az egyes választási lehetőségek állapotának uniójából álló állapotot adnak vissza. A match: meghatározása az AlternationExpression esetében a következő:

```
match: inputState
    | finalState |
    finalState := alternative1 match: inputState.
    finalState addAll: (alternative2 match: inputState).
    ^ finalState
```

A RepetitionExpression match: művelete annyi esetleg illeszkedő állapotot próbál találni, amennyit csak lehet:

```
match: inputState
    | aState finalState |
    aState := inputState.
    finalState := inputState copy.
    [aState isEmpty]
        whileFalse:
            [aState := repetition match: aState.
            finalState addAll: aState].
        ^ finalState
```

A kimeneti állapot általában több állapotot tartalmaz, mint a bemeneti, mert a RepetitionExpression a repetition egy, kettő vagy több előfordulását előfordulását illesztheti a bemenetre. A bemeneti állapotok e lehetséges állapotokat ábrázolják, így a szabályos kifejezés következő elemei eldönthetik, melyik állapot a helyes.

A LiteralExpression match: művelete a kifejezés összetevőit megpróbálja minden lehetséges bemeneti adatfolyamra ráilleszteni, és csak azokat a folyamokat tartja meg, amelyekben illeszkedést talál:

```
match: inputState
    | finalState tStream |
    finalState := Set new.
    inputState
        do:
            [:stream | tStream := stream copy.
                (tStream nextAvailable:
                    components size
                ) = components
                    ifTrue: [finalState add: tStream]
            ].
        ^ finalState
```

A bemeneti adatfolyamot a `nextAvailable:` (következőElérhető) üzenet lépteti előre; ez az egyetlen léptető `match:` művelet. Figyeljük meg, hogy a visszakapott állapot a bemeneti adatfolyam másolatát tartalmazza, ezáltal biztosítva, hogy egy illeszkedő literál soha ne változtassa meg az adatfolyamot. Ez azért fontos, mert az `AlternationExpression` lehetőségeinek a bemenő adatfolyam azonos példányait kell látniuk.

Most, hogy meghatároztuk az elvont szintaxisfát felépítő osztályokat, leírhatjuk a felépítés módját. Nem írunk elemzőt a szabályos kifejezésekhez, inkább meghatározunk néhány műveletet a `RegularExpression` osztályokhoz. Így a Smalltalk kifejezések kiértékelése a megfelelő szabályos kifejezés elvont szintaxisfáját eredményezi, és ugyanúgy használhatjuk a Smalltalk beépített fordítóját, mintha a szabályos kifejezések elemzője lenne.

Az elvont szintaxisfa felépítéséhez az `"|"`, a `"repeat"`, illetve az `"& "` szimbólumokat a `RegularExpression` műveleteiként meg kell határoznunk. Az osztályban a meghatározások a következőképpen néznek ki:

```
& aNode
    ^ SequenceExpression new
          expression1: self expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
    alternative1: self alternative2: aNode asRExp

asRExp
    ^ self
```

Az `asRExp` művelet a literálokat `RegularExpression`-ökké alakítja. Ezeket a műveleteket a `String` (Karakterlánc) osztályban határozzuk meg:

```
& aNode
    ^ SequenceExpression new
          expression1: self asRExp expression2: aNode asRExp

repeat
    ^ RepetitionExpression new repetition: self

| aNode
    ^ AlternationExpression new
    alternative1: self asRExp alternative2: aNode asRExp

asRExp
    ^ LiteralExpression new components: self
```

Ha ezeket a műveleteket az osztályhierarchia magasabb fokán határoznánk meg (SequenceableCollection a Smalltalk-80-ban, IndexedCollection a Smalltalk/V-ben), akkor az olyan osztályokra is érvényesek lennének, mint az Array vagy az OrderedCollection, így a szabályos kifejezéseket bármilyen objektumból álló sorozatokra illeszthetnénk.

A második példa egy logikai (Boolean) kifejezéseket kezelő és kiértékelő program, C++ nyelven megvalósítva. E nyelv terminálszimbólumai logikai változók, vagyis a true és false állandók. A nemterminális szimbólumok az and, or és not operátorokat tartalmazó kifejezéseket jelölik. A nyelvtan meghatározása a következő:[1]

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
                '(' BooleanExp ')'
AndExp ::= BooleanExp 'and' BooleanExp
OrExp ::= BooleanExp 'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' | 'false'
VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'
```

A logikai kifejezésekre két műveletet határozunk meg. Az első, az Evaluate (Értékel), egy logikai kifejezést értékel ki, egy olyan környezetben, amely minden változóhoz igaz vagy hamis (true–false) értéket rendel. A második, a Replace (Cserél), egy változó kifejezésre való felcserélésével új logikai kifejezést állít elő. A Replace egyben azt is mutatja, hogy az Értelmező mintát nem csak kifejezések kiértékelésére használhatjuk; ebben az esetben például magának a kifejezésnek a kezelésére.

Itt csak a BooleanExp, a VariableExp és az AndExp (LogikaiKif, VáltozóKif, ÉsKif) osztályokat mutatjuk be részletesen. Az OrExp és a NotExp (VagyKif, NemKif) hasonlóak az AndExp-hez, a Constant (Állandó) osztály a logikai állandókat ábrázolja.

A BooleanExp határozza meg minden logikai kifejezést meghatározó osztály felületét:

```
class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();

    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};
```

A Context (Környezet) osztály a változók logikai értékekké alakítását határozza meg, amely értékeket a C++ true és false állandóival ábrázoljuk. A Context felülete a következő:

[1] Az egyszerűség kedvéért eltekintünk a műveletek kiértékelési sorrendjétől, és feltételezzük, hogy ez a szintaxisfát felépítő objektum felelőssége.

```
class Context {
public:
   bool Lookup(const char*) const;
   void Assign(VariableExp*, bool);
};
```

A `VariableExp` egy nevesített változót ábrázol:

```
class VariableExp : public BooleanExp {
public:
   VariableExp(const char*);
   virtual ~VariableExp();

   virtual bool Evaluate(Context&);
   virtual BooleanExp* Replace(const char*, BooleanExp&);
   virtual BooleanExp* Copy() const;
private:
   char* _name;
};
```

A konstruktor argumentumként a változó nevét várja:

```
VariableExp::VariableExp (const char* name) {
   _name = strdup(name);
}
```

Egy adott változó kiértékelésének eredménye a változó értéke az aktuális környezetben.

```
bool VariableExp::Evaluate (Context& aContext) {
   return aContext.Lookup(_name);
}
```

A változó másolása egy új `VariableExp`-et eredményez:

```
BooleanExp* VariableExp::Copy () const {
   return new VariableExp(_name);
}
```

Ahhoz, hogy egy változót egy kifejezésre cserélhessünk, ellenőrizzük, hogy a változó neve megegyezik-e az argumentumként kapott névvel.

```
BooleanExp* VariableExp::Replace (
   const char* name, BooleanExp& exp
) {
```

```
    if (strcmp(name, _name) == 0) {
            return exp.Copy();
    } else {
            return new VariableExp(_name);
    }
}
```

Az AndExp egy olyan kifejezést jelöl, amelyet két logikai kifejezés AND-del való összekapcsolásával állítunk elő.

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}
```

Ha kiértékelünk egy AndExp-et, a tényezőket értékeljük, a visszakapott eredmény pedig a logikai „és" lesz.

```
bool AndExp::Evaluate (Context& aContext) {
    return
            _operand1->Evaluate(aContext) &&
            _operand2->Evaluate(aContext);
}
```

Az AndExp a Copy (Másol) és Replace műveleteket a tényezőin ismételten végrehajtott hívásokkal valósítja meg:

```
BooleanExp* AndExp::Copy () const {
    return
            new AndExp(_operand1->Copy(), _operand2->Copy();
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
```

```
            new AndExp(
                  _operand1->Replace(name, exp),
                  _operand2->Replace(name, exp)
            );
}
```

Most már meghatározhatjuk a következő logikai kifejezést...

```
(true and x) or (y and (not x))
```

...és kiértékelhetjük az x és y változókhoz rendelt `true` és `false` értékek egy adott esetében:

```
BooleanExp* expression;
Context context;

VariableExp* x = new VariableExp("X");
VariableExp* y = new VariableExp("Y");

expression = new OrExp(
   new AndExp(new Constant(true), x),
   new AndExp(y, new NotExp(x))
);

context.Assign(x, false);
context.Assign(y, true);

bool result = expression->Evaluate(context);
```

A kifejezés értéke az x és y ezen esetében `true` lesz. A változók más értékeinél is kiértékelhetjük a kifejezést; ehhez egyszerűen csak meg kell változtatnunk a környezetet.

Végezetül, az y változót egy új kifejezésre cserélhetjük, majd újra kiértékelhetjük:

```
VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);
```

Ez a példa rámutat egy fontos dologra az Értelmező mintával kapcsolatban, mégpedig arra, hogy egy mondatot többféle művelet értelmezhet. A `BooleanExp` esetében meghatározott három művelet közül az `Evaluate` felel meg leginkább az arról alkotott elképzeléseinknek, hogy egy értelmezőnek mit kell tennie – értelmeznie egy programot vagy kifejezést, majd visszaadni egy egyszerű eredményt.

Ugyanakkor a `Replace` is tekinthető értelmezőnek, mégpedig olyannak, amelynek környezete a cserélendő változó neve, illetve a helyére lépő kifejezés, eredménye pedig egy új kifejezés. Még a `Copy`-ra is gondolhatunk úgy, mint egy üres környezetű értelmezőre. Furcsának tűnhet persze, hogy e két műveletet értelmezőnek tekintjük, hiszen tulajdonképpen nem mások, mint fákon végzett alapműveletek. A Látogató tervezési minta példáinál majd bemutatjuk, miként lehet a három műveletet újraépítve önálló „értelmező látogató"-ba helyezni, ami bizonyítja hasonlóságuk nagy fokát.

Az Értelmező minta nem csupán egy művelet, amelyet „szétosztunk" egy, az Összetétel mintát használó osztályhierarchiában. Az `Evaluate`-et azért tekintjük értelmezőnek, mert a `BooleanExp` osztályhierarchiára úgy gondolunk, mint ami egy nyelvet ábrázol.

Ha adott egy hasonló osztályhierarchia, amely alkatrészeket ábrázol, nem valószínű, hogy az olyan műveleteket, mint a Weight vagy a Copy értelmezőnek tekintjük – bár ugyanúgy egy, az Összetétel mintát alkalmazó osztályhierarchiában kapnak helyet –, egyszerűen mert az alkatrészekre nem gondolunk nyelvként. Mindez persze nézőpont kérdése: ha közzétennék nyelvtanukat, a velük dolgozó műveletekre is úgy nézhetnénk, mint a nyelv értelmezésének módszereire.

Ismert felhasználások

Az Értelmező mintát széles körben alkalmazzák az objektumközpontú nyelvek fordítóprogramjaiban, amilyen a Smalltalk is. A SPECTalk a minta segítségével a bemeneti fájlformátumok leírását értelmezi [Sza92], a QOCA megszorításfeloldó elemkészlet kötéseket vizsgál vele [HHMV92].

Ha legegyszerűbb formájára gondulunk (vagyis mint egy, az Összetétel mintán alapuló osztályhiererchiában elosztott műveletre), azt mondhatjuk, hogy az Összetétel minta szinte valamennyi alkalmazása tartalmazza az Értelmező mintát is, de használatát azokra az esetekre célszerű korlátozni, amikor az osztályhierarchia egy nyelvet ábrázol.

Kapcsolódó minták

Összetétel: az elvont szintaxisfa az Összetétel minta egy példája.

Pehelysúlyú: megmutatja, hogyan oszthatók meg az elvont szintaxisfa terminálszimbólumai.

Bejáró: az értelmező a szerkezet bejárásához felhasználhatja a Bejáró mintát.

Látogató: arra használható, hogy egy osztályban tartsuk az elvont szintaxisfa egyes csomópontjainak viselkedését.

Bejáró

Viselkedési objektumminta

Cél

Az összetett objektumok elemeinek soros elérését a háttérben megbúvó ábrázolás felfedése nélkül biztosító módszer kialakítása.

Egyéb nevek

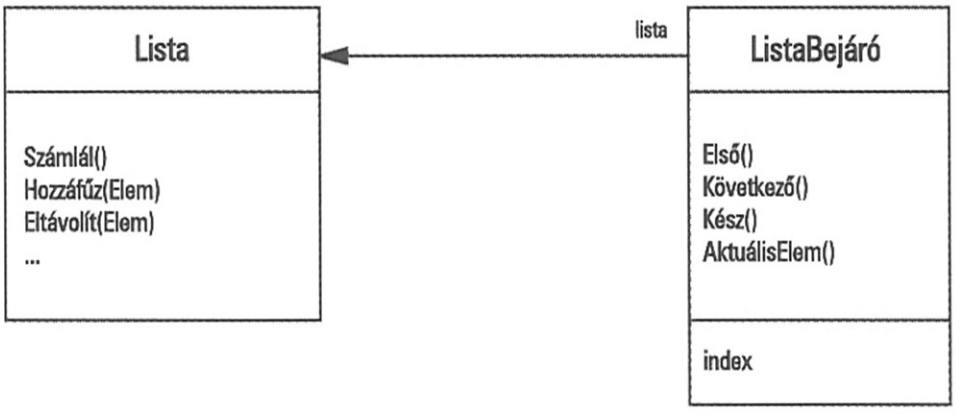
Iterator, Iterátor, Cursor (Kurzor, Sormutató)

Feladat

Az összesítő objektumoknak (aggregátumoknak), például a listáknak, módot kell adniuk arra, hogy elemeiket a belső szerkezet felfedése nélkül érhessük el. A listát emellett tudnunk kell különböző módokon bejárni, attól függően, hogy mit szeretnénk véghezvinni. A Lista (List) felületet azonban valószínűleg nem akarjuk felduzzasztani különféle bejáró műveletekkel, még akkor sem, ha pontosan tudjuk, milyen műveletekre lesz szükségünk. Arra is szükség lehet, hogy ugyanazt a listát egyszerre több módon járjuk be.

A Bejáró tervezési minta minderre módot ad. A minta kulcsa, hogy az elérés és bejárás felelősségi körét a lista objektumból egy **bejáró** (iterátor) objektumba helyezzük. A lista elemeinek elérésére szolgáló felületet a Bejáró (Iterator) osztály határozza meg; az ebbe az osztályba tartozó objektumok tartják számon, melyik az aktuális elem, vagyis hogy mely elemeket jártuk már be.

Egy Lista osztály például a ListaBejárót (ListIterator) hívja meg, és az alábbi kapcsolatok állnak fenn köztük:



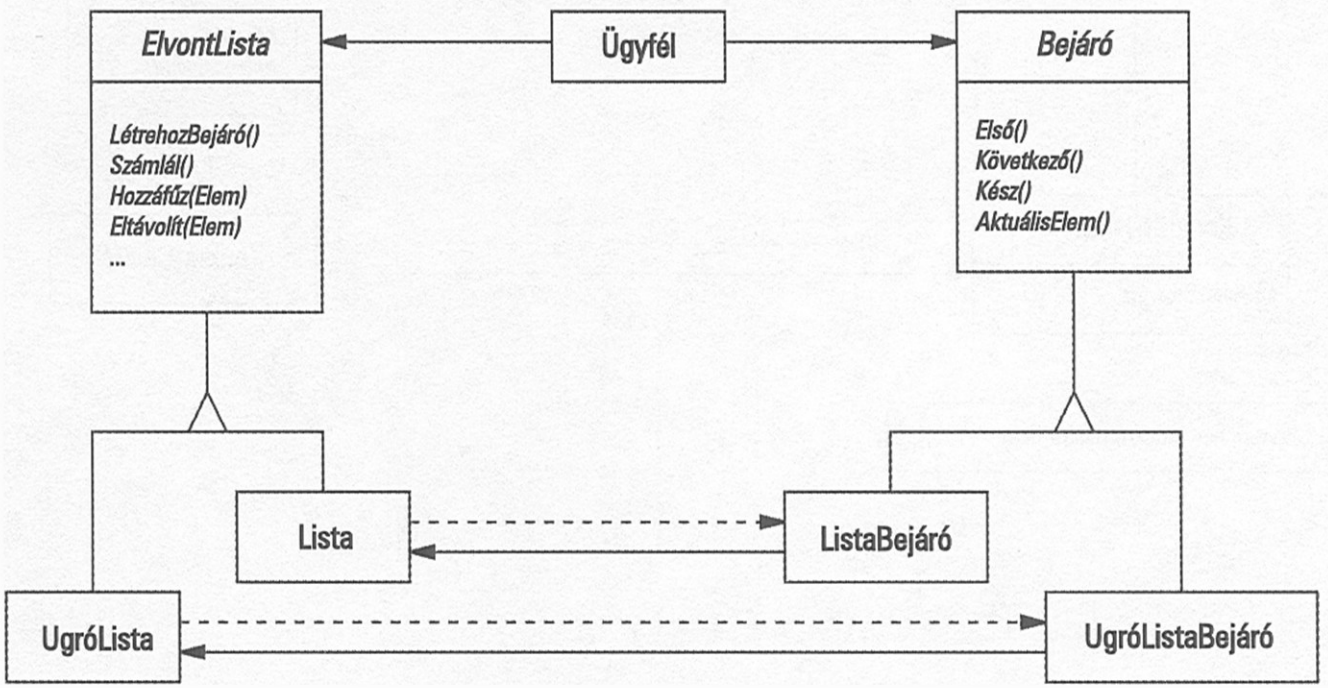
Mielőtt a ListaBejárót példányosíthatnánk, meg kell adnunk a bejárandó listát. Ha már rendelkezünk ListaBejáró példánnyal, a lista elemeit sorban érhetjük el. Az AktuálisElem (CurrentItem) művelet a lista aktuális elemét adja vissza, az Első (First) az első elemet teszi aktuális elemmé, a Következő (Next) pedig a következőt; a Kész (IsDone) feladata annak ellenőrzése, hogy túljutottunk-e már az utolsó elemen, vagyis hogy befejeztük-e a bejárást.

A bejárás elválasztása a Lista objektumtól lehetővé teszi, hogy a különböző bejárási módokhoz anélkül határozhassunk meg külön-külön bejárókat, hogy fel kellene sorolnunk azokat a Lista felületben. A SzűrőListaBejáró (FilteringListIterator) például csak azon elemekhez biztosítana hozzáférést, amelyek eleget tesznek bizonyos szűrési feltételeknek.

Megfigyelhetjük, hogy a bejáró és a lista összekapcsolódik, az ügyfélnek pedig tudnia kell, hogy egy *listát*, és nem más összesítő szerkezetet (aggregátumot) jár be. Ebből következik, hogy az ügyfél egy bizonyos típusú aggregátumhoz kötődik. Jobb lenne, ha az aggregátum osztályát az ügyfél kódjának megváltoztatása nélkül módosíthatnánk – ebben segít a bejárók általánosítása a **többalakú bejárás** támogatásához.

Példaként tegyük fel, hogy rendelkezünk egy UgróLista (SkipList) lista-megvalósítással is (az ugrólista [Pug90] olyan valószínűségi adatszerkezet, ami hasonlít a kiegyensúlyozott fákhoz), és mi olyan kódot szeretnénk írni, ami mind a Lista, mind az UgróLista objektumok esetében működik.

Először meghatározunk egy ElvontLista (AbstractList) nevű osztályt, ami a listák kezeléséhez közös felületet biztosít. Szükségünk lesz egy elvont Bejáró (Iterator) osztályra is, ami ugyanígy közös bejárási felületet ad. Ezután a különböző lista-megvalósításokhoz létrehozhatjuk a konkrét Bejáró alosztályokat. Mindennek eredménye a konkrét összesítő osztályoktól független bejárás lesz.



A kérdés azonban továbbra is az, hogyan hozzuk létre a bejárót. Mivel olyan kódot akarunk írni, ami független a konkrét Lista alosztályoktól, nem példányosíthatunk egyszerűen egy bizonyos osztályt. Ehelyett a lista objektumok felelősségi körébe utaljuk, hogy létrehozzák a nekik megfelelő bejárót. Ehhez egy LétrehozBejáró (CreateIterator) vagy hasonló művelet szükséges, amelyen keresztül az ügyfelek egy bejáró objektumot igényelhetnek.

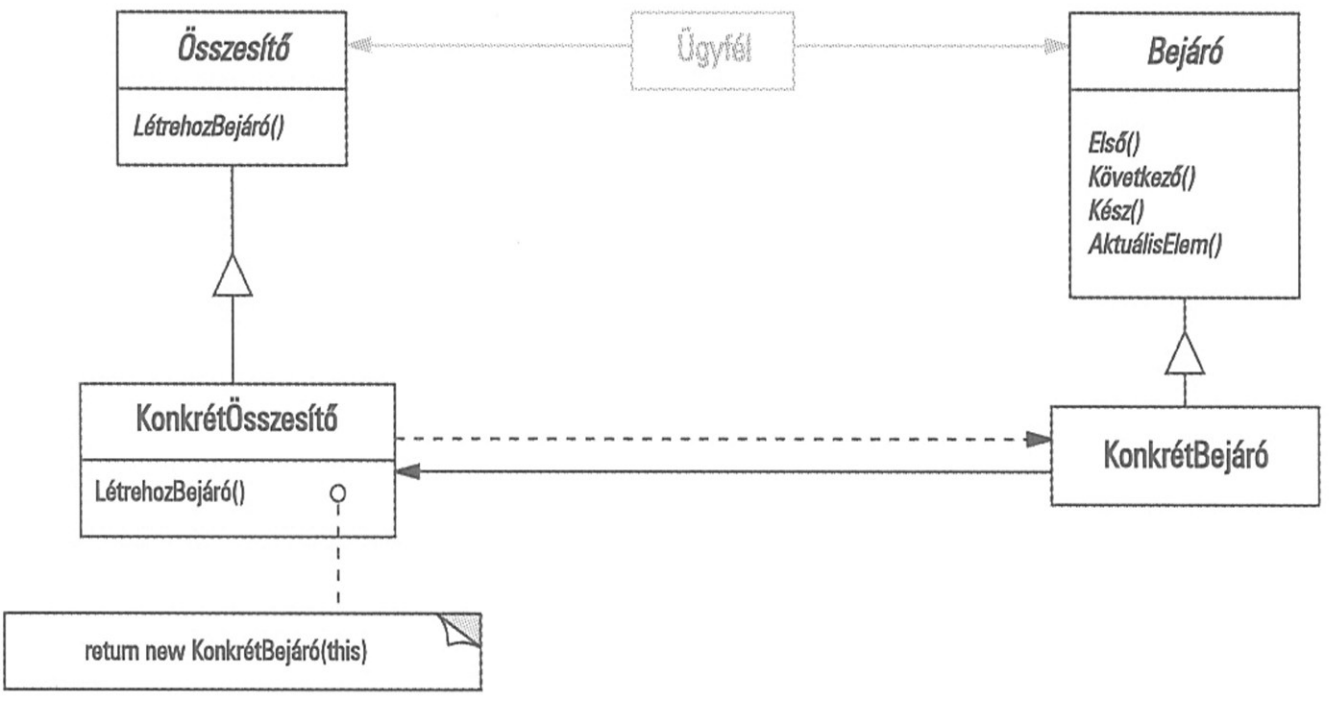
A LétrehozBejáró a Gyártófüggvény mintára mutat példát; itt arra használjuk, hogy az ügyfél a lista objektumtól a megfelelő bejárót kérhesse. A Gyártófüggvény alkalmazása két osztályhierarchiát eredményez; egyet a listák, egyet a bejárók számára. A LétrehozBejáró gyártófüggvény ezeket „köti össze".

Alkalmazhatóság

A Bejáró minta az alábbi esetekben alkalmazható:

- El szeretnénk érni egy összesítő (aggregát) objektum tartalmát, anélkül, hogy annak belső ábrázolását felfednénk.
- Több bejárási módot szeretnénk biztosítani összesítő objektumokhoz.
- Egységes felületet szeretnénk adni a különböző összesítő szerkezetek bejárásához (vagyis támogatni szeretnénk a többalakú bejárást).

Szerkezet



Résztvevők

- Bejáró
  - Felületet határoz meg elemek elérésére és bejárására.
- KonkrétBejáró
  - Megvalósítja a Bejáró (Iterator) felületet.
  - Számon tartja, melyik elemnél tartunk az összesítő objektum bejárása során.
- Összesítő
  - Felületet határoz meg a Bejáró objektumok létrehozására.
- KonkrétÖsszesítő
  - Megvalósítja a Bejáró-létrehozó felületet a megfelelő KonkrétBejáró (ConcreteIterator) példány visszaadásához.

Együttműködés

- A KonkrétBejáró számon tartja, melyik elemnél tartunk az összesítő objektum bejárása során, és képes kiszámítani, melyik a következő bejárandó objektum.

Következmények

A Bejáró mintának három lényeges előnye van:

1. *Támogatja az összesítő objektumok bejárásának különböző változatait.* Az összetett aggregátumok általában többféle módon is bejárhatók. A kód-előállítás vagy a jelentéstani ellenőrzés például elemzőfák bejárását igényli. Az első esetében a bejárás előrefelé vagy balról jobbra (inorder) is történhet. A bejárók segítségével a bejárási algoritmus könnyen módosítható, csak ki kell cserélnünk a bejárópéldányt egy másikra; új bejárási módszerek támogatásához pedig elég, ha új Bejáró alosztályokat határozunk meg.
2. *A bejárók egyszerűbbé teszik az Összesítő felületet.* A Bejáró bejárási felülete szükségtelenné teszi hasonló Összesítő felület létrehozását, így az összesítő felülete egyszerűsödik.
3. *Az összesítőn egyszerre több bejárás lehet folyamatban.* A bejárók számon tartják saját állapotukat (vagyis a bejárás állapotát), így egyszerre több bejárás is folyamatban lehet.

Megvalósítás

A Bejáró minta megvalósítására számos lehetőség kínálkozik; ezek közül az alábbiakban sorolunk fel néhányat. Előnyeik és hátrányaik többnyire az adott nyelv által biztosított vezérlési szerkezetektől függnek, egyes nyelvek (például a CLU [LG86]) azonban közvetlenül támogatják ezt a tervezési mintát.

1. *Ki vezérli a bejárást?* Alapvető kérdés annak eldöntése, melyik fél vezérelje a bejárást: a bejáró vagy az azt használó ügyfél? Amikor az ügyfél a vezérlő, a bejárót **külső bejárónak** hívjuk, amikor a bejáró, **belső bejárónak**[2]. A külső bejárót alkalmazó ügyfeleknek maguknak kell léptetniük a bejárást, és kifejezetten kérniük kell a következő elemet a bejárótól. A belső bejárónak ezzel szemben az ügyfél egy végrehajtandó műveletet ad át, amit az az összesítő minden elemére alkalmaz.
   A külső bejárók rugalmasabbak a belsőknél. Két gyűjtemény egyenlőségét például könnyen megvizsgálhatjuk egy külső bejáróval, míg ez belső bejáróval gyakorlatilag lehetetlen. A belső bejárók különösen az olyan nyelvekben gyengék, amelyek nem rendelkeznek névtelen függvényekkel, záradékokkal (closure, ilyen a C++), vagy „folytatásokkal" (continuations, ilyen a Smalltalk vagy a CLOS). Másrészről viszont a belső bejárók használata egyszerűbb, mert meghatározzák a bejárás logikáját.
2. *Ki határozza meg a bejárási algoritmust?* A bejáró nem az egyetlen hely, ahol a bejárási algoritmus meghatározható. Az összesítő meghatározhatja maga is, és a bejárót csupán arra használhatja, hogy a bejárás állapotát tárolja. A bejárók eme típusát **kurzornak** vagy sormutatónak hívjuk, miután feladata csak annyi, hogy az összesítő aktuális helyére mutasson. Az ügyfél meghívja a Következő (Next) műveletet az összesítőre a kurzorral mint argumentummal, a Következő pedig megváltoztatja a kurzor állapotát[3].
   Ha a bejárási algoritmusért a bejáró felel, könnyű ugyanazon az összesítőn különböző bejárási algoritmusokat alkalmazni, és ugyanannak az algoritmusnak a használata különböző összesítőkön is egyszerűbb. Más részről a bejárási algoritmusnak szüksége lehet arra, hogy elérje az összesítő privát változóit, mely esetben az algoritmusnak a bejáróba helyezése megsérti az összesítő egységbe zárását.
3. *Mennyire ellenálló a bejáró?* Az összesítő módosítása a bejárás közben veszélyes lehet. Ha ekkor elemeket adunk hozzá vagy veszünk el belőle, előfordulhat, hogy egy elemet kétszer érünk el, vagy teljesen kihagyjuk. A legegyszerűbb megoldás, ha másolatot készítünk az összesítőről, és a másolatot járjuk be, de ez általában túl költséges.
   Egy **ellenálló bejáró** biztosítja, hogy a hozzáadás és eltávolítás nem befolyásolja a bejárást, és ezt anélkül éri el, hogy másolatot készítene az összesítőről. Az ilyen bejárók megvalósítására számos mód kínálkozik. A legtöbbnek az az alapja, hogy a bejárót bejegyeztetjük az összesítő számára. Elem beszúrásakor vagy eltávolításakor az összesítő módosítja az általa alkalmazott bejárók belső állapotát, vagy a helyes bejárás biztosításához belsőleg tárolja a szükséges információt.
   Kofler [Kof93] kitűnő leírást ad az ellenálló (robusztus) bejárók megvalósításáról az ET++-ban, míg Murray [Mur93] a megvalósítást a USL StandardComponents List osztályával kapcsolatban tárgyalja.

---

[2] Booch **aktív** és **passzív** bejáróként hivatkozik rájuk [Boo94]. Ezek a fogalmak az ügyfél szerepére, és nem a bejáró aktivitásának fokára vonatkoznak.

[3] A kurzorok az Emlékeztető mintára adnak egyszerű példát, és megvalósításuk is sok szempontból hasonló.

4. *További bejáró műveletek.* A lehetséges legkisebb Bejáró felület az Első, Következő, Kész és AktuálisElem (First, Next, IsDone, CurrentItem) műveletekből áll[4]. Emellett azonban további műveletek is hasznosak lehetnek. A rendezett aggregátumok például rendelkezhetnek egy Előző (Previous) művelettel, ami a bejárót a megelőző elemre állítja. Egy UgrásIde (SkipTo) művelet a rendezett vagy sorszámozott (indexelt) gyűjtemények esetében jöhet jól. Egy ilyen művelettel a bejárót egy olyan objektumra állíthatjuk, amely megfelel bizonyos követelményeknek.
5. *Többalakú bejárók használata a C++-ban.* A többalakú bejáróknak megvan a maguk ára. Azt igénylik, hogy a bejáró objektumot egy gyártófüggvénnyel dinamikusan hozzuk létre. Emiatt használatukat célszerű azokra az esetekre korlátozni, amikor tényleg szükség van a többalakúságra. Más esetekben alkalmazzunk inkább konkrét bejárókat, amelyeket a verembe helyezhetünk.
   A többalakú bejáróknak van egy másik hátulütőjük is: törlésükért az ügyfél felelős. Az ilyen esetekben nagy a hibalehetőség, hiszen könnyen megfeledkezhetünk a kupacra helyezett bejáró objektum megsemmisítéséről, ha már nincs rá szükség. Ez különösen akkor valószínű, ha egy műveletben több kilépési pont van; ha valahol kivétel lép fel, a bejáró objektum által elfoglalt hely felszabadítására soha nem kerül sor.
   Gyógyírt a Helyettes tervezési minta kínál. Ekkor a tényleges bejáró helyett egy veremfoglalású helyettest használunk, amely destruktorában törli a bejárót, így amikor a helyettes kikerül a hatókörből, vele együtt az „igazi" bejáró is megsemmisül. A helyettes megfelelő takarítást biztosít, még kivételek fellépése esetén is. A megoldás nem más, mint a jól ismert C++-elv, „az erőforrás-foglalás egyben előkészítés (inicializálás)" [ES90] alkalmazása. További részleteket a Példakód részben láthatunk.
6. *A bejárók kivételezett hozzáféréssel rendelkezhetnek.* A bejárókra úgy is tekinthetünk, mint az őket létrehozó összesítők bővítéseire. A bejáró és az összesítő szoros csatolásban állnak egymással; ezt a kapcsolatot a C++-ban például úgy fejezhetjük ki, ha a bejárót összesítője „barátjává" (`friend`) tesszük. Ekkor az összesítőben nem kell olyan műveleteket meghatároznunk, amelyek egyetlen célja, hogy lehetővé tegyék a bejárók számára a bejárás hatékony megvalósítását.
   Mindazonáltal az ilyen kivételezett hozzáférés az új bejárási módok megadását megnehezíti, mert az új barát hozzáadása az összesítő felületének módosítását igényli. Ennek elkerülésére a Bejáró osztály védett (`protected`) műveleteket tartalmazhat, melyeken keresztül az összesítő fontos, de nyilvánosan nem elérhető tagjaihoz férhetünk hozzá. A Bejáró alosztályai (és *kizárólag* azok) ezekkel a védett műveletekkel nyerhetnek kivételezett hozzáférést az összesítő objektumhoz.

[4] A felületet még kisebbé tehetjük, ha a Következő, Kész és AktuálisElem műveleteket egyetlen műveletben egyesítjük, amely a következő objektumra lép és visszaadja azt. Ha a bejárás befejeződött, a művelet egy különleges értékkel tér vissza (például 0-val), ami a bejárás végét jelzi.

7. *Bejárók összetételek számára.* A külső bejárók megvalósítása az olyan önhívó összesítő szerkezetek (rekurzív aggregátumok) esetében, mint amilyeneket az Összetétel mintában találhatunk, nehéz lehet, mert a szerkezet egy adott pozíciója a beágyazott aggregátumok számos szintjét átfoghatja. Ekkor a külső bejárónak tárolnia kell az összetételen keresztül vezető útvonalat, hogy megállapíthassa, melyik az aktuális objektum. Ennél sokszor egyszerűbb egy belső bejáró használata, ami az aktuális helyet egyszerűen önmaga ismételt meghívásával képes rögzíteni, s így egyben tárolni az útvonalat a hívási veremben.
   Ha az adott összetétel csomópontjai rendelkeznek egy felülettel, ami a testvérekre, szülőkre vagy gyermekekre ugrást szolgálja, egy kurzor alapú bejáró jobb megoldás lehet. A kurzornak csak az aktuális csomópontot kell számon tartania, az összetétel bejárásához támaszkodhat az említett felületre.
   Az összetételeket gyakran többféleképpen szükséges bejárni. A leggyakoribb az előre haladó, a visszafelé haladó, a balról jobbra haladó (inorder) és a szélességi bejárás (horizontális bejárás, breadth-first). Ezeket külön bejáró osztályokkal támogathatjuk.
8. *Null bejárók.* A **NullBejáró** olyan csökevényes bejáró, amely a korlátfeltételek kezelésénél jöhet jól. Meghatározása szerint a NullIterator *mindig* végzett a bejárással, vagyis Kész (IsDone) művelete mindig true-ra értékelődik ki.
   A NullBejáró a faszerkezetű aggregátumok (amilyenek az Összetételek is) bejárását teheti könnyebbé. A bejárás minden pontján elkérjük az aktuális elem gyermekeit; az összesítő elemek a szokott módon egy konkrét bejárót adnak vissza, a levélelemek azonban egy NullBejáró példányt. Ezzel a megoldással a teljes szerkezetet egységesen járhatjuk be.

Példakód

Egy egyszerű Lista osztály (List) megvalósítását fogjuk megnézni, amely része az alapkönyvtárunknak (lásd a C függeléket). Két Bejáró-megvalósítást mutatunk be: egyet a lista előre haladó bejárásához, egyet pedig a visszafelé haladáshoz. (Az alapkönyvtár csak az elsőt támogatja.) Ezután megmutatjuk, hogyan kell használni ezeket a bejárókat, és hogyan kerülhetjük el, hogy egy adott megvalósításhoz kötődjünk. Ezt követően némi módosítást hajtunk végre, hogy gondoskodhassunk a bejárók megfelelő törléséről. Az utolsó példa egy belső bejárót mutat be, és összehasonlítja azt külső megfelelőjével.

1. *A List és Iterator felületek.* Először tekintsük meg a List felületnek azt a részét, amelyik a bejárók megvalósítása szempontjából lényeges. A teljes felületet a C függelékben találhatjuk meg.

```
template <class Item>
class List {
public:
        List(long size = DEFAULT_LIST_CAPACITY);

        long Count() const;
```

```
        Item& Get(long index) const;
        //...
};
```

A `List` osztály nyilvános felületén keresztül meglehetősen hatékony módot ad a bejárás támogatására; mindkét típusú bejárás megvalósításához megfelel. Így nincs szükség arra, hogy a bejáróknak kivételezett hozzáférést adjunk a háttérben megbúvó adatszerkezethez, vagyis a bejáró osztályok nem barátjai (`friend`) a `List` osztálynak. A különböző bejárások észrevétlen használatát úgy támogatjuk, hogy létrehozunk egy elvont `Iterator` osztályt, amely meghatározza a bejáró felületet.

```
template <class Item>
class Iterator {
public:
        virtual void First() = 0;
        virtual void Next() = 0;
        virtual bool IsDone() const = 0;
        virtual Item CurrentItem() const = 0;
protected:
        Iterator();
};
```

2. *Az Iterator alosztályainak megvalósítása.* A `ListIterator` (ListaBejáró) az `Iterator` alosztálya.

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
        ListIterator(const List<Item>* aList);
        virtual void First();
        virtual void Next();
        virtual bool IsDone() const;
        virtual Item CurrentItem() const;

private:
        const List<Item>* _list;
        long _current;
};
```

A `ListIterator` megvalósítása egyszerű. A `List`-et a `_current` indexszel együtt tárolja:

```
template <class Item>
ListIterator<Item>:: ListIterator (
        const List<Item>* aList
) : _list(aList), _current(0) {
}
```

A `First` (Első) az első elemre állítja a bejárót:

```
template <class Item>
void ListIterator<Item>::First () {
        _current = 0;
}
```

A `Next` (Következő) a következő elemre léptet:

```
template <class Item>
void ListIterator<Item>::Next () {
        _current++;
}
```

Az `IsDone` (Kész) ellenőrzi, hogy az index a listán belüli elemre hivatkozik-e:

```
template <class Item>
void ListIterator<Item>::IsDone () const {
        return _current >= _list->Count();
}
```

Végül a `CurrentItem` (AktuálisElem) az aktuális index által jelzett helyen levő elemet adja vissza. Ha a bejárás már befejeződött, `IteratorOutOfBounds` kivételt váltunk ki:

```
template <class Item>
Item ListIterator<Item>::CurrentItem () const {
        if (IsDone()) {
                throw IteratorOutOfBounds;
        }
        return list->Get(current);
}
```

A visszafelé haladó `ReverseListIterator` megvalósítása ugyanilyen, de annak `First` művelete a `_current` indexet a lista végére állítja, a `Next` pedig az első elem irányába csökkenti a `_current` értékét.

3. *A bejárók használata.* Tételezzük fel, hogy van egy listánk, amely `Employee` (Alkalmazott) objektumokat tartalmaz, és az összes alkalmazottat ki szeretnénk íratni. Ezt az `Employee` osztály a `Print` (Kiír) művelettel támogatja. A lista kiírásához a `PrintEmployees` (KiírAlkalmazottak) műveletet határozzuk meg, amelynek argumentuma egy bejáró, amellyel a művelet bejárja és kiírja a listát.

```
void PrintEmployees (Iterator<Employee*>& i) {
        for (i.First(); !i.IsDone(); i.Next()) {
                i.CurrentItem()->Print();
        }
}
```

Mivel mind előre haladó, mind visszafelé haladó bejáróval rendelkezünk, a fenti művelet újrahasznosításával mindkét sorrendben kiírathatjuk az alkalmazottakat.

```
List<Employee*>* employees;
//...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

4. *Egy adott lista-megvalósításhoz való kötődés elkerülése.* Vizsgáljuk meg, hogyan érintené bejáró kódunkat, ha a `List`-et ugrólistaként valósítanánk meg. A `List` `SkipList` alosztályának biztosítania kell egy `SkipListIterator`-t (UgróListaBejáró), amely megvalósítja az `Iterator` felületet. A `SkipListIterator`-nak a bejárás hatékony végrehajtásához nem elég csupán egy indexet fenntartania, de mivel az osztály megfelel az `Iterator` felületnek, a `PrintEmployees` művelet akkor is használható, ha az alkalmazottakat egy `SkipList` objektumban tároljuk.

```
SkipList<Employee*>* employees;
//...

SkipListIterator<Employee*> iterator(employees);
PrintEmployees(iterator);
```

Bár ez a megközelítés működik, jobb lenne, ha nem kötődnénk egy bizonyos `List`-megvalósításhoz (ebben az esetben konkrétan a `SkipList`-hez). Bevezethetnénk egy `AbstractList` (ElvontLista) nevű elvont osztályt, amellyel szabványosíthatjuk a listafelületet a különböző lista-megvalósításokhoz. Ekkor a `List` és a `SkipList` az `AbstractList` alosztályai lennének.
A többalakú bejárás lehetővé tételéhez az `AbstractList` a `CreateIterator` (LétrehozBejáró) gyártófüggvényt határozhatja meg, amit az alosztályok felülírnak, hogy visszaadják a megfelelő bejárót:

```
template<class Item>
class AbstractList {
public:
        virtual Iterator<Item>* CreateIterator() const = 0;
        //...
};
```

Egy másik lehetőség, hogy egy általános mixin osztályt határozunk meg `Traversable` (Bejárható) néven, ami a bejárók létrehozásának felületét írja le. Az összesítő osztályok így a `Traversable` „bekeverésével" támogathatják a többalakú bejárást.
A `List` a `CreateIterator` felülbírálásával egy `ListIterator` objektumot ad vissza:

```
template<class Item>
Iterator<Item>* List<Item>::CreateIterator() const {
        return new ListIterator<Item>(this);
}
```

Most már abban a helyzetben vagyunk, hogy konkrét megvalósítástól független kódot írhatunk az alkalmazottak listájának kiíratására.

```
// csak azt tudjuk, hogy van egy AbstractList-ünk
AbstractList<Employee*>* employees;
//...

Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

5. *A bejárók törlésének biztosítása.* Figyeljük meg, hogy a `CreateIterator` egy újonnan létrehozott bejáró objektumot ad vissza, melynek törléséért mi felelünk. Ha elfelejtjük, memóriaszivárgást idézhetünk elő. Az ügyfelek dolgát megkönnyítendő biztosítunk egy `IteratorPtr` (BejáróMutató) nevű mutatót, ami bejáróhelyettesként viselkedik, és gondoskodik az `Iterator` objektum megsemmisítéséről, ha arra nincs többé szükség.
   Az `IteratorPtr`-nek mindig a veremben foglalunk helyet.[5] A C++ automatikusan meghívja majd a destruktorát, amely törli magát a bejárót. Az `IteratorPtr` túlterheli mind az `operator->`, mind az `operator*` műveleteket, így pontosan egy bejárót címző mutatóként kezelhető. Az `IteratorPtr` tagjainak megvalósítása helyben kifejtett (inline), így nem okoznak többletterhet.

```
template<class Item>
class IteratorPtr {
public:
        IteratorPtr(Iterator<Item>* i): _i(i) { }
        ~IteratorPtr() { delete _i; }
        Iterator<Item>* operator->() { return _i; }
        Iterator<Item>& operator*() { return *_i; }
private:
        // a másolás és hozzárendelés tiltása, hogy
        // elkerüljük az _i többszöri törlését

        IteratorPtr(const IteratorPtr&);
        IteratorPtr& operator=(const IteratorPtr&);
private:
        Iterator<Item>* _i;
};
```

   Az `IteratorPtr` lehetővé teszi, hogy egyszerűsíthessük a kiíró kódot:

```
AbstractList<Employee*>* employees;
//...

IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```

6. *Egy belső listabejáró.* Utolsó példaként nézzük meg egy belső vagy passzív `List-Iterator` osztály egy lehetséges megvalósítását. Itt a bejáró vezérli a bejárást, és minden elemen végrehajt egy műveletet.
   A kérdés ebben az esetben az, hogyan adjuk át a bejárónak paraméterként az elemeken végrehajtandó műveleteket. A C++ nem támogatja a névtelen függvényeket vagy záradékokat (closure), amelyek más nyelvekben e feladatra rendelkezésre állnak. Legalább két lehetőségünk azért akad: (1) egy függvényre hivatkozó (globális vagy statikus) mutatóban átadni az információt, vagy (2) alosztályokra támaszkodni.

[5] Ezt fordításkor egyszerűen biztosíthatjuk, ha privát `new` (új) és `delete` (töröl) műveleteket vezetünk be. Megvalósítás nem szükséges hozzájuk.

Az első esetben a bejáró minden elemnél meghívja a számára átadott műveletet, a másodikban egyetlen műveletet hív meg, amelyet a megfelelő viselkedés biztosítása érdekében az alosztályok felülbírálnak.
Egyik megoldás sem tökéletes. A bejárás közben gyakran szükség lehet az állapot nyomon követésére, a függvények pedig nem igazán alkalmasak erre; az állapotot statikus változókkal kellene számon tartanunk. Egy `Iterator` alosztályban kényelmesen elhelyezhetjük az állapotinformációt, mondjuk egy példányváltozóban, de a különböző bejárásokhoz külön-külön alosztályt létrehozni rengeteg munkát igényel.
Íme a második, alosztályokat használó megoldás vázlata. A belső bejáró neve itt `ListTraverser`.

```
template<class Item>
class ListTraverser {
public:
        ListTraverser(List<Item>* aList);
        bool Traverse();
protected:
        virtual bool ProcessItem(const Item&) = 0;
private:
        ListIterator<Item> _iterator;
};
```

A `ListTraverser` paraméterként egy `List` példányt kap, és egy külső `ListIterator` segítségével hajtja végre a bejárást. A `Traverse` elkezdi a bejárást, és minden elemre meghívja a `ProcessItem` (FeldolgozElem) műveletet. A belső bejáró a bejárást azzal fejezheti be, hogy `false` értéket ad vissza a `ProcessItem`-ből. A `Traverse` tájékoztat, ha a bejárás idő előtt befejeződött.

```
template<class Item>
ListTraverser<Item>::ListTraverser (
        List<Item>* aList
) : _iterator(aList) { }

template<class Item>
bool ListTraverser<Item>::Traverse () {
        bool result = false;

        for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
        ) {
                result = ProcessItem(_iterator.CurrentItem());

                if (result == false) {
                        break;
                }
        }
        return result;
}
```

A `ListTraverser` segítségével írassuk ki az első tíz alkalmazottot a listából. Ehhez alosztályokat kell származtatnunk a `ListTraverser`-ből, és felül kell bírálnuk a `ProcessItem` műveletet. A kiírt alkalmazottak számát a `_count` példányváltozóban számláljuk meg.

```
class PrintNEmployees : public ListTraverser<Employee*> {
public:
        PrintNEmployees(List<Employee*>* aList, int n) :
                ListTraverser<Employee*>(aList),
                _total(n), _count(0) { }

protected:
        bool ProcessItem(Employee* const&);
private:
        int _total;
        int _count;
};

bool PrintNEmployees::ProcessItem (Employee* const& e) {
        _count++;
        e->Print();
        return _count < _total;
}
```

A `PrintNEmployees` az alábbi módon írja ki az első tíz alkalmazott nevét:

```
List<Employee*>* employees;
//...

PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Megfigyelhetjük, hogy az ügyfél nem határozza meg a bejárási ciklust; a teljes bejárási logika újrahasznosítható. Ez a belső bejárók legfontosabb haszna. Kicsit több munkát igényel ugyan, mint a külső bejárók használata, mert meg kell határoznunk egy új osztályt. Vessük össze ezt egy külső bejáró alkalmazásával:

```
ListIterator<Employee*> i(employees);
int count = 0;

for (i.First(); !i.IsDone(); i.Next()) {
        count++;
        i.CurrentItem()->Print();

        if (count >= 10) {
                break;
        }
}
```

A belső bejárók többféle bejárást zárhatnak egységbe. `FilteringListTraverser` (SzűrőListaBejáró) néven például egy olyan bejárást adhatunk meg, amely csak azokat az elemeket dolgozza fel, amelyek megfelelnek bizonyos követelményeknek:

```
template<class Item>
class FilteringListTraverser {
public:
        FilteringListTraverser(List<Item>* aList);
        bool Traverse();
protected:
        virtual bool ProcessItem(const Item&) = 0;
        virtual bool TestItem(const Item&) = 0;
private:
        ListIterator<Item> _iterator;
};
```

Ez a felület megegyezik a `ListTraverser`-ével, azzal a különbséggel, hogy hozzáadtuk a követelményeket ellenőrző `TestItem` (TesztelElem) tagfüggvényt. Az alosztályok ennek felülbírálásával hajtják végre az ellenőrzést.
A `Traverse` az ellenőrzés eredményének függvényében dönti el, hogy folytatja-e a bejárást:

```
template<class Item>
void FilteringListTraverser<Item>::Traverse() {
        bool result = false;

        for (
          _iterator.First();
          !_iterator.IsDone();
          _iterator.Next()
        ) {

          if (TestItem(_iterator.CurrentItem())) {
                result = ProcessItem(_iterator.CurrentItem());
                if (result == false) {
                      break;
                }
          }
        }
        return result;
}
```

A fenti osztály egy másik változatában úgy is meghatározhatnánk a `Traverse` műveletet, hogy már akkor is eredményt adjon, ha legalább egy elem teljesíti a követelményeket.[6]

[6] A `Traverse` művelet ezekben a példákban Sablonfüggvény, melynek alapműveletei a `TestItem` és a `ProcessItem`.

Ismert felhasználások

A bejárók hétköznapinak számítanak az objektumközpontú rendszerekben; a legtöbb gyűjteményosztály-könyvtár kínál bejárókat valamilyen formában.

Íme egy példa erre a Booch komponensek közül [Boo94], amely egy népszerű gyűjteményosztály-könyvtár. A könyvtárban megtalálható a sor (queue) rögzített méretű (korlátos) és dinamikusan növekvő (nem korlátos) megvalósítása is. Felületét az elvont Queue osztály írja le. A különböző megvalósításokhoz a könyvtár úgy biztosítja a többalakú bejárást, hogy a sorbejárót az elvont Queue osztályfelületre alapozva valósítja meg. Ennek az az előnye, hogy nincs szükség gyártófüggvényre, ami a sor-megvalósításoktól elkéri a megfelelő bejárót. Természetesen az elvont Queue osztály felületének elegendő szolgáltatást kell biztosítania ahhoz, hogy a bejárót hatékonyan megvalósíthassuk.

A Smalltalk nem igényli a bejárók kifejezett meghatározását. A szabványos gyűjteményosztályok (Bag, Set, Dictionary, OrderedCollection, String stb.) mind meghatároznak egy belső bejáró metódust (`do:`), amely argumentumként egy blokkot (vagyis záradékot) kap. A gyűjtemény minden eleme kapcsolódik a blokkban levő helyi változóhoz, majd a blokk végrehajtódik. A Smalltalk emellett Stream (Folyam) osztályokat is tartalmaz, amelyek támogatnak egy, a bejárókhoz hasonló felületet. A ReadStream (OlvasFolyam) lényegében véve egy bejáró, és minden soros elérésű gyűjtemény külső bejárójaként alkalmazható. A nem soros elérésű gyűjtemények (pl. Set, Dictionary) számára nem áll rendelkezésre szabványos külső bejáró.

A korábban leírt takarító helyettest és többalakú bejárókat az ET++ tároló osztályai biztosítják [WGM88]. A Unidraw grafikus szerkesztő keretrendszeri osztályai kurzor alapú bejárókat használnak [VL90].

Az ObjectWindows 2.0 [Bor94] bejárók egész osztályhierarchiáját kínálja a tárolókhoz; segítségükkel a különböző tárolótípusok ugyanúgy járhatók be. A bejárás itt az utótagos növelő művelet (++) túlterhelésén alapul, amely előre lépteti a bejárót.

Kapcsolódó minták

Összetétel: Bejárókat gyakran alkalmazunk az olyan önhívó szerkezetek esetében, mint az Összetételek.

Gyártófüggvény: A többalakú bejárók a megfelelő Bejáró alosztályok példányosításához gyártófüggvényekre támaszkodnak.

Emlékeztető: Gyakran alkalmazzák együtt a Bejáró mintával. A bejárók belsőleg tárolt „emlékeztetők" segítségével rögzíthetik a bejárás állapotát.

Közvetítő

Viselkedési objektumminta

Egyéb nevek

Mediator

Cél

A cél meghatározni egy objektumot, amely objektumok egy halmazának együttműködését irányítja. (Vagyis ezeket egyetlen objektumba tokozzuk be.) A módszerrel laza csatolást hozunk létre, amelyben az egyes objektumok közvetlenül nem hivatkozhatnak egymásra, a köztük levő kapcsolatok pedig egymástól függetlenül módosíthatók.

Feladat

Az objektumközpontúság arra ösztönöz, hogy a kívánt viselkedést objektumok között osszuk szét. A viselkedés elosztása viszont olyan objektumszerkezetet eredményezhet, amelyben az objektumok között számtalan kapcsolat létezik – a legrosszabb esetben minden objektum tudni fog minden objektumról. Bár egy rendszer több objektumra való felosztása általában növeli az újrahasznosíthatóságot, a kapcsolatok szövevénye csökkenti azt. Ha egy objektum számos másikkal áll bonyolult kapcsolatban, valószínűbb, hogy azok segítsége nélkül nem képes ellátni a feladatát, így a rendszer „tömbszerűen" viselkedik. Sőt, a rendszer működésének jelentősebb megváltoztatása is nehezebbé válik, hiszen a viselkedést számos objektum között osztottuk szét, így arra kényszerülhetünk, hogy a rendszer működésének testreszabásához sok-sok alosztályt kelljen létrehoznunk.

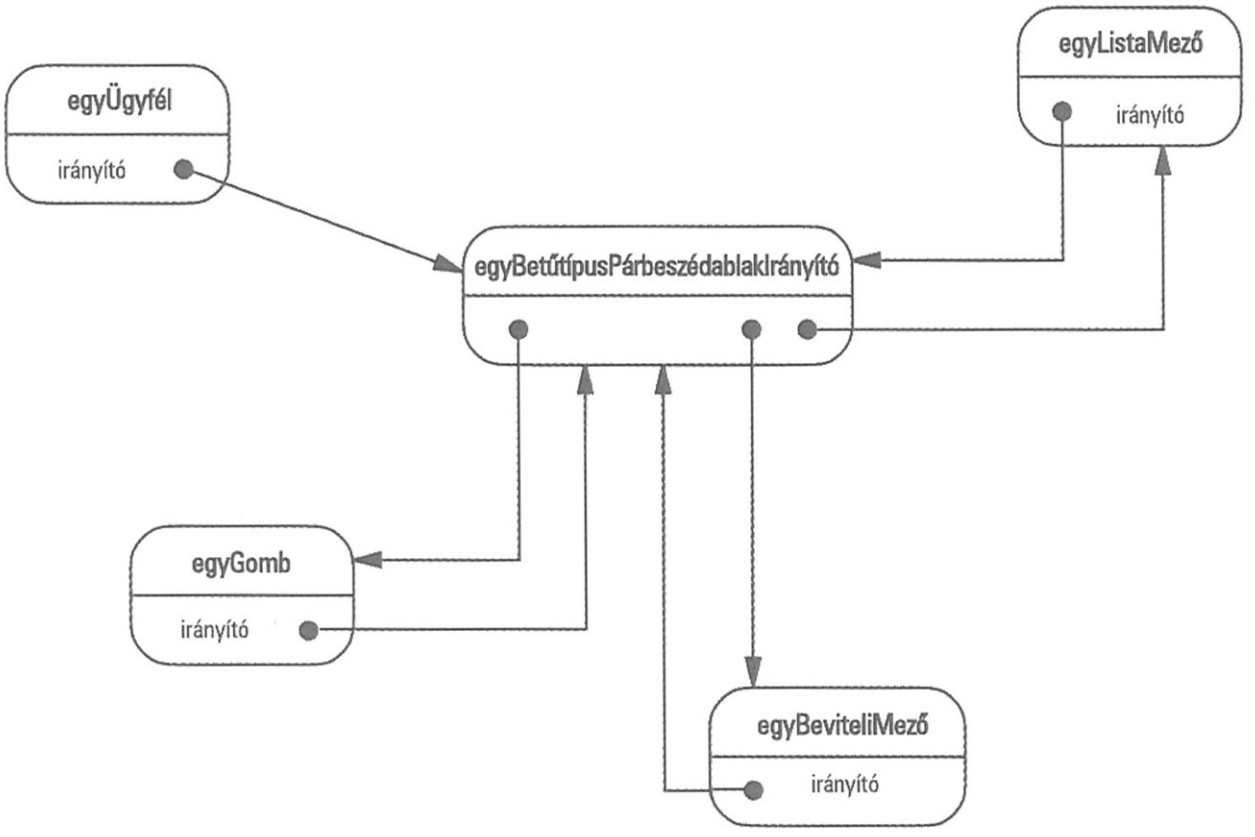
Példaként vegyük a párbeszédablakok megvalósítását egy grafikus felhasználói felületen. A párbeszédablak az alábbi ábrán bemutatotthoz hasonló módon egy ablak segítségével kínál fel számos grafikus vezérlőelemet, például gombokat, menüket vagy beviteli mezőket:

Az ablak vezérlői között gyakran függőségek állnak fenn, például valamelyik gombot le kell tiltanunk, ha egy adott beviteli mező üres, vagy meg kell változtatnunk a mező tartalmát, ha a felhasználó kiválaszt egy elemet egy listamezőből. Ezek fordítottja is előfordulhat, vagyis a mezőbe írás is automatikusan kiválthatja a megfelelő elem vagy elemek kijelölését a listamezőben, illetve egyes gombok elérhetővé válhatnak, hogy a felhasználó számára lehetővé tegyék, hogy felhasználhassa a beírt szöveget, mondjuk módosítsa vagy törölje azt a dolgot, amire a szöveg hivatkozik.

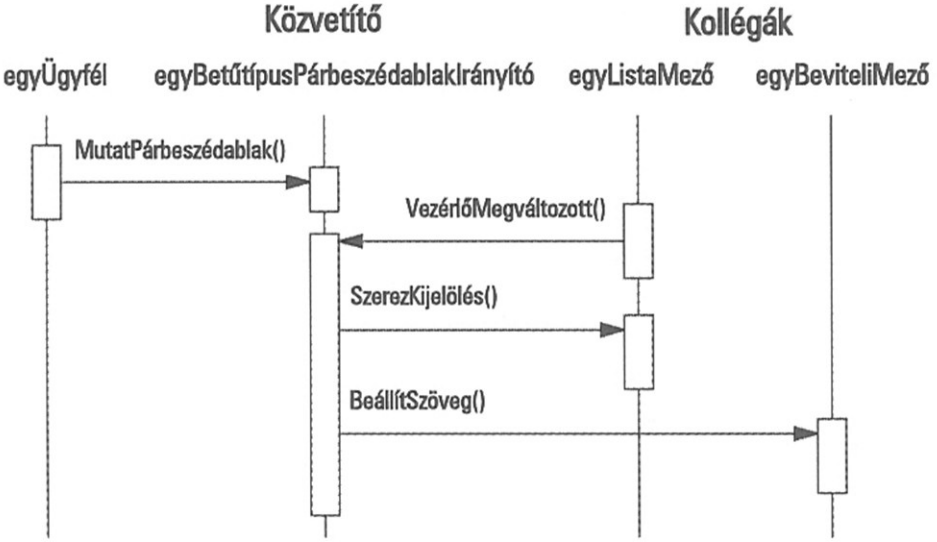
A különböző párbeszédablakokban a vezérlők között különféle kapcsolatok állhatnak fenn. Így bár ezek az ablakok általában hasonló elemeket tartalmaznak, nem használhatják fel automatikusan a vezérlőosztályokat; előbb azok testreszabására van szükség, hogy tükrözzék az adott párbeszédablakban fennálló függőségeket. Az alosztályok létrehozásával történő egyedi testreszabás mindazonáltal kimerítő lehet, hiszen rengeteg osztályról van szó.

Mindeme problémákat úgy kerülhetjük el, ha a közös viselkedést egy önálló **közvetítő** objektumba zárjuk. A közvetítő feladata, hogy vezérelje és összehangolja objektumok egy csoportjának együttműködését. A közvetítő olyan köztes rétegként szolgál, amely megakadályozza, hogy a csoport objektumai közvetlenül hivatkozzanak egymásra. Az objektumok csak a közvetítőt ismerik, így a keresztkapcsolatok száma jelentősen lecsökken.

Például létrehozunk egy közvetítőt **BetűtípusPárbeszédablakIrányító** (FontDialogDirector) néven, amely egy betűtípus-választó párbeszédablak vezérlőit hangolja össze. A FontDialogDirector ismeri valamennyi vezérlőt, így kommunikációs elosztóként viselkedik a vezérlők számára:



Az alábbi együttműködési diagram azt illusztrálja, hogyan működnek együtt az objektumok, ha a listamezőben kijelölt elem módosul:

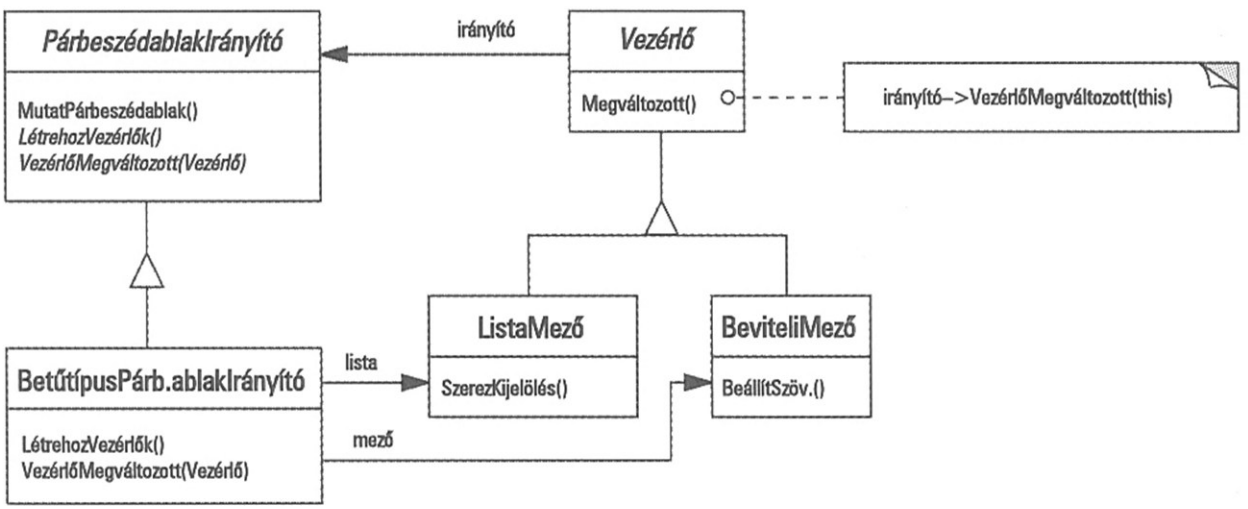


Íme az események sorrendje, melyek során a listamezőben kijelölt elem átadódik a beviteli mezőnek:

1. A listamező közli az irányítóval (director), hogy megváltozott.
2. Az irányító elkéri a kijelölt elemet a listától.
3. Az irányító átadja a kijelölt elemet a beviteli mezőnek.
4. Most, hogy a beviteli mező szöveget tartalmaz, az irányító elérhetővé teszi a műveletek kezdeményezésére („félkövér", „dőlt") szolgáló gombokat.

Figyeljük meg, hogyan közvetít az irányító a lista és a beviteli mező között. A vezérlők egymással nem társalognak közvetlenül, csak közvetetten, az irányítón keresztül. Nem kell tudniuk egymásról, csak az irányítót kell ismerniük. Továbbá, mivel a viselkedést egyetlen osztály tartalmazza, az eme egyetlen osztály bővítésével vagy kicserélésével módosítható.

Az alábbi ábrán azt láthatjuk, hogyan építhető be a BetűtípusPárbeszédablakIrányító elvont szerkezete egy osztálykönyvtárba:



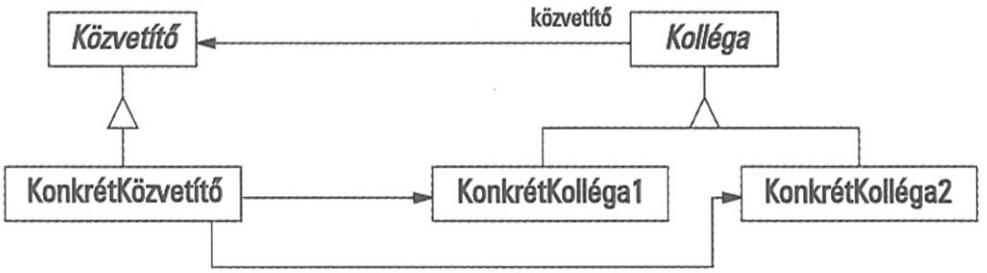
A PárbeszédablakIrányító (DialogDirector) egy elvont osztály, amely a párbeszédablakok általános viselkedését írja le. Az ügyfelek a MutatPárbeszédablak (ShowDialog) művelet meghívásával megjeleníthetik az ablakot a képernyőn. A LétrehozVezérlők (CreateWidgets) elvont művelet az ablak vezérlőinek létrehozására szolgál. A VezérlőMegváltozott (WidgetChanged) egy másik elvont művelet, amelyet a vezérlők akkor hívnak meg, amikor tájékoztatni akarják az irányítót, hogy állapotuk megváltozott. A PárbeszédablakIrányító alosztályai a LétrehozVezérlők felülírásával hozzák létre a megfelelő vezérlőket, a változások kezeléséhez pedig a VezérlőMegváltozott műveletet bírálják felül.

Alkalmazhatóság

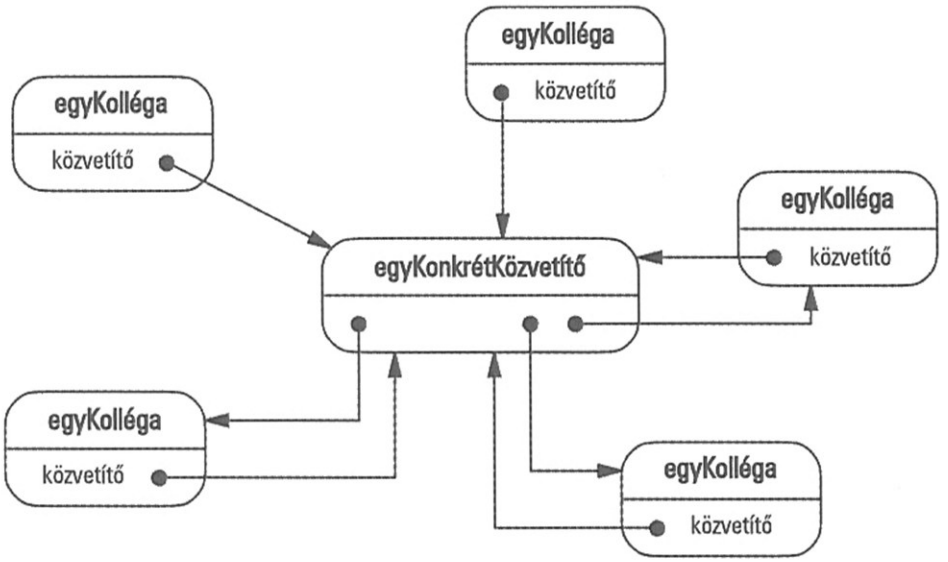
A Közvetítő minta alkalmazása a következő esetekben célszerű:

- Objektumok egy halmaza jól meghatározott, de bonyolult módon kommunikál egymással. Az előálló kölcsönös függőségek szerkezete esetleges és nehezen átlátható.
- Egy objektum újrahasznosítása nehéz, mert számos más objektumra hivatkozik, illetve számos más objektummal tart kapcsolatot.
- Több osztály között elosztott viselkedést kellene alosztályok sokasága nélkül testreszabnunk.

Szerkezet



Egy jellegzetes objektumszerkezet így nézhet ki:



Résztvevők

- **Közvetítő** (PárbeszédablakIrányító)
  - Felületet határoz meg a Kolléga (Colleague) objektummal való kapcsolattartáshoz.
- **KonkrétKözvetítő** (BetűtípusPárbeszédablakIrányító)
  - A Kolléga objektumok összehangolásával kialakítja az együttműködést.
  - Ismeri kollégáit és gondoskodik róluk.
- **Kolléga osztályok** (ListaMező, BeviteliMező)
  - Minden Kolléga osztály ismeri a Közvetítő (Mediator) objektumát.
  - A kollégák minden olyan esetben a közvetítőhöz fordulnak, amikor egy másik kollégával szeretnének kommunikálni.

Együttműködés

- A kollégák kérelmeket küldenek a Közvetítő objektumoknak és kérelmeket fogadnak onnan. Az együttműködést a közvetítő alakítja ki azzal, hogy továbbítja a kérelmeket a megfelelő kollégá(k)nak.

Következmények

A Közvetítő tervezési minta előnyei és hátrányai a következők:

1. *Csökkenti az alosztályok számát.* A közvetítő egy helyre gyűjt egy olyan viselkedést, amit másképp több objektum között kellene elosztani. A viselkedés megváltoztatásához így csak a Közvetítő osztályból kell alosztályokat származtatnunk, a Kolléga osztályok eredeti formájukban újrahasznosíthatók.
2. *Elválasztja a kollégákat.* A közvetítő laza csatolást hoz létre a kollégák között. A Kolléga és Közvetítő osztályok egymástól függetlenül variálhatók és újrahasznosíthatók.
3. *Egyszerűsíti az objektumprotokollokat.* A közvetítő a sok–sok kapcsolatokat a közvetítő és kollégái egy–sok kapcsolataival helyettesíti. Az egy–sok kapcsolatok könnyebben átláthatók, illetve könnyebb fenntartani és bővíteni azokat.
4. *Az objektumok együttműködését elvonttá teszi.* A közvetítés mint külön fogalom önálló objektumba zárása lehetővé teszi, hogy figyelmünket az objektumok együttműködésének módjára összpontosíthassuk, és ne kelljen foglalkoznunk egyedi viselkedésükkel. Így a rendszer objektumainak együttműködése tisztábban átlátható.
5. *Központosítja a vezérlést.* A Közvetítő minta az együttműködés egyszerűsítéséért a közvetítő összetettségével fizet. Miután a közvetítő tartalmazza a protokollokat, bonyolultabb lehet, mint bármelyik kolléga, és nehezen kezelhetővé, tömbszerűvé válhat.

Megvalósítás

A Közvetítő minta megvalósítása során a következőkre kell figyelni:

1. *Az elvont Közvetítő osztály kihagyása.* Ha a kollégák csak egyetlen közvetítővel állnak kapcsolatban, nincs szükség elvont Közvetítő osztály létrehozására. A Közvetítő által biztosított elvont csatolás lehetővé teszi a kollégáknak, hogy különböző Közvetítő alosztályokkal működjenek együtt, és ez megfordítva is igaz.
2. *Kapcsolattartás a Kolléga és Közvetítő osztályok között.* A kollégáknak kapcsolatba kell lépniük közvetítőjükkel, ha valamilyen számukra fontos esemény történik. A közvetítőt megvalósíthatjuk például megfigyelőként, a Megfigyelő tervezési minta alkalmazásával. A kollégák ebben az esetben Alanyok (Subject), amelyek állapotuk megváltozásakor értesítést küldenek a közvetítőnek, az pedig a változás hatásának a kollégák közötti elterjesztésével válaszol.
   Egy másik megközelítés lehet, ha a Közvetítő osztályban külön értesítő felületet határozunk meg, melynek segítségével a kollégák közvetlenebbül kommunikálhatnak. A Smalltalk/V for Windows a képviselet egy formáját alkalmazza: a kollégák a közvetítővel való társalgás folyamán saját magukat paraméterként adják át, így a közvetítő azonosíthatja a küldőt. Példakódunk is erre a megközelítésre épül, a Smalltalk/V megvalósítást pedig az Ismert felhasználások részben részletesebben is bemutatjuk.

Példakód

A Feladat részben bemutatott betűtípus-választó párbeszédablakot egy PárbeszédablakIrányító segítségével valósítjuk meg. A `DialogDirector` (PárbeszédablakIrányító) elvont osztály az irányítók felületét határozza meg.

```
class DialogDirector {
public:
   virtual ~DialogDirector();

   virtual void ShowDialog();
   virtual void WidgetChanged(Widget*) = 0;

protected:
   DialogDirector();
   virtual void CreateWidgets() = 0;
};
```

A `Widget` (Vezérlő) a grafikus vezérlők elvont alaposztálya, amely ismeri az irányítóját.

```
class Widget {
public:
   Widget(DialogDirector*);
   virtual void Changed();
```

```
    virtual void HandleMouse(MouseEvent& event);
    //...
private:
    DialogDirector* _director;
};
```

A Changed (Megváltozott) az irányító WidgetChanged (VezérlőMegváltozott) műveletét hívja meg. A vezérlők e művelet meghívásával tájékoztatják az irányítót, hogy valamilyen esemény történt.

```
void Widget::Changed () {
    _director->WidgetChanged(this);
}
```

A DialogDirector alosztályai a WidgetChanged felülbírálásával irányítják a megfelelő vezérlőket. A vezérlő átad egy önmagára mutató hivatkozást argumentumként a WidgetChanged műveletnek, így az irányító azonosíthatja a megváltozott vezérlőt. A DialogDirector alosztályok a CreateWidgets (LétrehozVezérlők) műveletet tisztán virtuálisként felülírják, így hozzák létre a vezérlőket a párbeszédablakban.

A ListBox, az EntryField és a Button (ListaMező, BeviteliMező, Gomb) a Widget alosztályai a felhasználói felület elemei számára. A ListBox a listában kijelölt elem kiolvasásához a GetSelection (SzerezKijelölés) műveletet biztosítja, míg az EntryField SetText (BeállítSzöveg) művelete új szöveget helyez a mezőbe.

```
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);

    virtual const char* GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent& event);
    //...
};

class EntryField: public Widget {
public:
    EntryField(DialogDirector*);

    virtual void SetText(const char* text);
    virtual const char* GetText();
    virtual void HandleMouse(MouseEvent& event);
    //...
};
```

A Button egy egyszerű vezérlő, amely a gomb megnyomásakor meghívja a Changed (Megváltozott) műveletet. Ez a HandleMouse (KezelEgér) megvalósításában történik:

```
class Button: public Widget {
   public:
   Button(DialogDirector*);

   virtual void SetText(const char* text);
   virtual void HandleMouse(MouseEvent& event);
   //...
};
   void Button::HandleMouse(MouseEvent& event) {
         //...
         Changed();
    }
```

A párbeszédablak vezérlői között a FontDialogDirector (BetűtípusPárbeszédablak-Irányító) közvetít, amely a DialogDirector alosztálya:

```
class FontDialogDirector : public DialogDirector {
public:
   FontDialogDirector();
   virtual ~FontDialogDirector();
   virtual void WidgetChanged(Widget*);

protected:
   virtual void CreateWidgets();

private:
   Button* _ok;
   Button* _cancel;
   ListBox* _fontList;
   EntryField* _fontName;
};
```

A FontDialogDirector számon tartja az általa megjelenített vezérlőket. A CreateWidgets felülírásával létrehozza azokat és előkészíti a rájuk mutató hivatkozásokat:

```
void FontDialogDirector::CreateWidgets () {
   _ok = new Button(this);
   _cancel = new Button(this);
   _fontList = new ListBox(this);
   _fontName = new EntryField(this);

   // a listamező feltöltése az elérhető betűtípus-nevekkel

   // a párbeszédablak vezérlőinek elkészítése
}
```

A vezérlők helyes együttműködéséről a `WidgetChanged` (VezérlőMegváltozott) gondoskodik:

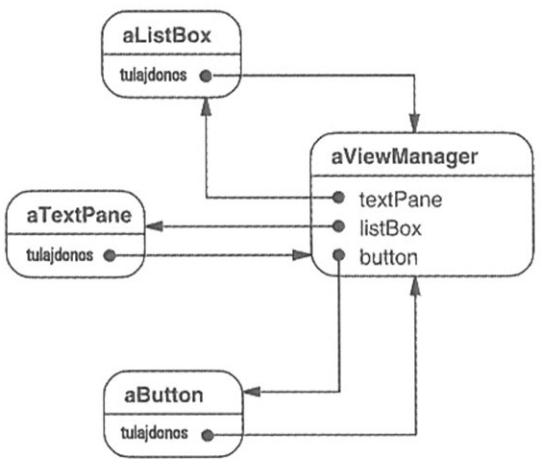
```
void FontDialogDirector::WidgetChanged (
    Widget *theChangedWidget
) {
    if (theChangedWidget == _fontList) {
            fontName->SetText(_fontList->GetSelection());

    } else if (theChangedWidget == _ok) {
            // a betűtípus megváltoztatása és az ablak bezárása
            //...
    } else if (theChangedWidget == _cancel) {
            // a párbeszédablak bezárása
    }
}
```

A `WidgetChanged` bonyolultsága a párbeszédablak összetettségével egyenesen arányosan nő. A nagy párbeszédablakok természetesen egyéb okokból kifolyólag sem kívánatosak, de a közvetítő bonyolultsága háttérbe szoríthatja a tervezési minta előnyeit.

Ismert felhasználások

Vezérlők közötti közvetítőként mind az ET++ [WGM88], mind a THINK C osztálykönyvtára [Sym93b] Irányító-szerű objektumokat használ a párbeszédablakokban.

A Smalltalk/V for Windows alkalmazások szerkezete is a Közvetítő mintán alapul [LaL94]. Ebben a környezetben az alkalmazások egy Window (Ablak) objektumból állnak, amely „ablaktáblákat" (Pane) tartalmaz. A könyvtár számos előre elkészített Pane objektumot (TextPane, ListBox, Button stb.) bocsát rendelkezésre; ezek alosztályok létrehozása nélkül felhasználhatók. A programfejlesztőnek csak a ViewManager (NézetKezelő) osztályból kell alosztályokat származtatnia; ez az osztály felel a táblák összehangolásáért. Tehát a ViewManager a közvetítő, a táblák pedig csak saját nézetkezelőjüket ismerik, amely az adott tábla „tulajdonosának" számít. A táblák közvetlenül nem hivatkoznak egymásra. Az alábbi objektumdiagram egy alkalmazás futásidejű pillanatfelvételét mutatja:



A Pane és ViewManager objektumok közötti kapcsolattartáshoz a Smalltalk/V eseményeket használ. A táblák eseményt váltanak ki, ha információt akarnak nyerni a közvetítőtől, vagy közölni kívánják vele, hogy valami fontos dolog történt. Az események egy szimbólummal (pl. `#select`) azonosítják az eseményt. Az esemény kezeléséhez a nézetkezelő bejegyeztet egy metódusválasztót a táblával; ez a választó lesz az esemény kezelője, és meghívására minden esetben sor kerül, ha valamilyen esemény történik.

Az alábbi kódrészlet azt mutatja be, hogyan jön létre egy ListPane objektum egy ViewManager alosztályon belül, és a nézetkezelő hogyan jegyezteti be a `#select` esemény eseménykezelőjét:

```
self addSubPane: (ListPane new
   paneName: 'myListPane';
   owner: self;
   when: #select perform: #listSelect:).
```

A Közvetítő minta egy másik alkalmazása az összetett frissítések lebonyolítása. Erre a Megfigyelő mintánál bemutatott VáltozásKezelő (ChangeManager) osztály ad példát. A VáltozásKezelő alanyok és megfigyelők között közvetít, hogy elkerülhessük a felesleges többszöri frissítést. Amikor egy objektum megváltozik, értesíti a VáltozásKezelőt, amely az objektumtól függő más objektumok értesítésével összehangolja a frissítést.

Ehhez hasonló alkalmazást láthatunk a Unidraw grafikai keretrendszerben [VL90], ahol a CSolver nevű osztály az úgynevezett „konnektorok" közötti kapcsolati kötéseket kezeli. A grafikus szerkesztők objektumai különböző módokon összekapcsolva jelenhetnek meg. A konnektorok (összekötők) azokban a programokban lehetnek hasznosak, amelyekben a kapcsolatok fenntartása automatikus (ilyenek például a diagramszerkesztők vagy az áramkörtervező rendszerek). A CSolver a konnektorok közötti közvetítő, amelynek feladata a kapcsolati kötések feloldása és a konnektorok helyének frissítése a kötések változásának megfelelően.

Kapcsolódó minták

Homlokzat: A Homlokzat tervezési minta annyiban különbözik a Közvetítőtől, hogy objektumok elvont alrendszere révén kényelmesebb felületet biztosít. Protokollja egyirányú, vagyis a Homlokzat objektumok intézhetnek kérelmeket az alrendszer osztályaihoz, de ez visszafelé nem működik. A Közvetítő minta ezzel szemben olyan együttműködési lehetőségeket biztosít, amiket a kolléga objektumok nem képesek biztosítani, a protokoll pedig többirányú.

Megfigyelő: A kollégák a közvetítővel a Megfigyelő minta segítségével tarthatnak fenn kapcsolatot.

Emlékeztető

Viselkedési objektumminta

Cél

Az egységbe zárás (betokozás, enkapszuláció) megsértése nélkül rögzíteni és felfedni egy objektum belső állapotát, hogy az később ebbe az állapotba visszaállítható legyen.

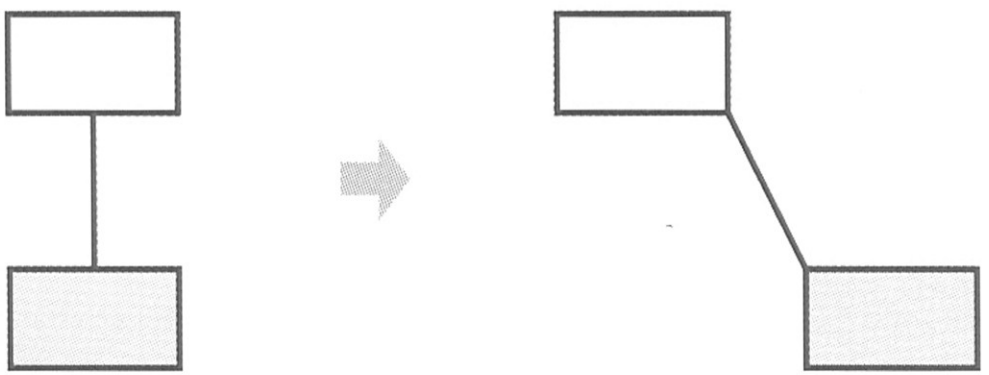
Egyéb nevek

Memento, Pillanatfelvétel, Token

Feladat

Időnként szükséges lehet egy objektum belső állapotának rögzítése. Ez különösen akkor fontos, amikor ellenőrző pontokat vagy művelet-visszavonási lehetőségeket valósítunk meg, melyek segítségével a felhasználó „kihátrálhat" egyes műveletekből, vagy a program hiba utáni helyreállítást végezhet. Ahhoz, hogy objektumokat egy korábbi állapotba állíthassunk vissza, valahová állapotinformációkat kell mentenünk. Az objektumok azonban általában elrejtik állapotukat; más objektumok nem férhetnek hozzá, tehát a külső mentés sem lehetséges. Ezen állapot felfedése az egységbe zárás megsértése lenne, aminek a program megbízhatósága és bővíthetősége látná kárát.

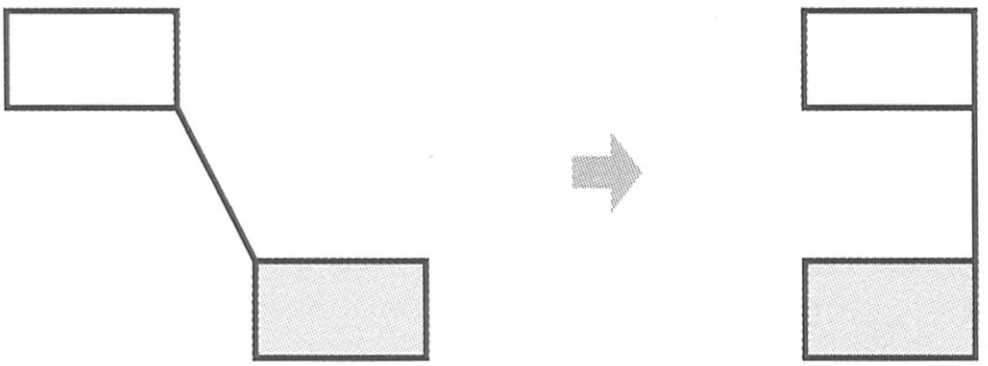
Vegyünk példaképpen egy grafikus szerkesztőt, amely támogatja az objektumok összekapcsolását. A felhasználó összeköthet két téglalapot egy vonallal, és a téglalapok összekötve maradnak akkor is, ha a felhasználó valamelyiküket elmozdítja. A szerkesztő gondoskodik róla, hogy a kapcsolat fenntartásához a vonal megnyúljon.



Az objektumok közötti kapcsolódások fenntartásának egy jól ismert módja a megszorítás-feloldó rendszerek használata. Ezt a szolgáltatást egy MegszorításFeloldó (ConstraintSolver) objektumba zárhatjuk, amely rögzíti a létrehozott kapcsolatokat, és azokat leíró matematikai egyenleteket állít elő. Amikor a felhasználó létrehoz egy kapcsolatot vagy más módon

megváltoztatja a diagramot, az objektum ezeket az egyenleteket oldja meg (solve), majd számításainak eredményét felhasználva újrarendezi a grafikus elemeket, hogy a kapcsolatok megfelelően megmaradjanak.

Egy ilyen típusú programban a művelet-visszavonás támogatása egyáltalán nem olyan könnyű, mint amilyennek tűnik. Egy mozgató művelet visszavonására kézenfekvőnek látszhat a megtett távolság tárolása, és ugyanakkora távolság megtétele visszafelé, ez azonban nem garantálja, hogy minden objektum ugyanott fog megjelenni, ahol eredetileg volt. Ha a kapcsolatról nem gondoskodunk megfelelően, a téglalap egyszerű visszafelé mozgatása az eredeti helyére nem feltétlenül jár a kívánt eredménnyel.



A MegszorításFeloldó nyilvános felülete általában nem elégséges a pontos visszavonáshoz. A visszavonó műveleteknek szorosabban kell együttműködniük a MegszorításFeloldóval a korábbi állapot visszaállításához, de az objektum belső szerkezetét nem szabad felfednünk előttük.

A problémát az Emlékeztető tervezési mintával oldhatjuk meg. Az **emlékeztető** olyan objektum, amely egy másik objektum, a **kezdeményező** (originator) belső állapotáról készít pillanatfelvételt. A visszavonó művelet a kezdeményezőtől emlékeztetőt kér, amikor ellenőriznie kell annak állapotát. Az emlékeztető kezdeti értéke az aktuális állapotot jellemző információ lesz. Az emlékeztetőben csak a kezdeményező tárolhat, illetve csak ő nyerhet ki onnan információt, más objektumok számára az emlékeztető „átlátszatlan".

A grafikus szerkesztő imént tárgyalt példájában a MegszorításFeloldó a kezdeményező. A visszavonás folyamata az események következő láncolatából áll:

1. A szerkesztő mozgató művelet „mellékhatásaként" emlékeztetőt kér a MegszorításFeloldótól.
2. A MegszorításFeloldó létrehozza és átadja az emlékeztetőt, ami ebben az esetben a FeloldóÁllapot (SolverState) osztály példánya. A FeloldóÁllapot emlékeztető olyan adatszerkezeteket tartalmaz, amelyek leírják a MegszorításFeloldó belső egyenleteinek és változóinak jelenlegi állapotát.
3. Később, amikor a felhasználó visszavonja a mozgató műveletet, a szerkesztő visszaadja a FeloldóÁllapotot a MegszorításFeloldó objektumnak.

4. A FeloldóÁllapotban tárolt adatok alapján a MegszorításFeloldó megváltoztatja belső szerkezeteit, hogy az egyenleteket és változókat pontosan visszaállíthassa korábbi állapotukba.

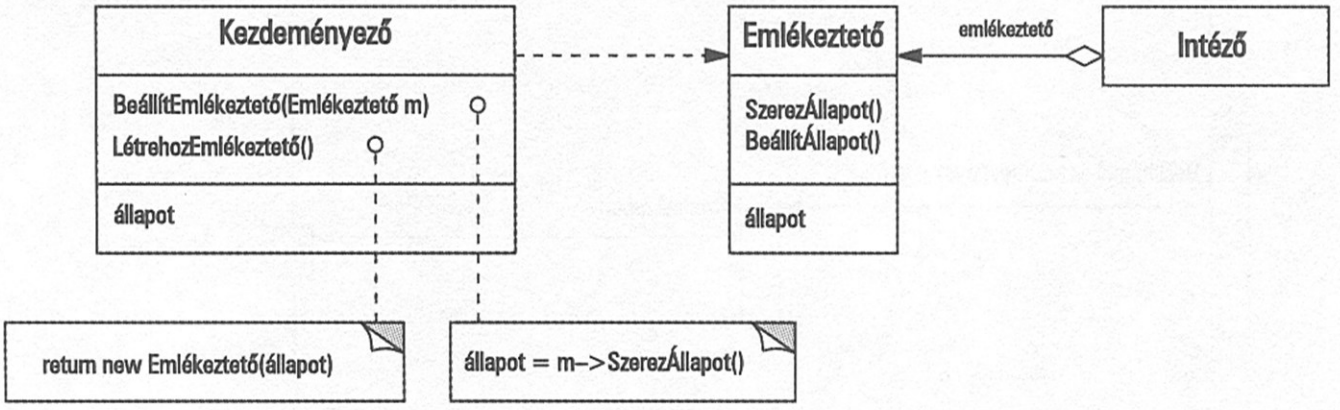
A fenti lépések lehetővé teszik a MegszorításFeloldó számára, hogy más objektumokra bízza a korábbi állapot visszaállításához szükséges információkat, anélkül, hogy belső szerkezetét felfedné előttük.

Alkalmazhatóság

Az Emlékeztető mintát az alábbi esetekben célszerű alkalmazni:

- Egy objektum (vagy objektumrész) állapotáról pillanatfelvételt kell készíteni, hogy később ebbe az állapotba visszaállítható legyen, *és*
- az állapotot közvetlenül lekérdező felület felfedné a megvalósítás részleteit, és megsértené az objektum egységbe zárását.

Szerkezet



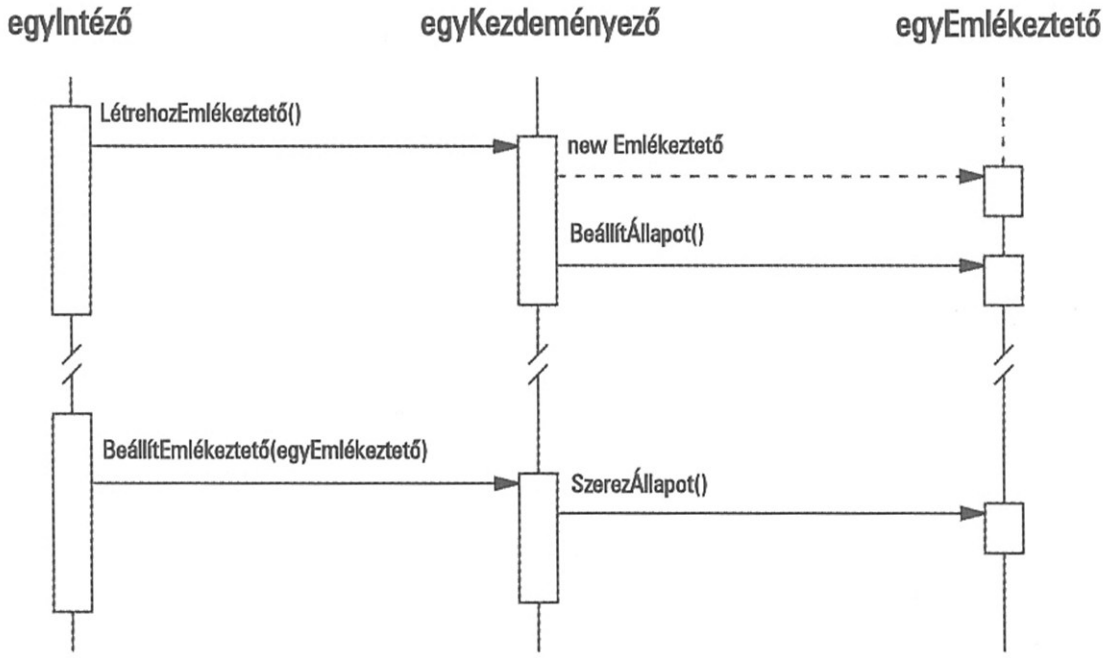
Résztvevők

- Emlékeztető (FeloldóÁllapot)
  - A Kezdeményező (Originator) objektum belső állapotát tárolja. Csak annyi információt raktároz el, amennyit szükséges.
  - Megakadályozza, hogy a kezdeményezőn kívül más objektumok is hozzáférhessenek az adatokhoz. Az emlékeztetőknek két felületük van. Az Intéző (Caretaker) csak egy *keskeny* felületet lát, ezért mást nem tehet, csak továbbadhatja az emlékeztetőt más objektumoknak, a Kezdeményező ezzel szemben széles felületet, így minden adathoz hozzáférhet, amire szüksége van korábbi állapotának visszaállításához. Ideális esetben csak az emlékeztetőt készítő kezdeményező számára engedélyezett, hogy az emlékeztető belső állapotához hozzáférjen.

- **Kezdeményező** (MegszorításFeloldó)
  - Emlékeztetőt készít, ami az aktuális belső állapotának pillanatfelvételét tartalmazza.
  - Az emlékeztető segítségével visszaállítja korábbi belső állapotát.
- **Intéző** (visszavonó rendszer)
  - Az emlékeztető biztonságáért felel.
  - Az emlékeztető tartalmát soha nem vizsgálja, azon műveleteket nem végez.

Együttműködés

- Az intéző emlékeztetőt kér a kezdeményezőtől, tárolja egy ideig, majd visszaadja annak, amint azt a következő együttműködési diagram is ábrázolja:



  Az intéző abban az esetben nem adja vissza az emlékeztetőt a kezdeményezőnek, ha annak nincs szüksége rá a korábbi állapot visszaállításához.
- Az emlékeztetők passzívak, állapotukat csak az őket létrehozó kezdeményező kérdezheti le vagy változtathatja meg.

Következmények

Az Emlékeztető minta előnyei és hátrányai a következők:

1. *Megőrzi az egységbe zárás határait.* Az Emlékeztető minta segítségével elkerülhetjük azon információk felfedését, amelyeket csak a kezdeményezőnek szabad ismernie, de a tárolás ettől függetlenül az objektumon kívül történik. A minta elszigeteli a kezdeményező esetleg bonyolult belső szerkezetét a többi objektumtól, így megőrzi az egységbe zárás határait.

2. *Egyszerűbbé teszi a kezdeményező objektumot.* Az egységbe zárást megőrző más szerkezetekben a kezdeményező számon tartja az ügyfelek által kért belső állapotok változatait, ami a tárolás kezelésének minden terhét a kezdeményező vállára helyezi. Az, hogy az ügyfelek maguk kezelik a kért állapotot, egyszerűbbé teszi a kezdeményező felépítését, és azt is szükségtelenné teszi, hogy az ügyfeleknek értesíteniük kelljen a kezdeményezőt, ha munkájukkal végeztek.
3. *Az emlékeztetők használata költséges lehet.* Az emlékeztetők jelentős többletterhet róhatnak a programra, ha a kezdeményezőnek nagy mennyiségű adatot kell az emlékeztetőbe másolnia, vagy ha az ügyfelek gyakran kérnek, illetve adnak vissza emlékeztetőket. A tervezési minta alkalmazása csak akkor éri meg, ha a Kezdeményező egységbe zárása és visszaállítása „olcsó". (Lásd még a fokozatosságról írottakat a Megvalósítás részben.)
4. *A keskeny és széles felületek meghatározása nem mindig könnyű.* Egyes nyelvekben nehéz biztosítani, hogy az emlékeztető állapotához csak a kezdeményező férhessen hozzá.
5. *Az emlékeztetők kezelésének rejtett költségei vannak.* Az Intéző felel az emlékeztetők törléséért, csakhogy arról már nincs fogalma, mennyi adat tárolódik az emlékeztetőben, így az egyébként egyszerű intéző az emlékeztetők tárolásához jelentős tárhelyet emészthet fel.

Megvalósítás

Az Emlékeztető minta megvalósítása során többek között két dologra kell figyelnünk:

1. *Nyelvi támogatás.* Az emlékeztetők két felülettel rendelkeznek: egy szélessel a kezdeményező, és egy keskennyel a többi objektum számára. Ideális esetben a megvalósításra használt nyelv a statikus védelem két szintjét támogatja. A C++ például azzal, hogy a kezdeményezőt az emlékeztető barátjává, az emlékeztető széles felületét pedig priváttá tehetjük. Csak a keskeny felületet kell nyilvánosként meghatároznunk. Lássunk egy példát:

```
class State;

class Originator {
public:
        Memento* CreateMemento();
        void SetMemento(const Memento*);
        //...
private:
        State* _state;        // belső adatszerkezetek
        //...
};

class Memento {
public:
        // keskeny nyilvános felület
```

```
        virtual ~Memento();
    private:
        // csak a Kezdeményező számára elérhető privát tagok
        friend class Originator;
        Memento();

        void SetState(State*);
        State* GetState();
        //...
    private:
        State* _state;
        //...
    };
```

2. *A változások fokozatos tárolása.* Ha az emlékeztetők létrehozása és visszaadása a kezdeményezőnek előre látható módon zajlik, megtehetjük, hogy az emlékeztetőben csak az eredeti állapothoz képest beállt *változásokat* rögzítjük.
   Egy előzménylistában található visszavonható parancsok például emlékeztetők segítségével biztosíthatják, hogy visszavonáskor pontosan helyreálljon a korábbi állapot (lásd a Parancs tervezési mintát). Az előzménylista meghatározza, milyen sorrendben lehet a parancsokat visszavonni és újból végrehajtani, így az emlékeztetőknek elég csak a parancsok által előidézett változásokat tárolni, az érintett objektumok teljes állapotát nem szükséges. A Feladat részben feljebb bemutatott példában a MegszorításFeloldó csak azokat a belső szerkezeteket tárolja, amelyek megváltoznak, hogy a téglalapokat összekötő vonalakat megtartsák, nem pedig az objektumok abszolút helyzetét.

Példakód

Az itt szereplő C++ kód a korábban tárgyalt MegszorításFeloldó (ConstraintSolver) példához tartozik. A grafikus objektumok egyik helyről a másikra mozgatására, illetve ezen művelet visszavonására `MoveCommand` (MozgatParancs) objektumokat használunk (lásd a Parancs tervezési mintát). A grafikus elemek mozgatásához a szerkesztő a parancs `Execute` (Végrehajt) műveletét hívja meg, a visszavonáshoz pedig az `Unexecute`-ot (Visszavon). A parancs tárolja a célobjektumot, a megtett távolságot, és a `ConstraintSolverMemento` (MegszorításFeloldóEmlékeztető) nevű emlékeztető egy példányát, amely a megszorításfeloldó állapotát rögzíti.

```
class Graphic;
   // a szerkesztő grafikus objektumainak alaposztálya

class MoveCommand {
public:
   MoveCommand(Graphic* target, const Point& delta);
   void Execute();
   void Unexecute();
private:
```

```
    ConstraintSolverMemento* _state;
    Point _delta;
    Graphic* _target;
};
```

A kapcsolatokat a `ConstraintSolver` osztály hozza létre. Kulcsfüggvénye a `Solve` (Felold), amely az `AddConstraint` (HozzáadKötés) művelettel bejegyzett kapcsolatokat oldja fel. A visszavonás támogatásához a `ConstraintSolver` állapota a `CreateMemento` (LétrehozEmlékeztető) művelettel tehető „külsővé", amely az állapotot egy `ConstraintSolverMemento` példányba helyezi. A megszorításfeloldó a `SetMemento` (BeállítEmlékeztető) hívásával állítható vissza korábbi állapotába. A `ConstraintSolver` Egyke.

```
class ConstraintSolver {
public:
    static ConstraintSolver* Instance();

    void Solve();
    void AddConstraint(
            Graphic* startConnection, Graphic* endConnection
    );
    void RemoveConstraint(
            Graphic* startConnection, Graphic* endConnection
    );
    ConstraintSolverMemento* CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // nem triviális állapot és műveletek
    // a kapcsolat létrehozásához
};

class ConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();

    // privát megszorításfeloldó-állapot
};
```

A fenti felületekkel a következő módon valósíthatjuk meg a `MoveCommand` `Execute` és `Unexecute` tagjait:

```
void MoveCommand::Execute () {
    ConstraintSolver* solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento(); // emlékeztető létrehozása
    _target->Move(_delta);
    solver->Solve();
}
```

```
void MoveCommand::Unexecute () {
   ConstraintSolver* solver = ConstraintSolver::Instance();
   _target->Move(-_delta);
   solver->SetMemento(_state);       // az állapot visszaállítása
   solver->Solve();
}
```

Az `Execute` a grafika mozgatása előtt kér egy `ConstraintSolverMemento` emlékeztetőt; az `Unexecute` visszahelyezi a grafikát, visszaállítja a megszorításfeloldó korábbi állapotát, végül pedig arra utasítja azt, hogy számítsa ki a kapcsolatot.

Ismert felhasználások

Az előző példakód a Unidraw kapcsolatokat támogató CSolver osztályán [VL90] alapult.

A Dylan [App92] gyűjteményei egy olyan bejáró felületet biztosítanak, ami szintén az Emlékeztető mintára épül. E gyűjtemények ismerik az „állapotobjektum" fogalmát, ami egy olyan emlékeztető, amely a bejárás állapotát jelöli. A gyűjtemények a bejárás állapotának ábrázolására bármilyen módszert választhatnak, az ügyfelek elől az tökéletesen rejtve marad. A Dylan bejárási megközelítése C++ nyelven valahogy így festene:

```
template<class Item>
class Collection {
public:
   Collection();

   IterationState* CreateInitialState();
   void Next(IterationState*);
   bool IsDone(const IterationState*) const;
   Item CurrentItem(const IterationState*) const;
   IterationState* Copy(const IterationState*) const;

   void Append(const Item&);
   void Remove(const Item&);
   //...
};
```

A `CreateInitialState` (LétrehozKezdetiÁllapot) egy kezdőértékkel ellátott `IterationState` (BejárásÁllapot) objektumot ad vissza a gyűjteménynek. A `Next` (Következő) az állapotobjektumot a bejárás következő pozíciójára állítja, ezzel növelve a bejárási indexet. Az `IsDone` (Kész) értéke `true` lesz, ha a `Next` túllépett a gyűjtemény utolsó elemén. A `CurrentItem` (Aktuális elem) követi az állapotobjektum hivatkozását, és a gyűjtemény hivatkozott elemét adja vissza, a `Copy` (Másol) pedig az adott állapotobjektum másolatát. Ezzel a bejárás egy adott pontját jelölhetjük meg.

Ha adott egy `ItemType` (ElemTípus) nevű osztály, példányainak gyűjteményét a következő módon járhatjuk be:

```
class ItemType {
public:
   void Process();
   //...
};

Collection<ItemType*> aCollection;
IterationState* state;

state = aCollection.CreateInitialState();

while (!aCollection.IsDone(state)) {
   aCollection.CurrentItem(state)->Process();
   aCollection.Next(state);
}
delete state;[7]
```

Az emlékeztető alapú bejárási felületnek két érdekes előnye van:

1. Egy gyűjteményhez egynél több állapot is tartozhat. (Ugyanez igaz a Bejáró mintára is.)
2. A bejárás támogatásához nincs szükség a gyűjtemény egységbe zárásának megsértésére. Az emlékeztetőt csak maga a gyűjtemény értelmezi, más nem férhet hozzá. A bejárás más megközelítéseinél előfordulhat, hogy az egységbe zárást fel kell törnünk, például ha a bejáró osztályokat gyűjteményosztályaik barátjává kell tennünk (lásd a Bejáró mintát). Az emlékeztető alapú megvalósításban ennek éppen a fordítottja a helyzet: a `Collection` (Gyűjtemény) az `IteratorState` barátja.

A QOCA megszorításfeloldó elemkészlet az emlékeztetőkben csak a különbségeket tárolja [HHMV92]. Itt az ügyfelek egy emlékeztetőt kapnak, amely a kötések egy bizonyos rendszerére adott aktuális megoldás jellemzőit tartalmazza. Az emlékeztető csak azokat a változókat tárolja, amelyek az utolsó megoldás óta megváltoztak, ami általában csak a változók egy kis részhalmaza. E részhalmaz elegendő ahhoz, hogy a feloldó visszatérhessen a megelőző megoldáshoz. A QOCA az előzményekre támaszkodik; a korábbi megoldások a közbenső megoldások helyreállított emlékeztetőiből állíthatók vissza, így az emlékeztetők sorrendje nem módosítható.

Kapcsolódó minták

Parancs: A parancsok emlékeztetőket alkalmazhatnak a visszavonható műveletek állapotának rögzítésére.

Bejáró: Az emlékeztetők a korábban leírt módon bejárás támogatására is használhatók.

---

[7] Megfigyelhetjük, hogy a példában az állapotobjektumot a bejárás végén töröljük. A `delete` (töröl) azonban nem hívódik meg, ha a `ProcessItem` (FeldolgozElem) kivételt vált ki, így szemét keletkezik. Ez a C++-szal szemben a szemétgyűjtéssel rendelkező Dylan-ben nem okoz gondot. A problémára a Bejáró mintánál láthattunk egy megoldást.

Megfigyelő

Viselkedési objektumminta

Cél

Objektumok között egy–sok függőségi kapcsolatot létrehozni, így amikor az egyik objektum állapota megváltozik, minden tőle függő objektum értesül erről és automatikusan frissül.

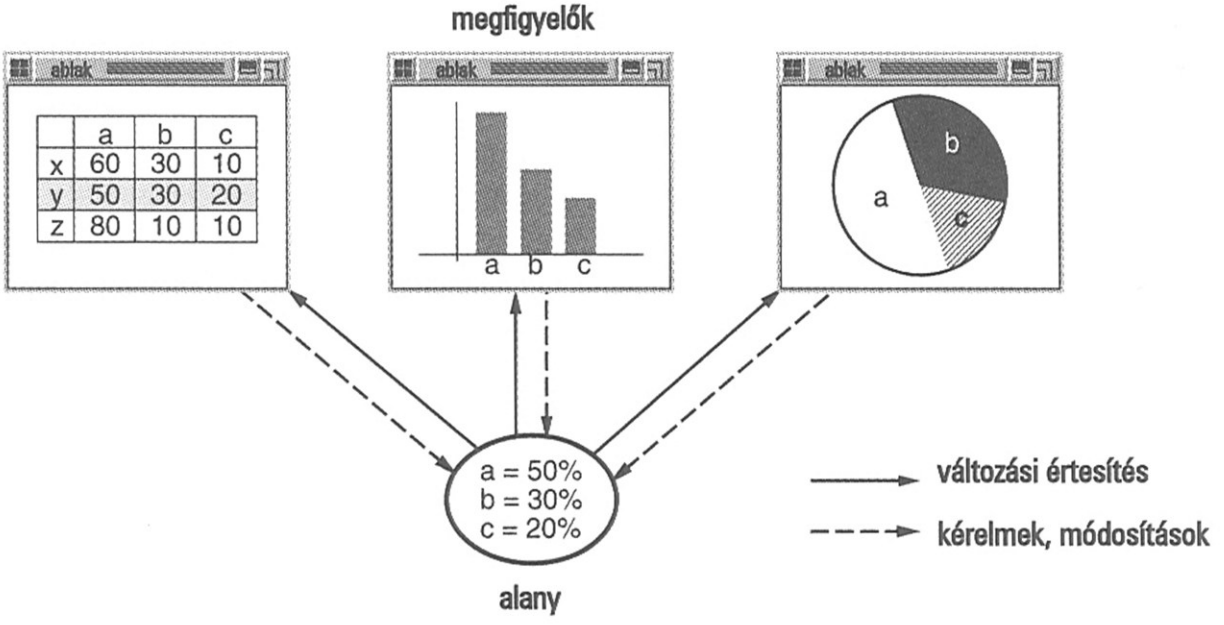
Egyéb nevek

Observer, Figyelő, Dependents (Függőségek), Publish-Subscribe (Közzététel–Előfizetés)

Feladat

Gyakori probléma, hogy amikor egy rendszert együttműködő osztályokra bontunk, gondoskodnunk kell a kapcsolatban álló objektumok következetességének fenntartásáról is. Ezt többnyire nem szoros csatolással szeretnénk elérni, hiszen ez csökkentené az osztályok újrahasznosíthatóságát.

Számos, grafikus felhasználói felületekhez készített elemkészlet például különválasztja a megjelenítést a mögötte megbúvó alkalmazásadatoktól [KP88, LVC89, P+88, WGM88], így az e kettőt meghatározó osztályok egymástól függetlenül újrahasznosíthatók, bár természetesen együtt is működhetnek. Különböző megjelenítési módszereket használva ugyanabban az alkalmazási adatobjektumban egy táblázat objektum és egy oszlopdiagram objektum is megjeleníthet információt. A táblázat és az oszlopdiagram nem tudnak egymásról (bár úgy *viselkednek*, mintha tudnának), így elég csak a számunkra szükséges objektum újrahasznosítása. Amikor a felhasználó módosít egy adatot a táblázatban, az oszlopdiagram azonnal tükrözi a változtatásokat, és ugyanez igaz megfordítva is.



Ez a viselkedés maga után vonja, hogy a táblázat és az oszlopdiagram az adatobjektumtól függ, így az annak állapotában bekövetkező bármilyen változás ezek értesítését igényli. A függő objektumok számát persze nem szükséges kettőre korlátozni; ugyanazt az adatot korlátlan számú felhasználói felületi elemmel megjeleníthetjük.

A Megfigyelő minta azt írja le, hogyan alakíthatjuk ki ezeket a kapcsolatokat. A minta kulcsobjektumai az **alany** (subject) és a **megfigyelő** (observer). Az alanynak bármennyi, tőle függő megfigyelője lehet, amelyek az alany állapotában beálló minden változásról értesülnek. Ezután a megfigyelők kérelmet intéznek az alanyhoz, hogy összehangolhassák állapotukat az alanyéval.

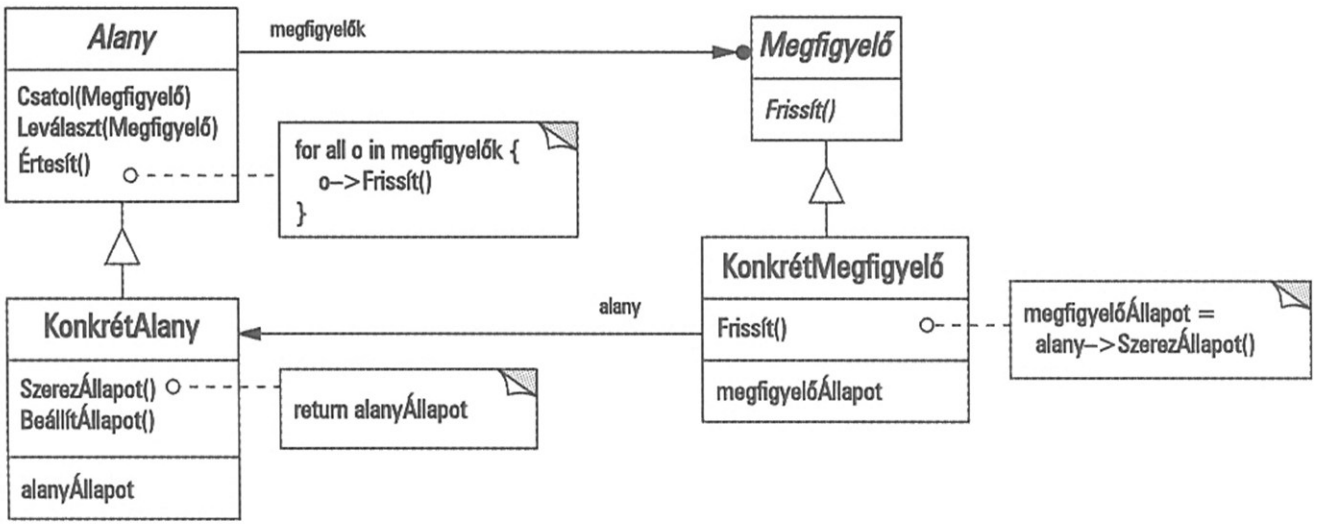
Ezt a fajta együttműködést közzététel–előfizetés néven is ismerik. Az alany az értesítések „közzétevője", amely az értesítéseket anélkül küldheti el, hogy tudná, kik a megfigyelői, a megfigyelők – amelyekből bármennyi lehet – pedig „előfizethetnek" ezen értesítésekre.

Alkalmazhatóság

A Megfigyelő tervezési mintát a következő helyzetekben célszerű alkalmazni:

- Egy fogalomhoz két ábrázolás kapcsolódik, és egyik a másiktól függ. Ezen ábrázolások külön objektumba zárása lehetővé teszi, hogy egymástól függetlenül módosíthassuk vagy újrahasznosíthassuk őket.
- Egy adott objektum módosítása más objektumok módosítását igényli, és nem tudjuk, hány objektumot kell megváltoztatnunk.
- Egy adott objektumnak képesnek kell lennie arra, hogy anélkül értesítsen más objektumokat, hogy feltételezésekkel élne azok kilétéről. Más szavakkal, az említett objektumok között nem szeretnénk szoros csatolást létrehozni.

Szerkezet



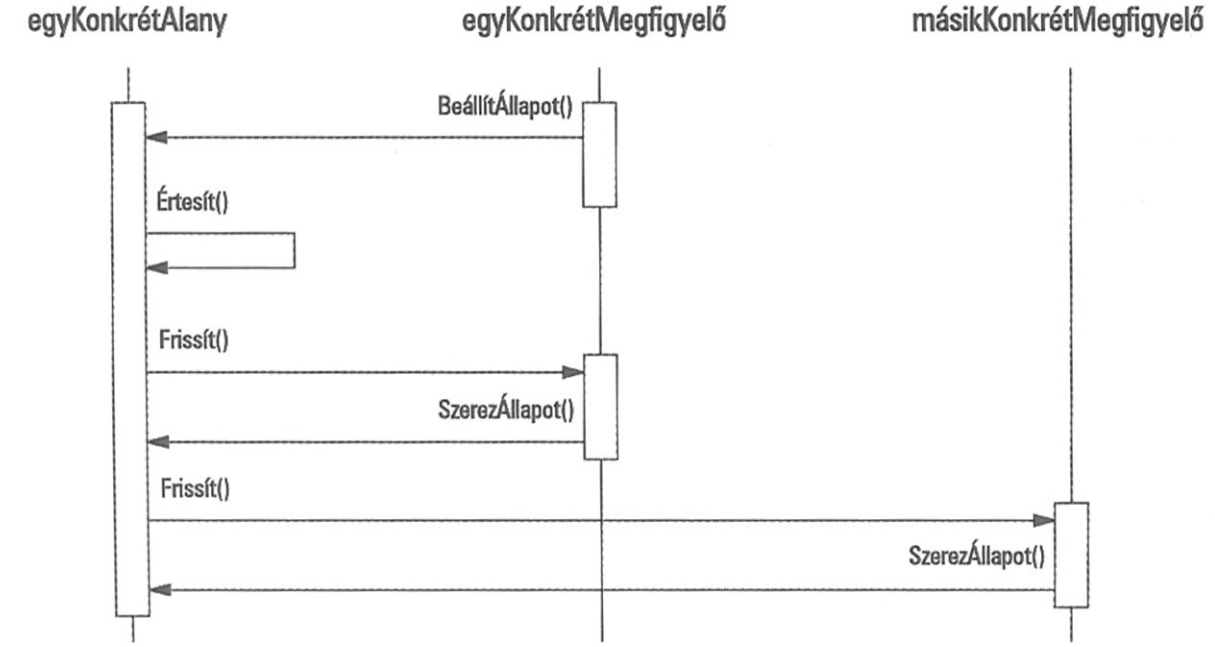
Résztvevők

- Alany
  - Ismeri a megfigyelőit. Az Alanyt (Subject) korlátlan számú Megfigyelő (Observer) objektum figyelheti meg.
  - Felületet biztosít a Megfigyelő objektumok csatolására és leválasztására.
- Megfigyelő
  - Frissítő felületet határoz meg az alanyban bekövetkező változásokról értesítendő objektumok számára.
- KonkrétAlany
  - A KonkrétMegfigyelő (ConcreteObserver) objektumok számára érdekes állapotot tárolja.
  - Értesítést küld megfigyelőinek, ha állapota megváltozik.
- KonkrétMegfigyelő
  - Hivatkozást tart fenn egy KonkrétAlany (ConcreteSubject) objektumra.
  - Tárolja azt az állapotot, amelynek összhangban kell maradnia az alany állapotával.
  - Megvalósítja a megfigyelőt frissítő felületet, így állapotát összehangolhatja az alanyéval.

Együttműködés

- A KonkrétAlany értesíti megfigyelőit, ha olyan változás történik, amelynek eredményeképpen azok állapota eltérne a sajátjától.
- Miután értesült a konkrét alanyban bekövetkezett változásról, a KonkrétMegfigyelő információt kérhet az alanytól, majd ezen információ segítségével összehangolhatja állapotát az alanyéval.

  Az alábbi együttműködési diagram egy alany és két megfigyelő együttműködését illusztrálja:



Megfigyelhetjük, hogyan halasztja el a módosítást kezdeményező Megfigyelő objektum saját frissítését, amíg értesítést nem kap az alanytól. Az Értesít (Notify) műveletet nem mindig az alany hívja meg; meghívhatja egy megfigyelő vagy egy teljesen más típusú objektum is. A Megvalósítás részben erre is megnézünk néhány lehetőséget.

Következmények

A Megfigyelő minta lehetővé teszi, hogy az alanyokat és megfigyelőket egymástól függetlenül módosítsuk. Az alanyok megfigyelőik felhasználása nélkül is újrahasznosíthatók, és viszont. Így az alany vagy más megfigyelők megváltoztatása nélkül hozhatunk létre új megfigyelőket.

A Megfigyelő minta előnyei és hátrányai többek között a következők:

1. *Elvont csatolás az alany és a megfigyelő között.* Az alany csupán annyit tud, hogy megfigyelői vannak, amelyek mindegyike megfelel az elvont Megfigyelő osztály egyszerű felületének, konkrét osztályukat egyáltalán nem ismeri. Így az alanyok és megfigyelők csatolása laza és elvont.
   Miután az Alany és a Megfigyelő nem szoros csatolással rendelkezik, a rendszer másmás elvont fogalmi szintjéhez tartozhatnak. Az alsóbb szintű alany társaloghat a magasabb szintű megfigyelővel, így a rendszer rétegezettsége érintetlen marad. Ha az alany és a megfigyelő összekapcsolódna, az előálló objektumnak vagy két réteget kellene átfognia (amivel megsértené a rétegezettséget), vagy választania kellene, melyik rétegben kíván „élni" (ami a fogalmi rétegeket tenné tönkre).
2. *Az üzenetszórás támogatása.* A szokványos kérelmektől eltérően az alany által kiküldött értesítésnek nem kell meghatároznia fogadóját. Az értesítés automatikusan, üzenetszórással (broadcast) jut el minden érdekelt előfizető objektumhoz. Ezek száma az alany számára érdektelen; az ő feladata csupán annyi, hogy értesítse megfigyelőit. Így bármikor új megfigyelőket adhatunk a rendszerhez, vagy vehetünk el onnan; az értesítések kezelése vagy figyelmen kívül hagyása a megfigyelők felelőssége.
3. *Váratlan frissítések.* Miután a megfigyelők nem tudnak egymás jelenlétéről, az alany módosításának végső költségéről sincs fogalmuk. Egy látszólag jelentéktelen művelet az alanyon frissítések láncolatát indíthatja el a megfigyelők és az azoktól függő objektumok között. Emellett a nem pontosan meghatározott vagy fenntartott függőségi feltételek zavaros frissítésekhez vezethetnek, amelyek oka nehezen felderíthető.
   A gondot súlyosbítja az a tény, hogy egy egyszerű frissítési protokoll nem árul el részleteket arról, mi változott meg az alanyban, így ha más protokollok nem segítik a megfigyelőket, a változások okának felderítése kemény dió lehet számukra.

Megvalósítás

Ebben a részben számos, a függőségi rendszer megvalósításával kapcsolatos kérdéssel foglalkozunk:

1. *Az alanyok hozzárendelése a megfigyelőkhöz.* Az alanyok legegyszerűbben úgy követhetik nyomon az értesítendő megfigyelőket, ha kifejezett hivatkozásokat tárolnak rájuk. Ez azonban túl költséges lehet, ha sok alanyunk,de kevés megfigyelőnk van. Megoldást jelenthet, ha a kisebb tárhelyért idővel fizetünk, és valamilyen társításos (asszociatív) keresést, például egy hasító- vagy kivonattáblát (hash) alkalmazunk az alanyok és megfigyelők egymáshoz rendelésére, így a megfigyelővel nem rendelkező alanyok nem jelentenek társzükséglet-többletet. Másfelől azonban ez a megközelítés növeli a megfigyelők elérésének költségét.
2. *Egynél több alany megfigyelése.* Egyes helyzetekben célszerű lehet, ha a megfigyelő egynél több alanytól függ, például ha egy táblázat több adatforráshoz kapcsolódik. Ilyen esetekben a Frissít (Update) felületet ki kell bővítenünk, hogy a megfigyelő tudja, *melyik* alanytól kapta az értesítést. Az alany egyszerűen átadja magát a Frissít művelet paramétereként, így a megfigyelő tudni fogja, melyik alanyt kell megvizsgálnia.
3. *Ki indítja el a frissítést?* Az alany és megfigyelői az értesítések segítségével maradnak összhangban. De vajon melyik objektum hívja meg az Értesít műveletet a frissítéshez? Két lehetőség áll rendelkezésre:
   (a) Az alany állapot-beállító műveleteivel hívatjuk meg az Értesít műveletet, miután megváltoztatták az alany állapotát. E megközelítés előnye, hogy nem az ügyfeleknek kell emlékezniük az Értesít meghívására, hátránya pedig, hogy több egymást követő művelet több egymást követő frissítést von maga után, ami esetleg nem túl hatékony.
   (b) Az ügyfelek felelősségévé tesszük, hogy a megfelelő időben meghívják az Értesít műveletet. Ennek előnye, hogy az ügyfél több állapotváltozást is megvárhat, mire elindítja a frissítést, így elkerülhetők a felesleges köztes frissítések. A megközelítés hátulütője, hogy az ügyfelek felelőssége a frissítés elindítása, ami azok „feledékenysége" miatt valószínűbbé teszi a hibák bekövetkeztét.
4. *Törölt alanyokra mutató árva hivatkozások.* Az alanyok törlése nem szabad, hogy elárvult hivatkozásokat eredményezzen megfigyelőikben. Az árva hivatkozások elkerülésének egyik módja, ha az alany értesíti a megfigyelőket törléséről, így azok megszüntethetik a rá mutató hivatkozást. Pusztán a megfigyelők törlése általában nem lehetséges, mert más objektumok hivatkozhatnak rájuk, illetve ők maguk több alanyt is megfigyelhetnek.
5. *Az állapot következetességének biztosítása az alanyban az értesítés előtt.* Fontos, hogy az Értesít meghívása előtt ellenőrizzük, hogy az Alany állapota következetes-e, mert a megfigyelők saját frissítésükhöz lekérdezik az alany aktuális állapotát.
   E szabályt akaratlanul is könnyen áthághatjuk, ha az Alany alosztályainak műveletei örökölt műveleteket hívnak meg. Az alábbi kódban szereplő értesítés útnak indítására például akkor kerül sor, amikor az alany állapota nem megfelelő:

```
void MySubject::Operation (int newValue) {
        BaseClassSubject::Operation(newValue);
                //értesítés elindítása

        _myInstVar += newValue;
                //az alosztály állapotának frissítése (túl késő!)
}
```

Ezt a buktatót úgy kerülhetjük el, ha sablonfüggvényekből küldünk értesítést az elvont Alany osztályokban. (A Sablonfüggvény mintát a fejezetben később tárgyaljuk.) Határozzunk meg egy egyszerű műveletet, amelyet az alosztályok majd felülírnak, az Értesítet pedig tegyük a sablonfüggvény utolsó műveletévé; ezzel biztosítjuk, hogy az objektum állapota megfelelő lesz, amikor az alosztályok felülbírálják az Alany műveleteit.

```
void Text::Cut (TextRange r) {
        ReplaceChange(r);    // az alosztályok felülírják
        Notify();
}
```

Mindig jó ötlet rögzíteni, hogy az Alany mely műveletei indítják el az értesítéseket.

6. *A megfigyelőfüggő frissítési protokollok elkerülése: a húzó és toló modell.* A Megfigyelő minta különböző megvalósításaiban az alany gyakran kiegészítő információkat is mellékel a változást jelző értesítéshez, amelyeket a Frissít művelet argumentumaként ad át. Az információ mennyisége megvalósításonként jelentősen eltérhet.
   Az egyik véglet, ha az alany részletes információkkal látja el a megfigyelőket, akár kérik azok, akár nem. Ezt hívjuk **toló modellnek** (push model). A másik véglet a **húzó modell** (pull model); ekkor az alany semmit nem küld az értesítésen kívül, és a megfigyelőknek kell kifejezetten érdeklődniük a részletek felől.
   A húzó modellben az alany nem törődik a megfigyelőkkel, míg a toló modell feltételezi, hogy az alanyok legalább részben ismerik megfigyelőik igényeit. Az utóbbiban a megfigyelők újrahasznosítása nehezebb, mert az Alany osztályok olyan feltételezésekkel élnek a Megfigyelő osztályokról, amelyek nem minden esetben bizonyulnak igaznak. Más részről viszont a húzó modell kevésbé hatékony, hiszen a megfigyelőknek az alany segítsége nélkül kell megállapítaniuk, mi változott meg.
7. *Az „érdekes" módosítások kifejezett meghatározása.* Növelhetjük a frissítés hatékonyságát, ha az alany bejegyző felületét úgy bővítjük ki, hogy megengedje olyan megfigyelők bejegyzését is, amelyek csak bizonyos eseményeket figyelnek. Amikor egy esemény bekövetkezik, az alany csak azokat a megfigyelőket értesíti, amelyek bejegyzés szerint az adott esemény iránt „érdeklődnek". E megoldás támogatásának egyik módja az **aspektusok** (szempont, tulajdonság) használata a Subject objektumokban. A megfigyelők ekkor a következő módon kapcsolódnak az alanyhoz és jelentik be, hogy egy bizonyos esemény érdekli őket:

```
void Subject::Attach(Observer*, Aspect& interest);
```

A fenti kódban az `interest` határozza meg az „érdekes" eseményt. Értesítéskor az alany a Frissít (Update) művelet paramétereként adja meg a megfigyelőknek, mely aspektusa változott meg:

```
void Observer::Update(Subject*, Aspect& interest);
```

8. *Az összetett frissítések egységbe zárása.* Ha az alanyok és megfigyelők közötti függőségi kapcsolat különösen összetett, külön objektumra lehet szükség, amely gondoskodik e kapcsolatról. Az ilyen objektumokat nevezzük **VáltozásKezelőnek** (ChangeManager). A VáltozásKezelő célja, hogy minimálisra csökkenthessük azt a munkát, ami ahhoz szükséges, hogy az alanyban beállt változást tükrözhessük a megfigyelőkben. Ha egy művelet például számos, egymástól függő alany megváltozását vonja maga után, szükség lehet arra, hogy biztosítsuk, a megfigyelők értesítésére csak azután kerül sor, hogy *minden* alany módosítása megtörtént, így elkerülhetjük a megfigyelők többszöri értesítését.

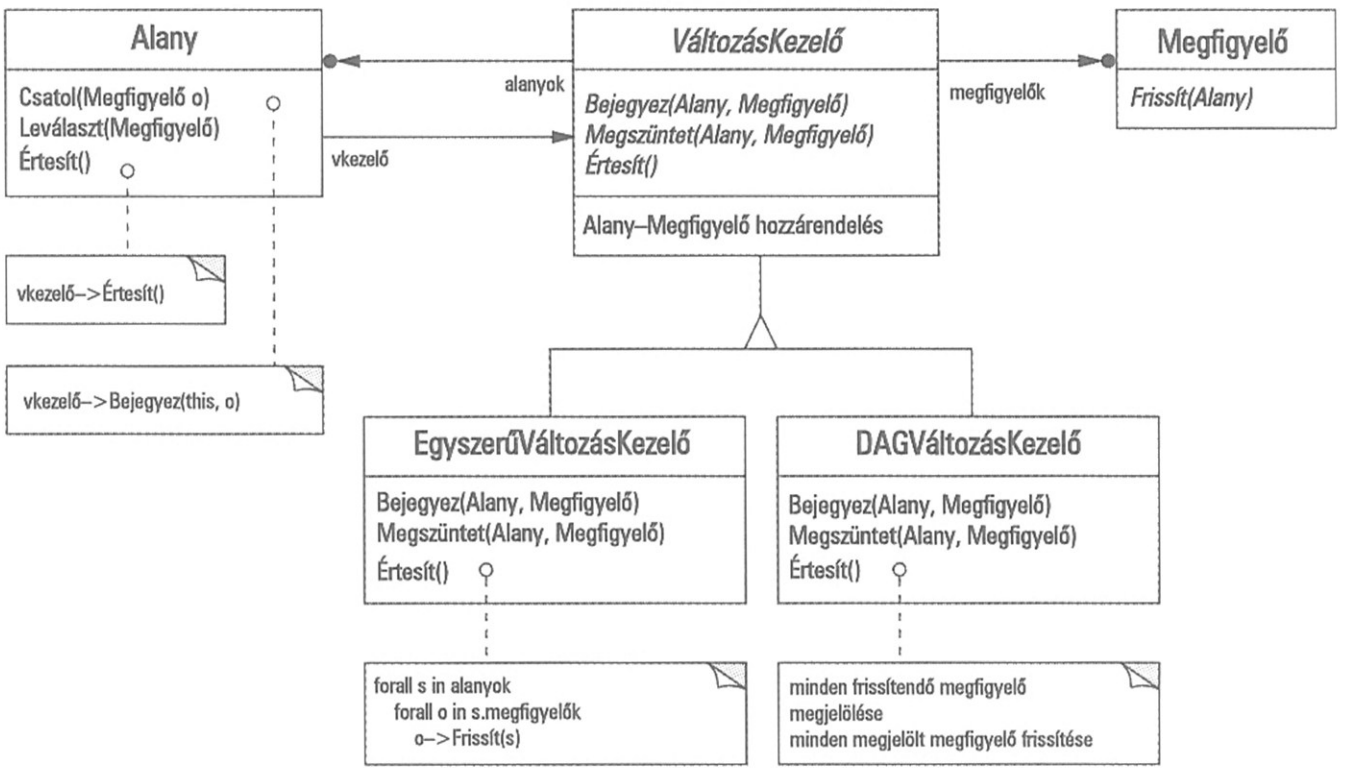
   A változáskezelőnek három feladata van:

   (a) Az alany összekapcsolása a megfigyelőkkel, és felület biztosítása e kapcsolat fenntartásához. Ez megszünteti annak szükségességét, hogy az alanyoknak hivatkozásokat kelljen fenntartaniuk a megfigyelőkre, és viszont.

   (b) Egy adott frissítési stratégia meghatározása.

   (c) Az alany kérésére az összes függő megfigyelő frissítése.

   A következő diagram a Megfigyelő minta egy egyszerű, változáskezelő alapú megvalósítását mutatja. Itt két egyedi célú változáskezelőt találunk. Az EgyszerűVáltozásKezelő (SimpleChangeManager) „naiv", mindig frissíti az alanyhoz tartozó valamennyi megfigyelőt. Ezzel szemben a DAGVáltozásKezelő (DAGChangeManager) az alanyok és megfigyelőik függőségeinek irányított körmentes gráfjait kezeli. Ha egy megfigyelő több alanyt is megfigyel, a DAGVáltozásKezelő a jobb választás. Ekkor a kettő vagy több alanyban bekövetkező változások felesleges frissítéseket okozhatnak, a DAGVáltozásKezelő viszont gondoskodik róla, hogy a megfigyelő frissítése csak egyszer történjen meg. Ha a többszöri frissítés veszélye nem merül fel, az EgyszerűVáltozásKezelő is megfelel.



A VáltozásKezelő a Közvetítő mintára mutat példát. Általában csak egy van belőle, és a programban globálisan „ismert". Ehhez az Egyke minta nyújthat segítséget.

9. *Az Alany és Megfigyelő osztályok egyesítése.* A többszörös öröklést nem tartalmazó programnyelveken (ilyen például a Smalltalk) írt osztálykönyvtárak általában nem határoznak meg külön Alany és Megfigyelő osztályokat, hanem azok felületét egyetlen osztályban egyesítik. Így olyan objektumokat hozhatunk létre, amelyek egyszerre alanyként és megfigyelőként is viselkedhetnek, többszörös öröklés nélkül. A Smalltalk nyelvben például az Alany (Subject) és Megfigyelő (Observer) felületeket az Object gyökérosztály határozza meg, így valamennyi osztály számára elérhetők.

Példakód

A Megfigyelő (Observer) felületet egy elvont osztály határozza meg:

```
class Subject;

class Observer {
public:
   virtual ~Observer();
   virtual void Update(Subject* theChangedSubject) = 0;
protected:
   Observer();
};
```

Ez a megvalósítás minden megfigyelő esetében több alanyt (Subject) támogat. Az `Update` (Frissít) műveletnek átadott alany lehetővé teszi a megfigyelő számára, hogy amennyiben több alanyt is megfigyel, megállapíthassa, melyik változott meg.

Ehhez hasonlóan az Alany (Subject) felületét is egy elvont osztály határozza meg:

```
class Subject {
public:
   virtual ~Subject();

   virtual void Attach(Observer*);
   virtual void Detach(Observer*);
   virtual void Notify();
protected:
   Subject();
private:
   List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
   _observers->Append(o);
}
```

```
void Subject::Detach (Observer* o) {
   _observers->Remove(o);
}

void Subject::Notify () {
   ListIterator<Observer*> i(_observers);

   for (i.First(); !i.IsDone(); i.Next()) {
         i.CurrentItem()->Update(this);
   }
}
```

A `ClockTimer` (ÓraIdőzítő) egy konkrét alany, amely az aktuális időt tárolja, és a megfigyelőket másodpercenként értesíti az idő változásáról. Az egyes időegységek – óra, perc, másodperc – lekérdezésére a `ClockTimer` felület szolgál.

```
class ClockTimer : public Subject {
public:
   ClockTimer();

   virtual int GetHour();
   virtual int GetMinute();
   virtual int GetSecond();

   void Tick();
};
```

A `Tick` (Ketyeg) műveletet egy belső időzítő hívja meg szabályos időközönként. A `Tick` frissíti a `ClockTimer` belső állapotát, és meghívja a `Notify` (Értesít) műveletet, hogy értesítse a megfigyelőket a változásról:

```
void ClockTimer::Tick () {
   // a belső idő frissítése
   //...
   Notify();
}
```

Most meghatározhatunk egy `DigitalClock` (DigitálisÓra) nevű osztályt, amely megjeleníti az időt. Az osztály a grafikus megjelenítéshez a `Widget` (Vezérlő) osztálytól örököl, amelyet a felhasználói felület elemkészlete biztosít. A megfigyelő felületet az `Observer` osztálytól való örökléssel a `DigitalClock` felületbe „keverjük".

```
class DigitalClock: public Widget, public Observer {
public:
   DigitalClock(ClockTimer*);
   virtual ~DigitalClock();
```

```
    virtual void Update(Subject*);
            // az Observer műveletének felülírása

    virtual void Draw();
            // a Widget műveletének felülírása;
            // a digitális óra kirajzolását határozza meg
private:
    ClockTimer* _subject
};

DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this);
}

DigitalClock::~DigitalClock () {
    _subject->Detach(this);
}
```

Mielőtt az `Update` művelet megrajzolja az óra számlapját, ellenőrzi, hogy az értesítő alany az óra alanya-e:

```
void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
            Draw();
    }
}

void DigitalClock::Draw () {
    // az új értékek lekérése az alanytól

    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // stb.

    // a digitális óra megrajzolása
}
```

Egy analóg óra (`AnalogClock` osztály) ehhez hasonlóan határozható meg:

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    //...
};
```

Az alábbi kód egy analóg és egy digitális órát hoz létre, amelyek mindig ugyanazt az időt mutatják:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

A két óra a `timer` minden „ütésénél" frissül, és újra kirajzolja magát.

Ismert felhasználások

A Megfigyelő minta első és talán legismertebb alkalmazása a Smalltalk felhasználói felületi keretrendszere, a modell–nézet–vezérlő (Model/View/Controller, MVC) megoldás [KP88]. Itt az MVC Model osztálya az alany, míg a View a megfigyelők alaposztálya. A Smalltalk, az ET++ [WGM88] és a THINK osztálykönyvtár [Sym93b] egy általános függőségkezelő megoldást tartalmaznak: az Alany és Megfigyelő felületeket a rendszer valamennyi osztályának szülőosztályába helyezték.

Ugyanezt a mintát alkalmazza az InterViews [LVC89], az Andrew Toolkit [P+88], és a Unidraw [VL90] felhasználói felületi elemkészlet is. Az InterViews kifejezett Observer (megfigyelő) és Observable (megfigyelhető, vagyis alany) osztályokat határoz meg, az Andrew „nézeteket" és „adatobjektumokat"; a Unidraw a grafikus szerkesztői objektumokat View (megfigyelői) és Subject (alany) részekre bontja.

Kapcsolódó minták

Közvetítő: Az összetett frissítések egységbe zárásával a VáltozásKezelő közvetítőként működik az alanyok és megfigyelők között.

Egyke: A VáltozásKezelő az Egyke minta segítségével egyedivé és globálisan elérhetővé tehető.

Állapot

Viselkedési objektumminta

Cél

Egy adott objektum számára engedélyezni, hogy belső állapotának megváltozásával megváltoztathassa viselkedését is. Az objektum ekkor látszólag módosítja az osztályát.

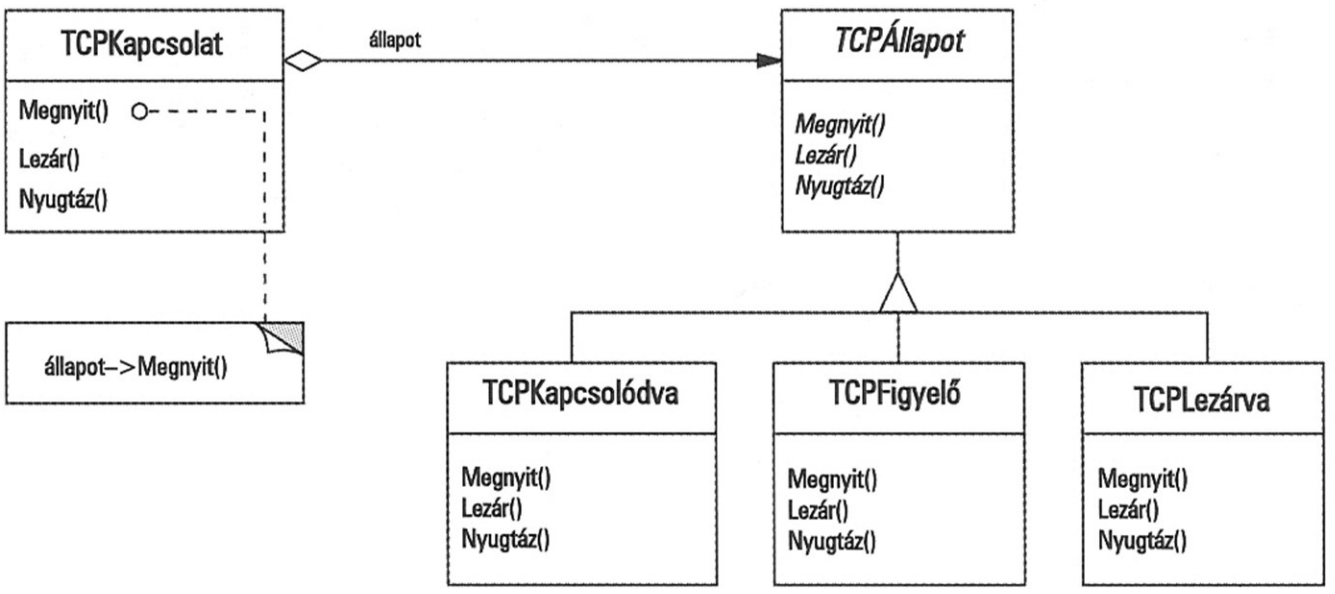
Egyéb nevek

State, Objects for States (Állapotobjektumok)

Feladat

Vegyünk egy TCPKapcsolat (TCPConnection) nevű osztályt, amely egy hálózati kapcsolatot ábrázol. A TCPKapcsolat objektum a következő állapotok valamelyikében lehet: Kapcsolódva (Established), Figyelő (Listening), Lezárva (Closed). Amikor az objektum kérelmeket fogad más objektumoktól, aktuális állapotától függően más-más válaszokat adhat. Egy Megnyit (Open) kérelem hatása például attól függhet, hogy a kapcsolat éppen Lezárva vagy Kapcsolódva állapotban van-e. Az Állapot tervezési minta leírja, hogyan mutathat a TCPKapcsolat az egyes állapotokban különböző viselkedéseket.

A minta kulcsa egy TCPÁllapot (TCPState) nevű elvont osztály bevezetése, ami a hálózati kapcsolat állapotait ábrázolja. A TCPÁllapot a különböző működési állapotokhoz tartozó osztályok közös felületét írja le, alosztályai pedig az állapottól függő viselkedést. A TCPKapcsolódva (TCPEstablished) és a TCPLezárva (TCPClosed) például a TCPkapcsolat Kapcsolódva és Lezárva állapotaira jellemző működést valósítják meg.



A TCPKapcsolat osztály egy állapotobjektumot tart fenn (a TCPÁllapot valamelyik alosztályának egy példányát), ami a TCP kapcsolat aktuális állapotát jelöli. Az osztály minden állapotfüggő kérelmet ehhez az objektumhoz továbbít, az pedig végrehajtja a kapcsolat állapotának megfelelő műveleteket.

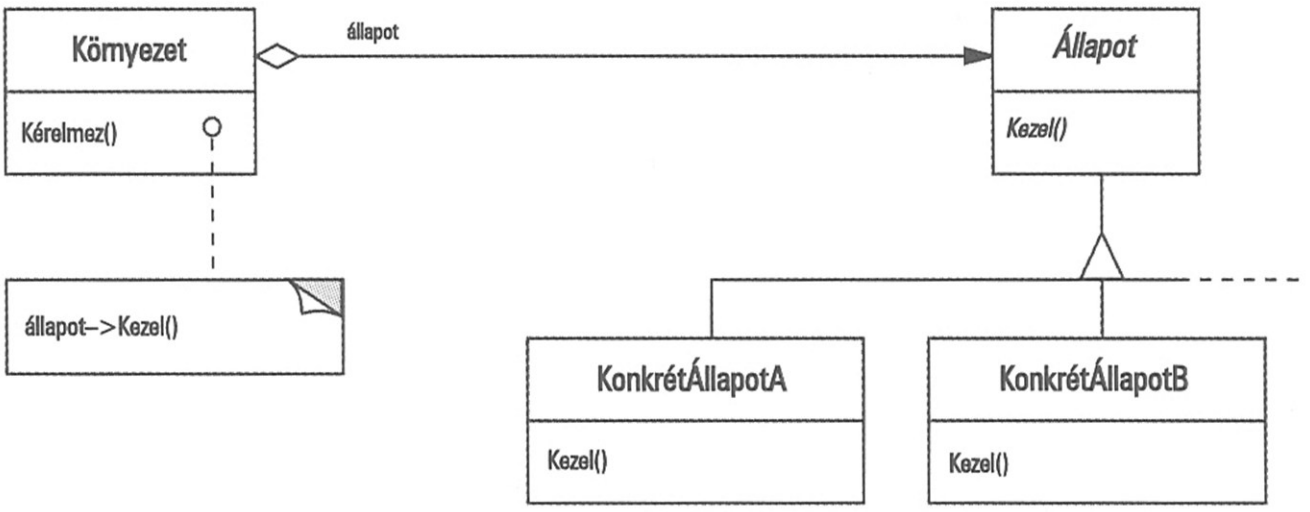
Amikor a kapcsolat állapota megváltozik, a TCPKapcsolat a megfelelő állapotobjektumra cseréli az éppen használtat; ha a kapcsolatot például lezárjuk, a TCPKapcsolódva példányt egy TCPLezárva példánnyal helyettesíti.

Alkalmazhatóság

Az Állapot minta a következő esetekben alkalmazható:

- Egy objektum viselkedése az állapotától függ, és ezen viselkedést futásidőben az állapotnak megfelelően kell változtatnia.
- A műveletekben hosszú, több részből álló feltételes utasítások találhatók, amelyek az objektum állapotától függnek. Az állapotot általában egy vagy több felsoroló állandó jelöli. Számos művelet ugyanazt a feltételes szerkezetet használja. Az Állapot minta a feltétel minden ágát külön osztályba helyezi, így az objektum állapotát is önálló objektumként kezelhetjük, amely a többi objektumtól függetlenül módosítható.

Szerkezet



Résztvevők

- Környezet (TCPKapcsolat)
  - Meghatározza az ügyfelek számára érdekes felületet.
  - Egy KonkrétÁllapot (ConcreteState) példányt tart fenn, amely meghatározza az aktuális állapotot.

- **Állapot** (TCPÁllapot)
  - Felületet határoz meg a Környezet (Context) egy adott állapotához kötődő viselkedés egységbe zárásához.
- **KonkrétÁllapot alosztályok** (TCPKapcsolódva, TCPFigyelő, TCPLezárva)
  - A Környezet egy-egy állapotához kapcsolódó viselkedést valósítják meg.

Együttműködés

- A Környezet az állapotfüggő kérelmek kezelését átruházza az aktuális KonkrétÁllapot objektumra.
- A Környezet argumentumként átadhatja magát a kérelmet kezelő állapotobjektumnak, így az tudomást szerezhet a környezetről, ha szükséges.
- A Környezet az ügyfelek elsődleges felülete. Az ügyfelek Állapot objektumokkal beállíthatják a környezetet, ezután nem kell közvetlenül foglalkozniuk az állapotobjektumokkal.
- Az állapotok sorrendjét, illetve az azt befolyásoló körülményeket a Környezet, illetve a KonkrétÁllapot alosztályok is meghatározhatják.

Következmények

Az Állapot tervezési minta előnyei és hátrányai a következők:

1. *Meghatározza és elkülöníti az egyes állapotokhoz tartozó viselkedéseket.* Az Állapot mintában az egy adott állapothoz tartozó valamennyi művelet egyetlen objektumba kerül. Miután minden állapotfüggő kód egy Állapot alosztályban található, új alosztályok készítésével könnyen vehetünk fel új állapotokat és átmeneteket.
   Egy másik megoldás, ha a belső állapotok meghatározására adatértékeket használunk, amelyeket a Környezet műveletei ellenőriznek. Ekkor azonban több egyforma feltételes vagy elágazó utasítást kapunk, amelyek a Környezet megvalósításában szétszórva helyzekednek el, így egy új állapot hozzáadása számos művelet módosítását igényelheti, ami megnehezíti a kód karbantartását.
   Az Állapot mintával az említett probléma elkerülhető, viszont éppen a minta miatt beleütközhetünk egy másikba. Az, hogy a minta a különböző állapotokhoz tartozó viselkedéseket több Állapot alosztályba osztja el, növeli az osztályok számát, így a kód kevésbé lesz tömör, mintha egyetlen osztályt használnánk. Az elosztás akkor hasznos, ha sok-sok állapotunk van, amelyek másképp nagy méretű feltételes utasításokat tennének szükségessé.
   A hosszú eljárásokhoz hasonlóan a hosszú feltételes utasítások sem kívánatosak. Tömbszerűek, rontják a kód átláthatóságát, így nehezen módosíthatók vagy bővíthetők. Az Állapot minta jobb megoldást nyújt az állapotfüggő kód szervezésére. Az állapotátmeneteket vezérlő logika ebben a mintában nem monolitikus `if` vagy `switch` utasításokban található, hanem eloszlik az Állapot alosztályok között. Azzal,

hogy minden állapotátmenetet és műveletet egy osztályba tokozunk, a működési állapotot egy teljes értékű objektum szintjére emeljük, ami világossá teszi a kód szerkezetét és célját.

2. *Világossá teszi az állapotátmeneteket.* Ha egy objektum az aktuális állapotát kizárólag belső adatértékekkel fejezi ki, állapotátmeneteinek nem lesz kifejezett ábrázolása; azok csupán egyes változók értékadásaiban jelennek meg. Azzal, hogy az egyes állapotokhoz önálló objektumokat vezetünk be, egyértelműbbé tesszük az átmeneteket.
   Emellett az Állapot objektumok védelmet nyújtanak a környezetnek a következetlen belső állapotok ellen, hiszen a Környezet szemszögéből az állapotátmenetek atomiak – nem több, hanem egyetlen változó (a Környezet Állapot objektumváltozója) értékének módosításával mennek végbe [dCLF93].
3. *Az állapotobjektumok megoszthatók.* Ha az állapotobjektumoknak nincsenek példányváltozói (vagyis az általuk ábrázolt állapotot teljes mértékben a típusuk kódolja), a környezetek közösen is használhatják őket. Az állapotok ilyen megosztott használata lényegében a Pehelysúlyú minta (lásd az előző fejezetben) alkalmazása, ahol belső állapot nincs, csak viselkedés.

Megvalósítás

Az Állapot tervezési minta megvalósításával kapcsolatban a következőkre kell figyelni:

1. *Ki határozza meg az állapotátmeneteket?* A minta nem ad útmutatást arra nézve, hogy melyik résztvevőnek kell meghatároznia az állapotátmenetek követelményeit. Ha a követelmények kötöttek, megvalósításuk teljes egészében történhet a Környezet objektumban. Mindazonáltal általában rugalmasabb és helyesebb megoldás, ha az Állapot alosztályokra bízzuk, hogy maguk határozzák meg, melyik állapot követheti őket, és milyen körülmények között. Ehhez a Környezetet ki kell bővítenünk egy felülettel, amely lehetővé teszi az állapotobjektumok számára, hogy a Környezet állapotát közvetlenül ők állítsák be.
   Az átmenetet vezérlő logika eme felosztása egyszerűbbé teszi a logika új Állapot alosztályokkal történő bővítését vagy módosítását, hátránya viszont, hogy az állapotobjektumok legalább egy társukat ismerni fogják, ami megvalósítási függőségeket okoz az alosztályok között.
2. *Egy táblázat alapú alternatíva.* A *C++ Programming Style* [Car92] című könyvben Cargill az állapotfüggő kód szervezésének egy másik módját írja le: táblázatokat használ a bemenetek és állapotátmenetek egymáshoz rendelésére. Az egyes állapotokhoz tartozó táblázatok minden lehetséges bemenetet egy következő állapothoz rendelnek, amelynek révén a feltételes kódot (és az Állapot minta esetében a virtuális függvényeket) egy táblázatban való kereséssé alakítjuk.

A táblázatok legfőbb előnye a szabályosság; az átmenet követelményeit adatok, és nem programkód módosításával változtathatjuk meg. A megoldásnak természetesen vannak hátrányai is:

- A táblázatban való keresés általában kevésbé hatékony, mint egy (virtuális) függvényhívás.
- Az átmenet logikájának egységes, táblázatos formába rendezése homályossá, nehezen átláthatóvá teszi az átmenet-követelményeket.
- Többnyire nehéz műveleteket adni az állapotátmenetekhez. A táblázatos megközelítés meghatározza az állapotokat és átmeneteiket, de ahhoz, hogy az átmeneteknél különféle számítási műveleteket végezhessünk, a megoldást ki kell bővítenünk.

A táblázat alapú állapotautomaták és az Állapot tervezési minta közötti legfontosabb különbség a következőképpen foglalható össze: az Állapot minta az állapotfüggő viselkedést modellezi, míg a táblázatos megközelítés az állapotátmenetek meghatározására összpontosít.

3. *Az Állapot objektumok létrehozása és megsemmisítése.* A megvalósítás során gyakori, hogy választás elé kerülünk: csak akkor hozzuk létre az állapotobjektumokat, amikor szükség van rájuk, és utána semmisítsük meg őket (1), vagy készítsük el őket előre, és soha ne pusztítsuk el (2)?

   Az első lehetőséget akkor célszerű választani, ha a futásidőben beálló állapotokat nem ismerjük előre, *és* a környezet állapota nem változik gyakran. Így elkerülhetjük olyan objektumok létrehozását, amelyeket később nem is használunk, ami fontos, ha az állapotobjektumok sok információt tárolnak. A második lehetőség mellett akkor célszerű dönteni, ha az állapotváltozások gyorsan követik egymást. Ilyenkor nyilván el szeretnénk kerülni az állapotok megsemmisítését, hiszen rövidesen újra szükség lehet rájuk. Ha ezt a megoldást választjuk, a példányosítás költségeit egyszer, előre kell csak megfizetnünk, a megsemmisítés pedig egyáltalán nem jár költségekkel. Mindazonáltal ez a megközelítés kényelmetlen lehet, mert a környezetnek minden lehetséges állapotra hivatkozást kell fenntartania.

4. *Dinamikus öröklés használata.* Egy adott kérelemhez tartozó viselkedést elvileg módosíthatnánk úgy, hogy az objektum osztályát futásidőben megváltoztatjuk, de ez a legtöbb objektumközpontú programozási nyelvben nem lehetséges. A kivételt a Self [US87] és más átruházás alapú nyelvek jelentik, amelyek biztosítanak ilyen lehetőséget, és ezáltal közvetlenül támogatják az Állapot tervezési mintát. A Self objektumai a dinamikus öröklés egy formáját azzal érik el, hogy átruházhatnak műveleteket más objektumokra. A megbízott futásidejű megváltoztatása módosítja az öröklési szerkezetet; e megoldással az objektumok módosíthatják viselkedésüket, és végeredményben osztályukat.

Példakód

A következő példa a Feladat részben leírt TCP kapcsolat C++ kódja; egyben a TCP protokoll egyszerűsített változata. (Egyszerűsített, mert nem írja le a teljes protokollt, illetve a TCP kapcsolatok valamennyi állapotát.[8])

Először is, meghatározzuk a `TCPConnection` (TCPKapcsolat) osztályt, amely az adatátvitel felületét biztosítja, illetve az állapotváltoztatási kérelmeket kezeli.

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
   TCPConnection();

   void ActiveOpen();
   void PassiveOpen();
   void Close();
   void Send();
   void Acknowledge();
   void Synchronize();

   void ProcessOctet (TCPOctetStream*);
private:
   friend class TCPState;
   void ChangeState(TCPState*);
private:
   TCPState* _state;
};
```

A `TCPConnection` a `TCPState` (TCPÁllapot) osztály egy példányát a `_state` tagváltozóban tárolja. A `TCPState` osztály lemásolja a `TCPConnection` állapotmódosító felületét. Műveletei paraméterként egy `TCPConnection` példányt kapnak, így az osztály hozzáférhet a `TCPConnection` adataihoz, és megváltoztathatja a kapcsolat állapotát.

```
class TCPState {
public:
   virtual void Transmit(TCPConnection*, TCPOctetStream*);
   virtual void ActiveOpen(TCPConnection*);
   virtual void PassiveOpen(TCPConnection*);
   virtual void Close(TCPConnection*);
   virtual void Synchronize(TCPConnection*);
   virtual void Acknowledge(TCPConnection*);
   virtual void Send(TCPConnection*);
protected:
   void ChangeState(TCPConnection*, TCPState*);
};
```

[8] A példa a Lynch és Rose [LR93] által leírt TCP kapcsolati protokollon alapul.

A TCPConnection minden állapotfüggő kérelmet _state TCPState példányára ruház át. A TCPConnection egy másik műveletet is biztosít, amely ezt a változót egy új TCPState-re állítja. A TCPConnection konstruktora az objektumnak a (később meghatározott) TCPClosed (TCPLezárva) állapotot adja kezdőértékül.

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

A TCPState tartalmazza a rá ruházott valamennyi kérelem teljesítéséhez szükséges alapértelmezett viselkedés megvalósítását. Emellett a TCPConnection állapotát is képes megváltoztatni, a ChangeState (VáltoztatÁllapot) művelettel. A TCPState-et a TCPConnection barátjaként vezetjük be, így kivételezett hozzáférést kap ehhez a művelethez.

```
void TCPState::Transmit (TCPConnection*, TCPOctetStream*) {}
void TCPState::ActiveOpen (TCPConnection*) {}
void TCPState::PassiveOpen (TCPConnection*) {}
void TCPState::Close (TCPConnection*) {}
void TCPState::Synchronize (TCPConnection*) {}

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    _t->ChangeState(s);
}
```

A TCPState alosztályai állapotfüggő viselkedést valósítanak meg. A TCP kapcsolat számos állapotot vehet fel – Kapcsolódva, Figyelő, Lezárva stb. –, és mindegyikhez egy-egy TCPState alosztály tartozik. Ezek közül hármat tárgyalunk részletesen: a TCPEstablished (TCPKapcsolódva), a TCPListen (TCPFigyelő), illetve a TCPClosed (TCPLezárva) osztályokat.

```
class TCPEstablished : public TCPState {
public:
   static TCPState* Instance();

   virtual void Transmit(TCPConnection*, TCPOctetStream*);
   virtual void Close(TCPConnection*);
};

class TCPListen : public TCPState {
public:
   static TCPState* Instance();

   virtual void Send(TCPConnection*);
   //...
};

class TCPClosed : public TCPState {
public:
   static TCPState* Instance();

   virtual void ActiveOpen(TCPConnection*);
   virtual void PassiveOpen(TCPConnection*);
   //...
};
```

A TCPState alosztályok helyi állapotinformációt nem tartanak fenn, így megoszthatók, és mindegyikből csak egy példányra van szükség. Az egyedi példányokat a statikus Instance (Példány) művelettel szerezzük meg.[9]

Az egyes TCPState alosztályok az állapothoz tartozó érvényes kérelmek állapotfüggő viselkedését valósítják meg:

```
void TCPClosed::ActiveOpen (TCPConnection* t) {
   // SYN küldése, SYN, ACK stb. fogadása

   ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen (TCPConnection* t) {
   ChangeState(t, TCPListen::Instance());
}
```

[9] Ennek révén a TCPState alosztályok az Egyke tervezési mintát követik. (Lásd az előző fejezetet.)

```
void TCPEstablished::Close (TCPConnection* t) {
   // FIN küldése, a FIN nyugtázásának (ACK) fogadása

   ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
   t->ProcessOctet(o);
}

void TCPListen::Send (TCPConnection* t) {
   // SYN küldése, SYN, ACK stb. fogadása

   ChangeState(t, TCPEstablished::Instance());
}
```

Az állapothoz kapcsolódó munka elvégzése után ezek a műveletek a `ChangeState` meghívásával megváltoztatják a `TCPConnection` állapotát. Maga a `TCPConnection` semmit sem tud a TCP kapcsolati protokollról; az állapotátmeneteket és TCP műveleteket a `TCPState` alosztályok határozzák meg.

Ismert felhasználások

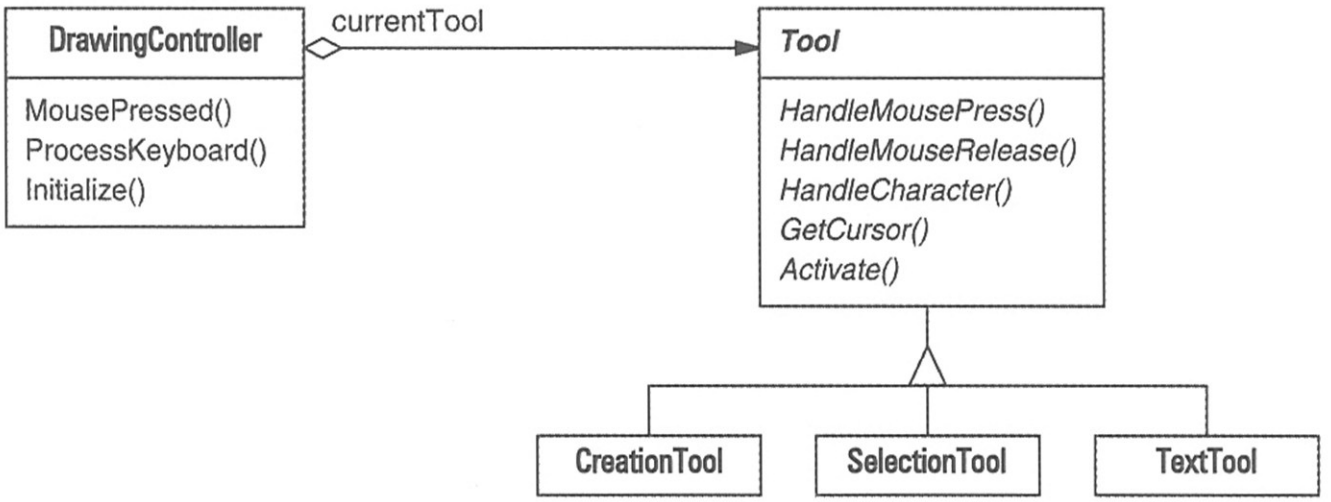
Az Állapot mintát és alkalmazását a TCP kapcsolati protokollokra Johnson és Zweig írták le [JZ91].

A legtöbb népszerű interaktív rajzolóprogram „eszközöket" biztosít a közvetlen művelet-végzéshez, például egy vonalrajzoló eszköz segítségével a vonal pusztán kattintással és húzással megrajzolható, egy kijelölő eszközzel pedig ugyanígy alakzatokat jelölhetünk ki. Az eszközök általában egy palettán kapnak helyet, ahol választhatunk közülük. A felhasználó úgy gondol rá, mintha felvenne egy eszközt és használatba venné, pedig a valóság az, hogy a szerkesztőprogram viselkedése változik meg a kijelölt eszköznek megfelelően. Ha egy rajzoló eszköz aktív, alakzatokat hozhatunk létre, ha egy kijelölő, akkor kijelölhetünk, és így tovább. A viselkedés megváltoztatása a választott eszköznek megfelelően az Állapot minta segítségével lehetséges.

Meghatározhatunk egy elvont Eszköz (Tool) osztályt, amelyből alosztályokat származtatva megvalósíthatjuk az egyes eszközöknek megfelelő viselkedéseket. A program nyomon követi, melyik az aktuális Eszköz objektum, és a kérelmeket hozzá irányítja. Ha a felhasználó másik eszközt választ, az objektum kicserélődik, és így a program viselkedése is megváltozik.

Ezt a megoldást alkalmazza mind a HotDraw [Joh92], mind a Unidraw [VL90] rajzoló keretrendszer, és lehetővé teszik a felhasználónak, hogy könnyedén készítsen új eszközöket.

A HotDraw-ban a DrawingController (RajzVezérlő) osztály továbbítja a kérelmeket az aktuális Tool objektumnak, míg a Unidraw megfelelő osztályai a Viewer (Figyelő) és a Tool. Az alábbi osztálydiagram a Tool és a DrawingController felületek vázlatát mutatja:



Coplien [Cop92] Envelope-Letter (boríték–levél) megoldása is hasonló az Állapot mintához, és azt teszi lehetővé, hogy egy objektum osztályát futásidőben megváltoztathassuk. Az Állapot minta ennél rögzítettebb, és arra összpontosít, hogyan kezelhetünk egy objektumot, amelynek viselkedése az állapotától függ.

Kapcsolódó minták

A Pehelysúlyú minta megmutatja, mikor és hogyan oszthatók meg az állapotobjektumok.

Az állapotobjektumok gyakran az Egyke mintát követik.

Stratégia

Viselkedési objektumminta

Cél

Algoritmus-család meghatározása, melyben az algoritmusokat egyenként egységbe zárjuk és egymással felcserélhetővé tesszük. E módszer révén az algoritmus az ügyféltől függetlenül módosítható.

Egyéb nevek

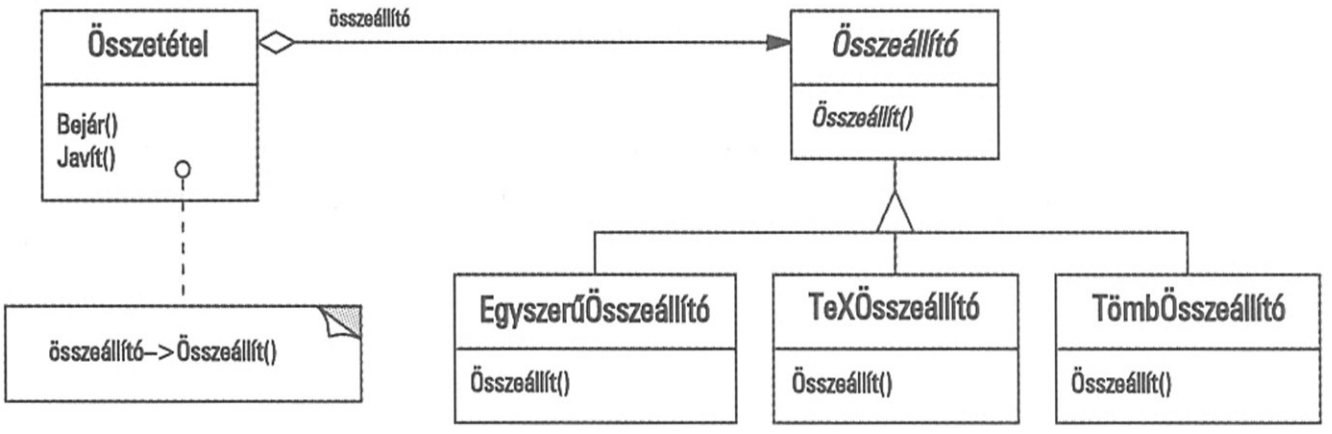
Strategy, Policy

Feladat

Szövegfolyamok sorokra tördelésére számos algoritmus létezik, de ezek „bedrótozása" azon osztályokba, amelyeknek szükségük van rájuk, több okból sem kívánatos:

- A sortörést igénylő ügyfelek túl naggyá, bonyolulttá és nehezen karbantarthatóvá válhatnak, ha tartalmazzák a sortörő kódot is, különösen ha több sortörő algoritmust is támogatnak.
- A különböző algoritmusokra különböző helyzetekben lehet szükség. Több sortörő algoritmust nem célszerű támogatni, ha nem használjuk mindegyiket.
- Nehéz új algoritmusokat felvenni, illetve a meglevőket módosítani, ha a sortörő kód az ügyfél szerves része.

A fenti gondokat úgy orvosolhatjuk, ha az egyes sortörő algoritmusokat egységbe záró osztályokat határozunk meg. Az így egységbe zárt algoritmusokat nevezzük **stratégiáknak**.



Tegyük fel, hogy egy Összetétel (Composition) nevű osztály felelős egy szövegnézőben megjelenített szöveg sortöréseinek kezeléséért és frissítéséért. A sortörő stratégiákat nem ez az osztály valósítja meg, hanem külön-külön az elvont Összeállító (Compositor) osztály alosztályai. Tehát az Összeállító alosztályokban különböző stratégiák megvalósításai találhatók:

- Az **EgyszerűÖsszeállító** (SimpleCompositor) egy egyszerű stratégiát nyújt, amely egyszerre egy sortörést határoz meg.
- A **TeXÖsszeállító** (TeXCompositor) a TeX algoritmust valósítja meg a sortörések megkeresése céljából. Ez a stratégia általános teljesítményfokozásra törekszik, így egyszerre mindig több sort, egy bekezdést vizsgál.
- A **TömbÖsszeállító** (ArrayCompositor) olyan stratégiát valósít meg, ami úgy töri meg a sorokat, hogy azokban egyenlő számú elem legyen. Ez például egy ikongyűjtemény sorokra tördelésénél lehet hasznos.

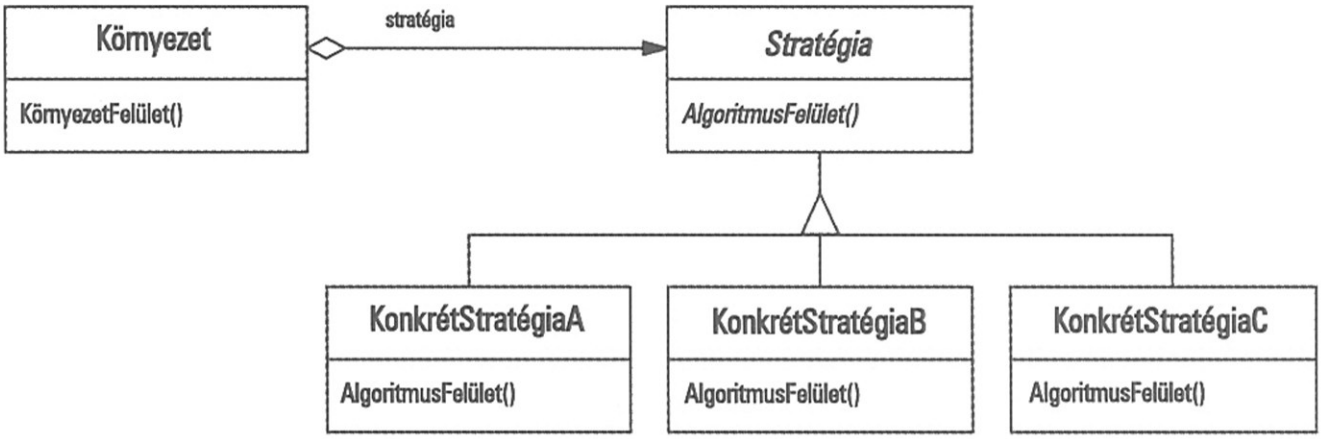
Az Összetétel objektumok hivatkozást tárolnak az Összeállító objektumokra. Amikor egy Összetétel újraformázza a szöveget, ezt a feladatot Összeállító objektumának továbbítja. Az Összetétel ügyfele úgy határozza meg, melyik objektumot kell használni, hogy a kívánt Összeállítót az Összetételbe telepíti.

Alkalmazhatóság

A Stratégia tervezési minta használata az alábbi esetekben célszerű:

- Számos, kapcsolatban álló osztály csak a viselkedésében különbözik egymástól. A stratégiák lehetővé teszik, hogy egy osztályhoz több közül egy adott viselkedést rendeljünk.
- Egy algoritmus több változatára van szükségünk, például különböző algoritmusokat határozunk meg aszerint, hogy az idő vagy a tárhely a fontosabb. A stratégiák jó szolgálatot tehetnek, ha a különböző változatok megvalósításai az algoritmusok osztályhierarchiáját alkotják [HO87].
- Egy algoritmus olyan adatokat használ, amelyekről az ügyfeleknek nem szabad tudniuk. A Stratégia minta alkalmazásával elkerülhető az összetett, algoritmusfüggő adatszerkezetek felfedése.
- Egy osztály többféle viselkedést határoz meg, és ezek műveleteiben többágú feltételes utasításokként jelentkeznek. A feltételágak helyett használjunk önálló stratégiaosztályokat.

Szerkezet



Résztvevők

- **Stratégia** (Összeállító)
  - Közös felületet határoz meg a támogatott algoritmusok számára. A Környezet (Context) ezt a felületet használja a KonkrétStratégia (ConcreteStrategy) által meghatározott algoritmus meghívására.
- **KonkrétStratégia** (EgyszerűÖsszeállító, TeXÖsszeállító, TömbÖsszeállító)
  - Megvalósítják az algoritmust a Stratégia felület segítségével.
- **Környezet** (Összetétel)
  - Egy KonkrétStratégia objektum állítja be.
  - Hivatkozást tart fenn egy Stratégia objektumra.
  - Meghatározhat egy felületet, amelyen keresztül a Stratégia hozzáférhet a műveleteihez.

Együttműködés

- A Stratégia és a Környezet objektumok együtt valósítják meg a választott algoritmust. A környezet minden, az algoritmus által igényelt adatot átadhat a stratégiának, amikor az algoritmus meghívására sor kerül, de önmagát is átadhatja argumentumként a Stratégia műveleteinek, így a stratégia szükség esetén visszahívhatja a környezetet.
- A környezet az ügyfeleitől érkező kérelmeket a hozzá tartozó stratégiához továbbítja. Az ügyfelek általában létrehoznak egy KonkrétStratégia objektumot, amelyet átadnak a környezetnek; ezt követően kizárólag a környezettel tartanak kapcsolatot. Az ügyfél számára többnyire KonkrétStratégia osztályok egész családja áll rendelkezésre, amelyből választhat.

Következmények

A Stratégia minta előnyei és hátrányai a következők:

1. *A rokon algoritmusokból családok alkothatók.* A Stratégia osztályok hierarchiája algoritmusok, illetve viselkedések családját alkotja, amelyet a környezetek újrahasznosíthatnak. A minden algoritmusban jelen levő szolgáltatásokat örökléssel biztosíthatjuk.
2. *Alternatívát nyújt az alosztályok létrehozásával szemben.* Több algoritmust, illetve viselkedést örökléssel is támogathatunk, ha egy környezetosztályból közvetlenül származtatunk különböző viselkedéseket megvalósító alosztályokat. Ez azonban „bedrótozza" a viselkedést a Környezetbe, az algoritmus megvalósításának keverése a környezetével pedig nehezíti a Környezet kódjának megértését, karbantartását, illetve bővítését. Emellett az algoritmus így dinamikusan nem változtatható, ráadásul számos rokon osztály jön létre, amelyeket csak az általuk alkalmazott algoritmus vagy viselkedés különböztet meg. Ha az algoritmust önálló Stratégia osztályokba zárjuk, a környezettel függetlenül cserélgethetjük azokat, ami megkönnyíti a módosítást és a bővítést.
3. *A stratégiák szükségtelenné teszik a feltételes utasításokat.* A Stratégia mintával kiválthatjuk a kívánt viselkedés kiválasztására szolgáló feltételes utasításokat. Ha különböző viselkedéseket egyetlen osztályba tuszkolunk, nehezen kerülhetjük el a megfelelő kiválasztását célzó feltételes utasítások használatát. A viselkedés önálló Stratégia osztályokba zárása ezt is szükségtelenné teszi.
   Stratégiák nélkül például a szöveget sorokra tördelő kód valahogy így nézne ki:

```
void Composition::Repair() {
        switch (_breakingStrategy) {
        case SimpleStrategy:
                ComposeWithSimpleCompositor();
                break;
        case TeXStrategy:
                ComposeWithTeXCompositor();
                break;
        //...
        }
        // az eredmények összeolvasztása a meglevő
        // összetétellel, ha szükséges
}
```

   A Stratégia mintában a `case` utasításra nincs szükség, mert a sortörés feladatát egy Stratégia objektumra ruházzuk át:

```
void Composition::Repair () {
        _compositor->Compose();
        // az eredmények összeolvasztása a meglevő
        // összetétellel, ha szükséges
}
```

Az olyan kód jelenléte, amelyben számos feltételes utasítás található, gyakran éppen arra utal, hogy érdemes lenne a Stratégia tervezési mintát alkalmazni.

4. *A megvalósítások választéka.* A stratégiák *ugyanannak* a viselkedésnek különböző megvalósításait nyújtják, amelyek közül az ügyfél aszerint választhat, hogy idővel vagy tárhellyel kíván-e inkább fizetni.
5. *Az ügyfeleknek tudniuk kell a különböző stratégiákról.* A minta egy lehetséges hátulütője, hogy az ügyfélnek tudnia kell, miben különböznek az egyes stratégiák, mielőtt kiválaszthatná a megfelelőt. Így a megvalósítás részleteinek felfedésére kerülhet sor, ezért a mintát csak akkor használjuk, ha az ügyfelek számára a viselkedések megkülönböztetése létfontosságú.
6. *A Stratégia és Környezet objektumok közötti kommunikációs többlet.* A KonkrétStratégia osztályok mind használják a megosztott Strategy felületet, függetlenül attól, hogy az általuk megvalósított algoritmus egyszerű vagy összetett-e. Így valószínű, hogy egyes KonkrétStratégia objektumok az e felületen keresztül kapott információk egy részét nem használják fel, sőt, a legegyszerűbbek talán semmit sem hasznosítanak belőle. Ez azt jelenti, hogy a környezet időnként olyan paramétereket hoz létre és lát el kezdőértékkel, amelyek használatára soha nem kerül sor. Ha ilyen probléma merül fel, szorosabb csatolásra van szükség a Stratégia és a Környezet között.
7. *Az objektumok nagy száma.* A stratégiák növelik az alkalmazás objektumainak számát. Ez a többletteher néha csökkenthető, ha a stratégiákat állapot nélküli objektumokként valósítjuk meg, amelyeket a környezetek megoszthatnak. Az állapotot ekkor a környezet tartja számon, és a Stratégia objektumokhoz irányuló kérelmekben adja át. A megosztott stratégiák nem szabad, hogy a meghívások között állapotot tároljanak. A Pehelysúlyú tervezési minta ismertetésénél (4. fejezet) részletesebben tárgyaltuk e megközelítést.

Megvalósítás

A megvalósítás során a következőkre kell ügyelnünk:

1. *A Stratégia és Környezet felületek meghatározása.* A Stratégia és Környezet felületek hatékony hozzáférést kell biztosítsanak a KonkrétStratégia objektumoknak bármely adathoz, amelyre azoknak szükségük van a környezettől, és viszont.
2. *Stratégiák mint sablonparaméterek.* A C++-ban a stratégiával rendelkező osztályok sablonokkal (template) állíthatók be. Ez a megoldás csak akkor kivitelezhető, ha (1) a stratégia fordításkor kiválasztható, és (2) futásidőben nem kell megváltoztatni. Ha ezek a feltételek fennállnak, a beállítandó osztályt (például a `Context`-et) sablonosztályként határozzuk meg, amelynek paramétere egy `Strategy` osztály:

```
template <class AStrategy>
class Context {
        void Operation() { theStrategy.DoAlgorithm(); }
        //...
private:
        AStrategy theStrategy;
};
```

Az osztályt ezután példányosításkor egy `Strategy` osztállyal állítjuk be:

```
class MyStrategy {
public:
        void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

Sablonok használata esetén nem kell elvont osztályt meghatároznunk, ami a `Strategy` felületét írja le, emellett a `Strategy` sablonparaméterként való alkalmazása azt is lehetővé teszi, hogy egy stratégiát statikusan kössünk a környezetéhez, ami növelheti a hatékonyságot.

3. *A Stratégia objektumok választhatóvá tétele.* A Környezet osztály egyszerűsíthető, amennyiben van értelme annak, hogy *ne* legyen Stratégia objektuma. Ekkor a Környezet hozzáférés előtt ellenőrzi, hogy rendelkezik-e Stratégia objektummal; ha igen, a környezet a szokásos módon használatba veszi, ha nem, az alapértelmezett viselkedést követi. A megközelítés előnye, hogy az ügyfeleknek egyáltalán nem kell törődniük a Strategy objektumokkal, *kivéve* ha nem tetszik nekik az alapértelmezett viselkedés.

Példakód

Itt a Feladat részben bemutatott példa magasszintű kódját adjuk meg, ami az InterViews [LCI+92] Composition (Összetétel) és Compositor (Összeállító) osztályainak megvalósításán alapul.

A `Composition` osztály `Component` (Elem) példányok gyűjteményét tartalmazza, amelyek a dokumentum szöveg- és grafikus elemeit jelölik. Az összetétel az elemobjektumokat egy, a sortörő stratégiát egységbe záró `Compositor` alosztály példányának segítségével sorokba rendezi. Minden elemhez tartozik egy természetes vagy alapméret (natural size), egy nyújthatósági (stretchability) és egy zsugoríthatósági (shrinkability) érték. A nyújthatósági érték azt adja meg, hogy az összetevő az alapméretéhez képest mennyire nőhet meg, a zsugoríthatósági érték pedig azt, hogy mennyit zsugorodhat. Az összetétel átadja ezeket az értékeket egy összeállítónak, amely segítségükkel megállapítja a sortörések legkedvezőbb helyét.

```
class Composition {
public:
   Composition(Compositor*);
   void Repair();
private:
   Compositor* _compositor;
   Component* _components;     // az elemek listája
   int _componentCount;        // az elemek száma
   int _lineWidth;             // az összetétel sorszélessége
   int _lineBreaks;            // a sortörések helye az
                               // elemekben
   int _lineCount;             // a sorok száma
};
```

Amikor csak új elrendezésre van szükség, az összetétel megkéri összeállítóját, hogy állapítsa meg a sortörések helyét. Három tömböt ad át neki, amelyek az elemek alapméreteit, illetve nyújthatósági és zsugoríthatósági értékeit tartalmazzák. Emellett átadja az elemek számát is, a sor szélességét, illetve még egy tömböt, amelyet az összeállító a sortörések helyével tölt majd fel. Az összeállító a sortörések kiszámított számát adja vissza.

A `Compositor` felület lehetővé teszi az összetételnek, hogy minden szükséges adatot átadjon az összeállítónak:

```
class Compositor {
public:
   virtual int Compose(
         Coord natural[], Coord stretch[], Coord shrink[],
         int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
   Compositor();
};
```

Észrevehetjük, hogy a `Compositor` elvont osztály, amelynek konkrét alosztályai határozzák meg a különböző sortörési stratégiákat.

Az összetétel `Repair` (Javít) műveletében hívja meg összeállítóját. A `Repair` először feltölti a tömböket az elemek alapméret, nyújthatóság és zsugoríthatóság értékeivel (aminek részleteit a rövidség kedvéért itt kihagyjuk), majd az összeállítótól elkéri a sortöréseket (a töréseket is mellőztük), végül azoknak megfelelően elrendezi az elemeket:

```
void Composition::Repair () {
   Coord* natural;
   Coord* stretchability;
   Coord* shrinkability;
   int componentCount;
   int* breaks;

   // a tömbök előkészítése a kívánt méretekkel
   //...

   // a törések helyének megállapítása
   int breakCount;
   breakCount = _compositor->Compose(
         natural, stretchability, shrinkability,
         componentCount, _lineWidth, breaks
   );

   // az elemek elhelyezése a töréseknek megfelelően
   //...
}
```

Most vessünk egy pillantást a `Compositor` alosztályaira. A `SimpleCompositor` (EgyszerűÖsszeállító) soronként megvizsgálja az elemeket, hogy megállapítsa, hová kerüljenek a törések:

```
class SimpleCompositor : public Compositor {
public:
   SimpleCompositor();

   virtual int Compose(
         Coord natural[], Coord stretch[], Coord shrink[],
         int componentCount, int lineWidth, int breaks[]
    );
    //...
};
```

A `TeXCompositor` (TeXÖsszeállító) stratégiája általánosabb. Egyszerre egy *bekezdést* vizsgál, figyelembe véve az elemek méretét és nyújthatóságát, emellett az elemek közötti üreshelyek lehető legkevesebbre csökkentésével egyenletes „színt" is próbál adni a bekezdésnek.

```
class TeXCompositor : public Compositor {
public:
   TeXCompositor();

   virtual int Compose(
         Coord natural[], Coord stretch[], Coord shrink[],
         int componentCount, int lineWidth, int breaks[]
    );
    //...
};
```

Az `ArrayCompositor` (TömbÖsszeállító) az elemeket szabályos közönként sorokra tördeli.

```
class ArrayCompositor : public Compositor {
public:
   ArrayCompositor(int interval);

   virtual int Compose(
         Coord natural[], Coord stretch[], Coord shrink[],
         int componentCount, int lineWidth, int breaks[]
    );
    //...
};
```

A fenti osztályok nem használnak fel minden információt, amit a `Compose`-ban (Összeállít) kaptak. A `SimpleCompositor` figyelmen kívül hagyja az összetevők nyújthatóságát, csak alapszélességükkel számol. A `TeXCompositor` minden átadott információt hasznosít, az `ArrayCompositor` viszont mindent figyelmen kívül hagy.

A `Composition` úgy példányosítható, hogy átadjuk neki a használni kívánt összeállítót:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

A `Compositor` felületét gondosan kell megtervezni, hogy minden elrendezési algoritmust támogasson, amit csak az alosztályok megvalósíthatnak. Ez azért fontos, mert nyilván nem szeretnénk minden új alosztály létrehozásakor megváltoztatni a felületet, ami a meglevő alosztályok megváltoztatását is maga után vonná. Általánosságban azt mondhatjuk, a Stratégia és Környezet felületek határozzák meg, hogy a tervezési minta mennyire éri el a célját.

Ismert felhasználások

Mind az ET++ [WGM88], mind az InterViews stratégiákat használ a fent leírt különböző sortörési algoritmusok egységbe zárására.

A fordítói kódoptimalizáló RTL rendszerben [JML92] stratégiák határozzák meg a különböző regiszterfoglalási sémákat (RegisterAllocator), illetve utasításkészlet-ütemező irányelveket (RISCscheduler, CISCscheduler). Ezzel az optimalizáló rugalmasabban alkalmazkodhat a különféle számítógép-architektúrákhoz.

Az ET++ SwapsManager számítómotor-keretrendszer különböző pénzügyi eszközök árait számítja ki [EG92]. Kulcsfogalmai az Instrument (Eszköz) és a YieldCurve (Hozamgörbe). Az egyes eszközöket az Instrument alosztályaiként valósították meg. A YieldCurve azokat a tényezőket számítja ki, amelyek a jövőbeni pénzforgalom jelenlegi értékét határozzák meg. Mindkét osztály Stratégia osztályokra ruház át bizonyos feladatokat. A keretrendszer KonkrétStratégia osztályok családjával számítja ki a pénzáramlást, a devizaárfolyamokat, és a leszámítolási tényezőket. Új számítómotorokat úgy készíthetünk, ha az Instrument és YieldCurve osztályokat más KonkrétStratégia objektumokkal állítjuk be. Ez a megközelítés támogatja a meglevő Stratégia-megvalósítások keverését és illesztését, valamint újak meghatározását is.

A Booch komponensek [BV90] a stratégiákat sablonargumentumként használják. A Booch gyűjteményosztályok háromféle memóriafoglalási stratégiát támogatnak: kezelt (foglalás gyűjtőtárból), ellenőrzött (a foglalást és felszabadítást zárak védik), illetve kezeletlen (szokványos memóriafoglalás). A stratégiát példányosításkor sablonargumentumként adjuk át

a gyűjteményosztálynak, a kezeletlen stratégiát alkalmazó UnboundedCollection (NemkorlátosGyűjtemény) példányosítása például az `UnboundedCollection<MyItemType*, Unmanaged>` formában történik.

A RApp rendszer integrált áramkörök elrendezését szolgálja [GA89, AG90]. A RApp-nak kell lefektetnie és összekötnie az áramkör alrendszereit összekapcsoló vezetékeket. A kapcsoló algoritmusok az elvont Router osztály alosztályai, amely maga egy stratégiaosztály.

A Borland ObjectWindows-a [Bor94] a párbeszédablakokban használ stratégiákat, annak ellenőrzésére, hogy a felhasználó érvényes adatokat adott-e meg. A számoknak például egy meghatározott tartományban kell lenniük, a számbeviteli mezők pedig csak számjegyeket fogadhatnak el. A karakterláncok érvényességének ellenőrzése táblázatos keresést igényelhet.

Az ObjectWindows Validator (Érvényesítő) objektumokkal zárja egységbe az érvényesítő stratégiákat. Az érvényesítők a stratégiaobjektumoknak felelnek meg. Az adatbeviteli mezők az érvényesítő stratégiával egy választható Validator objektumot bíznak meg. Az ügyfél igény esetén érvényesítőket kapcsol a mezőkhöz (a választható stratégia példája); amikor pedig a párbeszédablakot bezárják, a beviteli mezők felkérik érvényesítőiket az adatok ellenőrzésére. A leggyakrabban előforduló ellenőrzésekhez az osztálykönyvtár biztosítja az érvényesítőket, a számok esetében például a RangeValidator-t (Tartományellenőrző). Új, ügyfélfüggő érvényesítő stratégiákat is könnyen meghatározhatunk; ehhez elég a Validator osztályból új alosztályokat származtatnunk.

Kapcsolódó minták

Pehelysúlyú: a stratégiaobjektumok általában jól illeszkednek a Pehelysúlyú mintába.

Sablonfüggvény

Viselkedési osztályminta

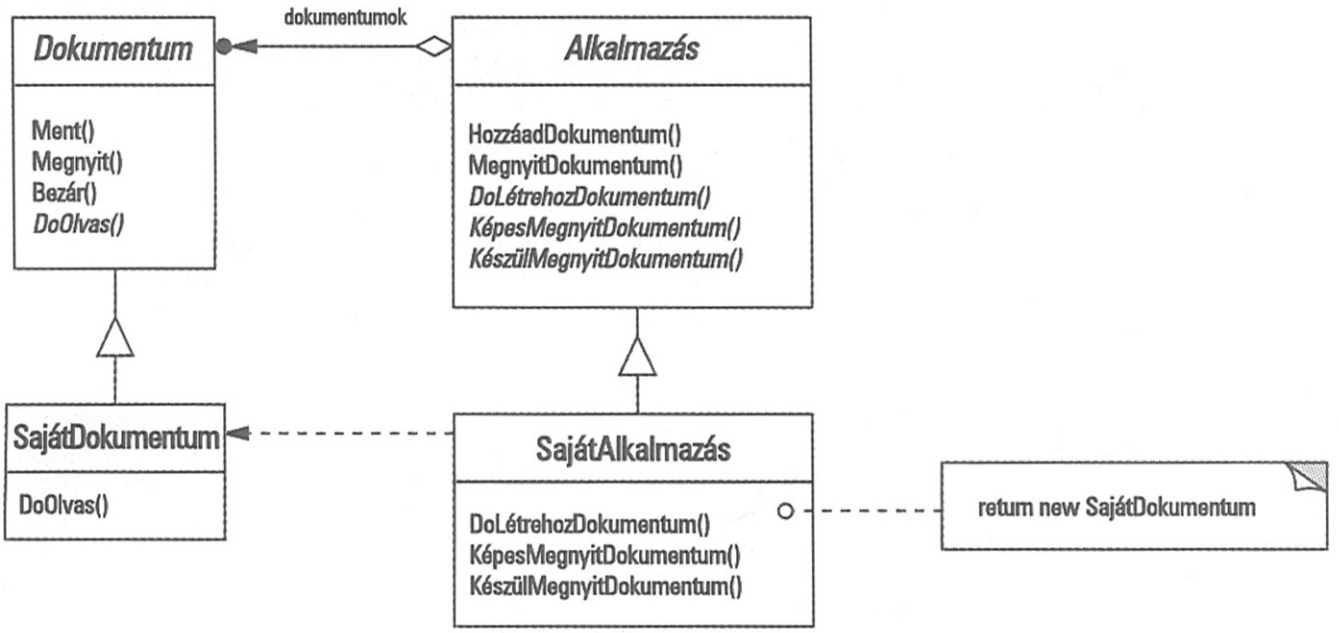
Cél

Egy adott művelet algoritmusának vázát elkészíteni, amelynek egyes lépéseit alosztályokra ruházzuk át. Így az alosztályok az algoritmus egyes lépéseit felülbírálhatják, anélkül, hogy az algoritmus szerkezete módosulna.

Feladat

Vegyünk egy alkalmazás-keretrendszert, amelyben találunk egy Alkalmazás (Application) és egy Dokumentum (Document) nevű osztályt. Az Alkalmazás osztály felel a „külső" formátumban – például fájlban – tárolt dokumentumok megnyitásáért, míg a Dokumentum objektum a dokumentum adatait ábrázolja, miután kiolvastuk azokat a fájlból.

A keretrendszer segítségével épített alkalmazások egyedi igényeiknek megfelelően alosztályokat származtathatnak az Alkalmazás és Dokumentum osztályokból. Egy rajzolóprogram például meghatározhat egy RajzAlkalmazás (DrawApplication) és egy RajzDokumentum (DrawDocument) nevű alosztályt, egy táblázatkezelő egy TáblázatKezelőAlkalmazás (SpreadSheetApplication) és egy TáblázatKezelőDokumentum (SpreadSheetDocument) nevűt, és így tovább.



Az elvont Alkalmazás osztály MegnyitDokumentum (OpenDocument) művelete határozza meg a dokumentumok megnyitására és olvasására szolgáló algoritmust:

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
            // a dokumentum nem nyitható meg
            return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
            _docs->AddDocument(doc);
            AboutToOpenDocument(doc);
            doc->Open();
            doc->DoRead();
    }
}
```

Az `OpenDocument` a dokumentum megnyitásának minden lépését meghatározza. Ellenőrzi, hogy a dokumentum megnyitható-e, létrehozza az alkalmazásra jellemző Dokumentum objektumot, hozzáadja a dokumentumok halmazához, és kiolvassa a dokumentumot (Document) a fájlból.

Az ilyen műveleteket **sablonfüggvényeknek** hívjuk. A sablonfüggvény elvont műveletekkel határoz meg egy algoritmust, mely műveleteket az alosztályok felülbírálják, hogy elérjék a megfelelő viselkedést. Az Alkalmazás alosztályai az algoritmusnak a dokumentum megnyithatóságát ellenőrző (KépesMegnyitDokumentum, CanOpenDocument), illetve a Dokumentum objektumot létrehozó (DoLétrehozDokumentum, DoCreateDocument) lépéseit határozzák meg, a Dokumentum osztályok pedig a dokumentum olvasására szolgálót (DoOlvas, DoRead). A sablonfüggvény egy olyan műveletet is meghatároz, amely értesíti az Alkalmazás alosztályokat arról, hogy a dokumentum megnyitására készülünk (KészülMegnyitDokumentum, AboutToOpenDocument), ha erre szükségük lenne.

Azzal, hogy az algoritmus egyes lépéseit elvont műveletekkel határozza meg, a sablonfüggvény rögzíti azok sorrendjét, de megengedi az Alkalmazás és Dokumentum alosztályoknak, hogy a lépéseket egyedi igényeikhez igazítsák.

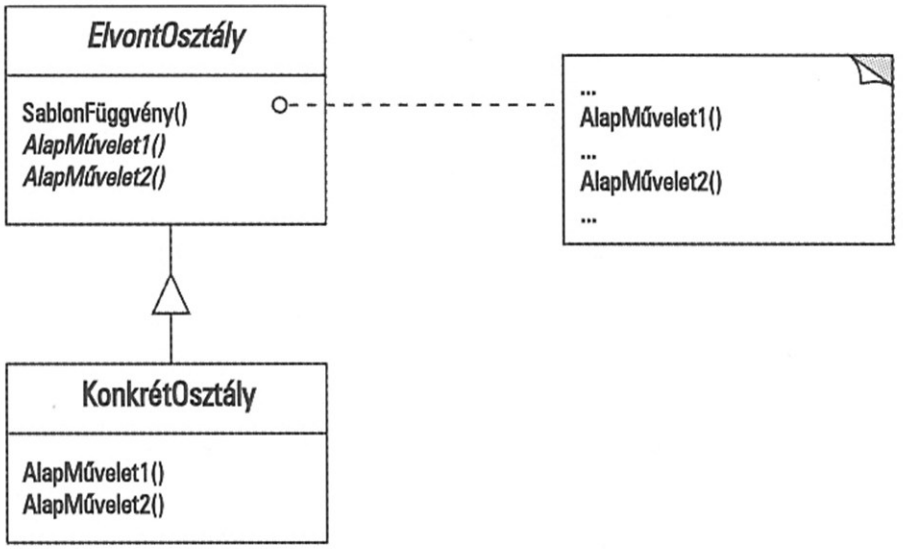
Alkalmazhatóság

A Sablonfüggvény tervezési minta alkalmazása az alábbi esetekben célszerű:

- Egy algoritmus nem változó részeit egyszerre szeretnénk megvalósítani, míg a változó viselkedés megvalósítását az alosztályokra hagynánk.
- Az alosztályok közös viselkedését a kódkettőződés elkerülése végett közös osztályban kell rögzíteni. („Refactoring to generalize", vagyis „újraépítés az általánosítás érdekében", lásd Opdyke és Johnson [OJ93].) Ehhez először meg kell keresnünk az eltéréseket a meglevő kódban, ezeket új műveletekbe kell helyeznünk, végül az eltérő kódot sablonfüggvénnyel kell helyettesítünk, ami az új műveletek valamelyikét hívja meg.

- Kézben szeretnénk tartani az alosztályok bővítését. Ha olyan sablonfüggvényt határozunk meg, amely adott pontokon „horog" (hook) műveleteket hív meg (lásd a Következmények részt), a bővítéseket ezekre a pontokra korlátozhatjuk.

Szerkezet



Résztvevők

- ElvontOsztály (Alkalmazás)
  - Elvont **alapműveleteket** határoz meg, amelyeket a konkrét alosztályok felülírnak, hogy megvalósítsák az algoritmus lépéseit.
  - Sablonfüggvényt biztosít, amely meghatározza az algoritmus vázát. A sablonfüggvény alapműveleteket és az ElvontOsztályban (AbstractClass) vagy más objektumokban meghatározott műveleteket is meghív.
- KonkrétOsztály (SajátAlkalmazás)
  - Megvalósítja az alapműveleteket, hogy végrehajthassa az algoritmus alosztályfüggő lépéseit.

Együttműködés

- A KonkrétOsztály (ConcreteClass) az ElvontOsztályra támaszkodik, amely az algoritmus nem változó lépéseit valósítja meg.

Következmények

A sablonfüggvények alapvető fontosságúak a kód-újrahasznosításban. Különösen fontos szerepet töltenek be az osztálykönyvtárakban, hiszen a könyvtár osztályainak közös viselkedését hivatottak leírni.

A sablonfüggvények alkalmazása fordított vezérlési szerkezetet erdményez, amire időnként a „Hollywood elv" néven hivatkoznak („Ne hívjon minket – mi hívjuk magát.") [Swe85]. Ez arra utal, hogy a szülő osztály hívja az alosztály műveleteit, és nem fordítva.

A sablonfüggvények az alábbi típusú műveleteket hívják meg:

- konkrét műveletek (a KonkrétOsztályokon vagy az ügyfélosztályokon);
- konkrét ElvontOsztály műveletek (azon műveletek, amelyek általánosságban hasznosak az alosztályok számára);
- alapműveletek (vagyis az elvont műveletek);
- gyártófüggvények (lásd a Gyártófüggvény mintát a 3. fejezetben); és
- **horogműveletek** (ezek biztosítják az alapértelmezett viselkedést, amelyet az alosztályok igényeiknek megfelelően kibővíthetnek; a horogművelet alapértelmezés szerint gyakran semmit nem csinál).

Lényeges, hogy a sablonfüggvények meghatározzák, mely műveletek horgok (ezek *esetleg* felülbírálhatók), és melyek elvont műveletek (ezeket felül *kell* bírálni). Ahhoz, hogy egy alosztály készítője hatékonyan használhasson fel egy elvont osztályt, tudnia kell, mely műveleteket kell felülírnia.

Az alosztályok úgy *bővíthetik* ki a szülő egy műveletének viselkedését, hogy felülírják a műveletet, és kifejezetten meghívják a szülőműveletet:

```
void DerivedClass::Operation () {
   ParentClass::Operation();
   // a DerivedClass bővített művelete
}
```

Sajnos az örökölt művelet meghívásáról könnyű megfeledkezni, de egy ilyen műveletet sablonfüggvénnyé is alakíthatunk, hogy a szülő felügyelhesse, hogyan bővítik ki alosztályai. A megoldás lényege, hogy a szülő osztály sablonfüggvényéből meghívunk egy horogműveletet, amelyet aztán az alosztályok felülbírálhatnak:

```
void ParentClass::Operation () {
   // a ParentClass viselkedése
   HookOperation();
}
```

A `HookOperation` (HorogMűvelet) semmit nem csinál a szülő osztályban:

```
void ParentClass::HookOperation () { }
```

Az alosztályok viszont a `HookOperation`-t felülírva kibővítik annak viselkedését:

```
void DerivedClass::HookOperation () {
   // a származtatott osztály bővítése
}
```

Megvalósítás

Három dolgot fontos megemlítenünk:

1. *A C++ hozzáférés-vezérlésének használata.* A C++-ban a sablonfüggvények által meghívott alapműveleteket védett tagokként is bevezethetjük, így biztosíthatjuk, hogy csak a sablonfügggvény hívhassa meg azokat. A *kötelezően* felülírandó alapműveleteket tisztán virtuálisként adjuk meg. Magát a sablonfüggvényt nem szabad felülírni, így az nem virtuális tagfüggvény lesz.
2. *Az alapműveletek számának a lehető legkisebbre csökkentése.* A sablonfüggvények írásának egyik fontos célja az alosztályok által az algoritmus megvalósításához kötelezően felülírandó alapműveletek számának csökkentése. Minél több műveletet kell felülbírálni, annál több az ügyfelek munkája.
3. *Az elnevezési rendszer.* A felülírandó műveleteket könnyen azonosíthatjuk, ha nevük elé valamilyen közös előtagot teszünk. A Macintosh alkalmazások MacApp keretrendszere [App89] például a sablonfüggvények neve elé a „Do" előtagot helyezi (DoCreateDocument, DoRead stb.).

Példakód

Az alábbi C++ példa azt mutatja, hogyan kényszeríthet egy szülő osztály egy invariánst (állandó állítást) alosztályaira. A példa forrása a NeXT AppKit [Add94]. Vegyünk egy `View` (Nézet) nevű osztályt, amelynek segítségével a képernyőre rajzolhatunk. A `View` ragaszkodik ahhoz, hogy alosztályai csak azután kezdhessék meg a rajzolást, hogy az ablak (view) megkapta a fókuszt. Ez bizonyos megjelenési tulajdonságok (színek, betűtípusok) megfelelő állapotbeállítását vonja maga után.

A beállításokat a `Display` (Megjelenít) sablonfüggvényre bízzuk. A `View` két konkrét műveletet határoz meg: a `SetFocus` (BeállítFókusz) beállítja, a `ResetFocus` (VisszaállítFókusz) pedig törli a rajzolási állapotot. A tényleges rajzolást a `View` `DoDisplay` horogművelete végzi. A `Display` a `DoDisplay` előtt meghívja a `SetFocus`-t, hogy beállítsa az állapotot, a végén pedig meghívja a `ResetFocust`-t, hogy visszaálljon az eredeti helyzet.

```
void View::Display () {
   SetFocus();
   DoDisplay();
   ResetFocus();
}
```

Az invariáns fenntartásához a `View` ügyfelei mindig a `Display`-t hívják, alosztályai pedig a `DoDisplay`-t írják felül.

A `DoDisplay` a `View`-ban semmit nem csinál:

```
void View::DoDisplay () { }
```

Az alosztályok e művelet felülírásával határozzák meg egyedi rajzoló műveletüket:

```
void MyView::DoDisplay () {
   // az ablak tartalmának megjelenítése
}
```

Ismert felhasználások

A sablonfüggvények alapvető jelentőségét mutatja, hogy szinte minden elvont osztályban megtalálhatók. Wirfs-Brock és szerzőtársai [WBWW90, WBJ90] remek áttekintést nyújtanak a sablonfüggvényekről.

Kapcsolódó minták

A sablonfüggvények gyakran támaszkodnak a Gyártófüggvény mintára. A Feladat részben szereplő DoCreateDocument, amelyet az OpenDocument sablonfüggvény hív meg, is gyártófüggvény.

A sablonfüggvények öröklés révén változtatják az algoritmus egyes részeit, a Stratégia mintában átruházás révén a teljes algoritmust kicserélhetjük.

Látogató

Viselkedési objektumminta

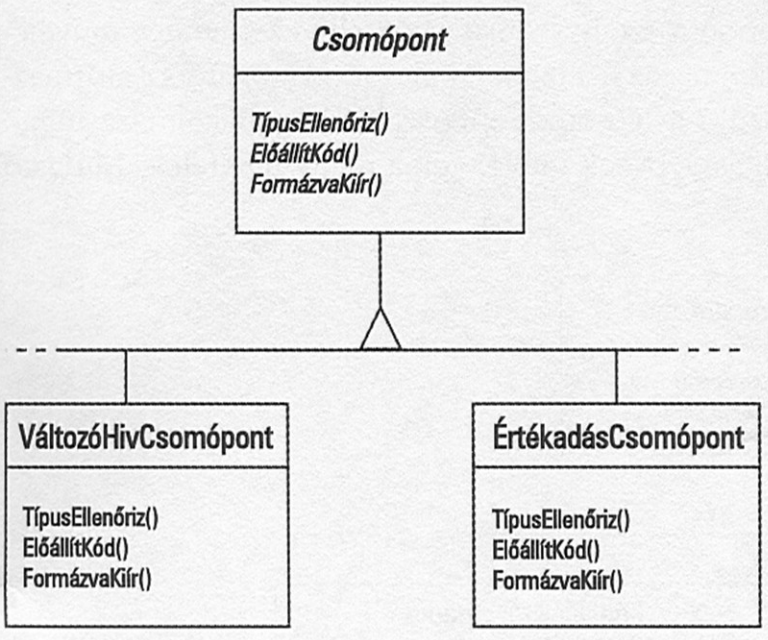
Cél

Egy objektumszerkezet elemein végrehajtandó műveletet ábrázolni: a Látogató minta segítségével anélkül határozhatunk meg egy új műveletet, hogy a benne részt vevő elemek osztályát meg kellene változtatnunk.

Feladat

Vegyünk egy fordítóprogramot, ami a programokat elvont szintaxisfákként ábrázolja. Ezeken „statikus jelentéselemző" műveleteket kell végeznie, például ellenőriznie kell, hogy minden változót meghatároztunk-e, emellett pedig kódot is elő kell állítania. Ennek megfelelően szükségünk lesz típusellenőrző, kódoptimalizáló, vezérléselemző műveletekre, olyanra, ami ellenőrzi, hogy használatba vétel előtt adtunk-e értéket a változóknak, és így tovább. Az elvont szintaxisfát ezenkívül használhatnánk formázott kiíratásra, program-újraszervezésre, kód-előállításra és különféle, a programmal kapcsolatos számításokra is.

Az említett műveletek legtöbbjének másképpen kell kezelnie azokat a csomópontokat, amelyek értékadó utasításokat jelölnek, mint azokat, amelyek változókat vagy matematikai kifejezéseket ábrázolnak. Ezért külön osztályunk lesz az értékadásokhoz, egy másik a változók eléréséhez, egy harmadik a matematikai kifejezésekhez, és így tovább. A csomópont-osztályok halmaza természetesen az alkalmazott programozási nyelvtől függ, de nem különbözhet jelentősen.



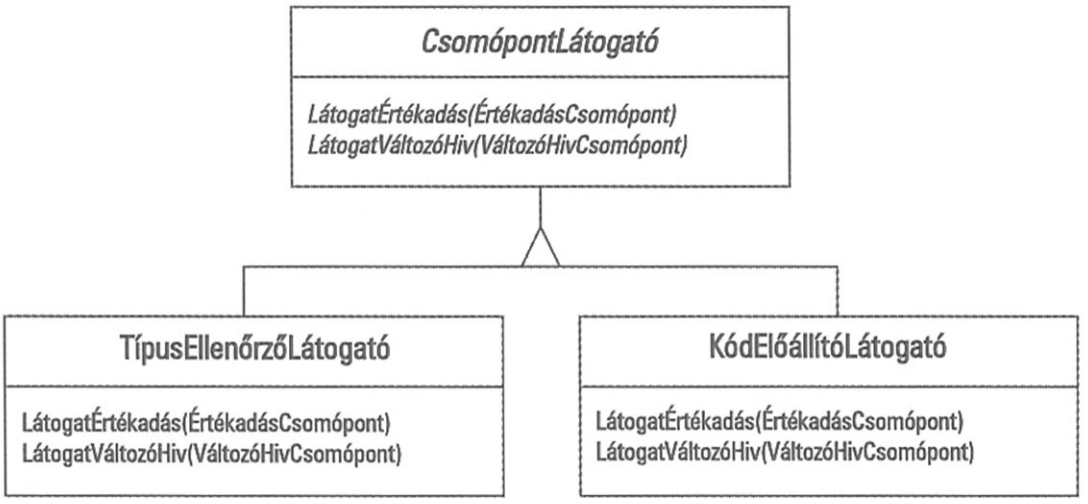
A fenti diagram a Csomópont (Node) osztályhierarchia egy részét mutatja. A gond itt az, hogy az említett műveletek különféle csomópont-osztályokba való elosztása révén egy

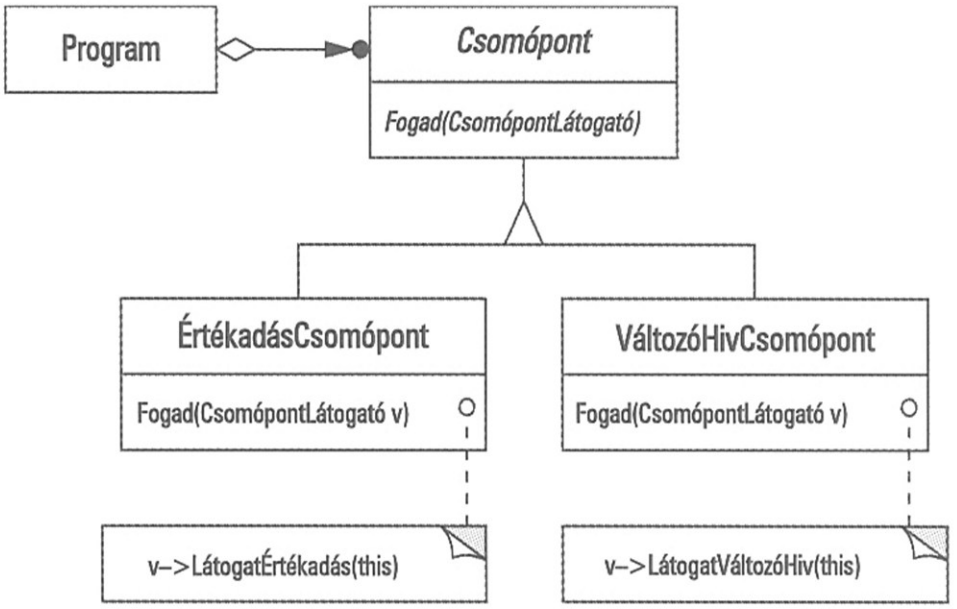
olyan rendszer áll elő, amely nehezen átlátható, nehezen karbantartható, és módosítása is fáradságos. Zavaró, hogy a típusellenőrző kód keveredik a formázó vagy a vezérléselemző kóddal, arról nem is beszélve, hogy egy új művelet hozzáadása valószínűleg az összes osztály újrafordítását igényli. Jobb lenne, ha az új műveleteket külön vehetnénk fel, a csomópont-osztályok pedig függetlenek lennének az őket használó műveletektől.

Mindkettő elérhető, ha az osztályokban található rokon műveleteket önálló objektumba, úgynevezett **látogatóba** csomagoljuk, majd ezt adjuk át az elvont szintaxisfa elemeinek bejárás közben. Amikor egy elem „fogadja" (accept) a látogatót, kérelmet küld annak, ami az adott elem osztályát tartalmazza, illetve magát az elemet mint argumentumot. A látogató ezután végrehajtja az elemnek megfelelő műveletet, ami korábban az elem osztályában szerepelt.

Például egy fordítóprogram, amely nem használ látogatókat, a TípusEllenőriz (TypeCheck) műveletet az elvont szintaxisfára meghívva végezhet típusellenőrzést egy eljáráson. Ekkor minden csomópont megvalósítja a TípusEllenőriz műveletet, azzal, hogy meghívja azt a csomópont elemeire (lásd az előző osztálydiagramot). Ha a fordító a típusellenőrzést látogatók segítségével végzi, létrehoz egy TípusEllenőrzőLátogató (TypeCheckingVisitor) objektumot, és azt argumentumként használva a Fogad (Accept) műveletet hívja meg az elvont szintaxisfára. Ennél a megoldásnál a csomópontok a látogató visszahívásával a Fogad műveletet valósítják meg: az értékadó (assignment) csomópontok a LátogatÉrtékadás (VisitAssignment), a változóhivatkozások a LátogatVáltozóHivatkozás (VisitVariableReference) művelet meghívásával. A korábban az ÉrtékadásCsomópont (AssignmentNode) osztályban levő TípusEllenőriz művelet így a TípusEllenőrzőLátogatóra meghívott LátogatÉrtékadás művelet lesz.

Ahhoz, hogy a látogatókkal ne csupán típusellenőrzést végezhessünk, szükségünk lesz az elvont szintaxisfa összes látogatójának elvont szülő osztályára (CsomópontLátogató, NodeVisitor), amelynek minden csomópont osztály számára be kell vezetnie egy műveletet. Egy programméreteket kiszámító alkalmazás ezután a CsomópontLátogatóból származtathat új alosztályokat, nem lesz többé szükség a csomópont osztályokban alkalmazásfüggő kódra. A Látogató minta a programfordítási lépések műveleteit a nekik megfelelő Látogató objektumba zárja.





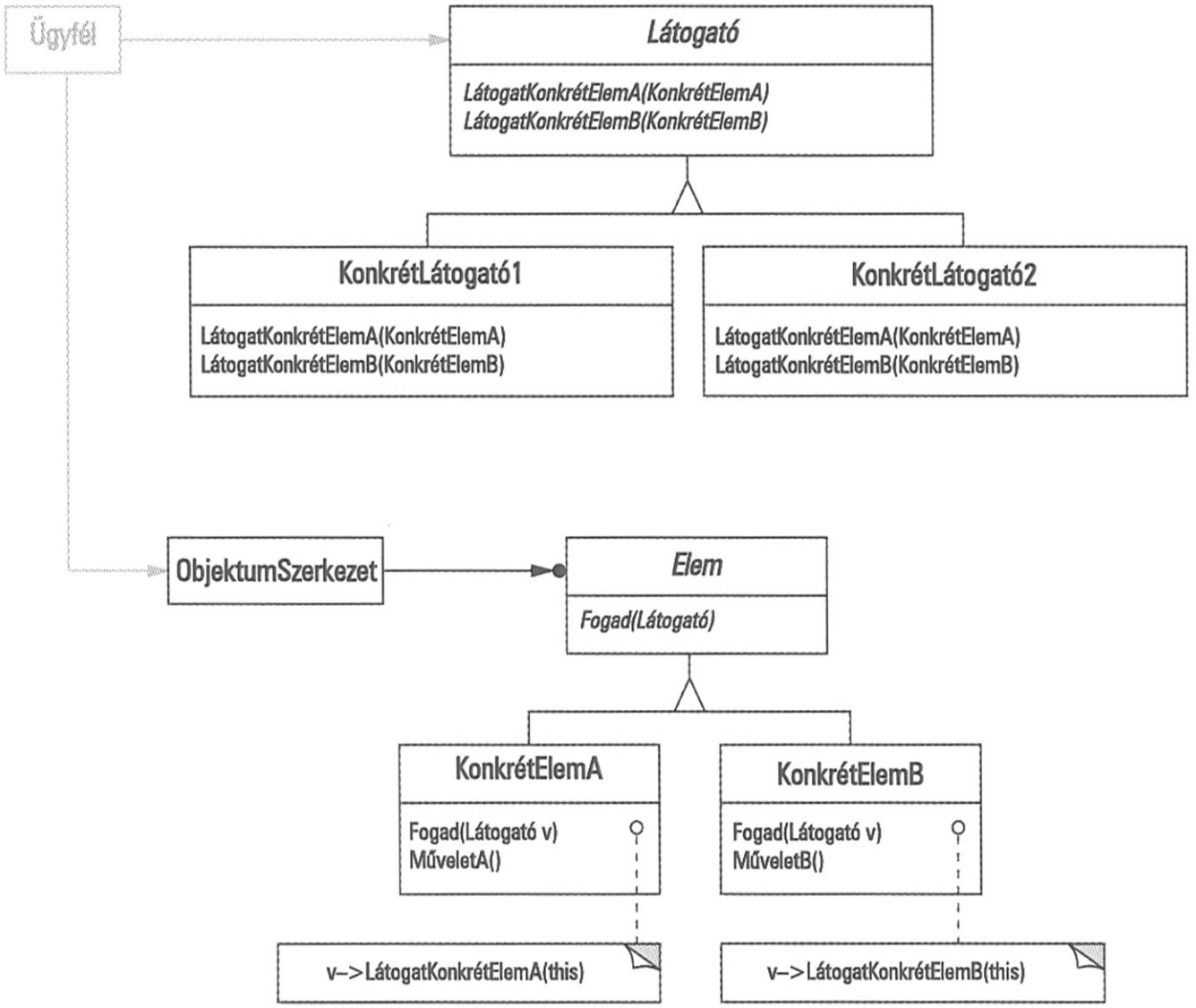
A Látogató mintában két osztályhierarchiát alakítunk ki: egyet az elemek számára, amelyeken műveleteket végzünk (Csomópont hierarchia), és egy másikat az elemeken végrehajtandó műveleteket meghatározó látogatók számára (CsomópontLátogató hierarchia). Új műveletet a látogató osztályhierarchia új alosztállyal való bővítésével határozhatunk meg. Amíg a fordító által elfogadott nyelvtan nem változik (vagyis amíg nem kell új Csomópont alosztályokat felvennünk), az új szolgáltatások beépítéséhez elég új CsomópontLátogató alosztályokat létrehoznunk.

Alkalmazhatóság

A Látogató tervezési minta alkalmazása a következő esetekben célszerű:

- Egy adott objektumszerkezet számos különböző felületű osztályt tartalmaz, és ezeken az objektumokon olyan műveleteket szeretnénk végezni, amelyek a konkrét osztályoktól függnek.
- Egy objektumszerkezet objektumain több önálló, egymással nem rokon műveletet kell végrehajtanunk, és nem akarjuk az osztályokat „beszennyezni" e műveletekkel. A Látogató minta segítségével a rokon műveletek együtt tarthatók, ha közös osztályban határozzuk meg azokat. Ha egy objektumszerkezetet több alkalmazás közösen használ, mindig érdemes a műveleteket a Látogató mintával az őket ténylegesen igénylő alkalmazásokba helyezni.
- Az adott objektumszerkezetet meghatározó osztályok ritkán változnak, de új műveletekre gyakran van szükség. Az említett osztályok megváltoztatása az összes látogató felületének módosítását igényli, ami valószínűleg költséges. Ha ezen osztályok gyakran változnak, érdemesebb a műveleteket beléjük helyezni.

Szerkezet



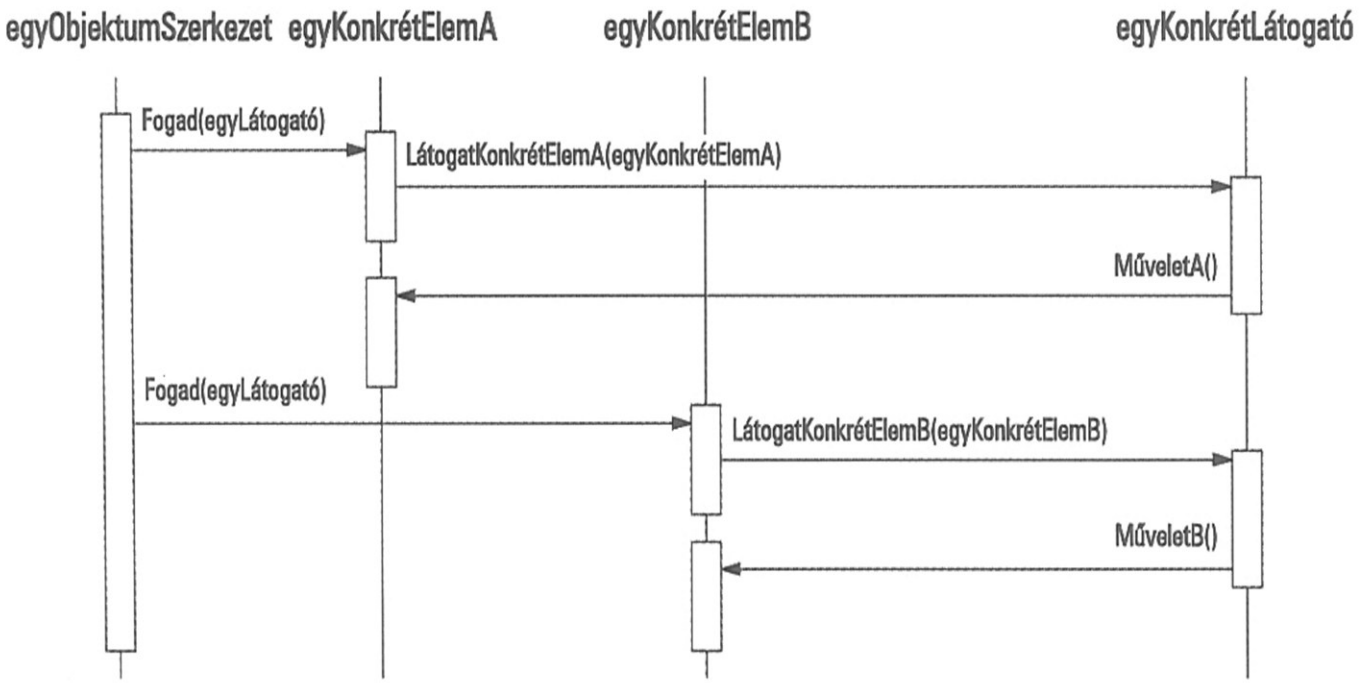
Résztvevők

- **Látogató** (CsomópontLátogató)
  - Az objektumszerkezet minden KonkrétElem (ConcreteElement) osztálya számára bevezet egy-egy Látogat (Visit) műveletet. A művelet neve és aláírása azonosítja a Látogat kérelmet küldő osztályt, így a látogató megállapíthatja a meglátogatott elem konkrét osztályát. Ezután a látogató az elemet közvetlenül a megfelelő felületen keresztül érheti el.
- **KonkrétLátogató** (TípusEllenőrzőLátogató)
  - Megvalósítja a Látogató (Visitor) által bevezetett műveleteket. Minden művelet az algoritmusnak a megfelelő osztályú objektum számára meghatározott részét valósítja meg. A KonkrétLátogató (ConcreteVisitor) biztosítja az algoritmus környezetét, és tárolja annak helyi állapotát. Gyakran ez az állapot gyűjti a bejárás során összegyűjtött eredményeket.

- **Elem** (Csomópont)
  - Meghatároz egy Fogad (Accept) műveletet, amelynek argumentuma egy látogató.
- **KonkrétElem** (ÉrtékadásCsomópont, VáltozóHivCsomópont)
  - Megvalósít egy Fogad műveletet, amelynek argumentuma egy látogató.
- **ObjektumSzerkezet** (Program)
  - Fel tudja sorolni az elemeit.
  - Magasszintű felületet biztosíthat, amelynek révén a látogató meglátogathatja az objektumszerkezet elemeit.
  - Lehet összetétel (lásd az Összetétel tervezési mintát a 4. fejezetben) vagy gyűjtemény, például lista vagy halmaz.

Együttműködés

- A Látogató mintát alkalmazó ügyfélnek létre kell hoznia egy KonkrétLátogató objektumot, majd a látogatóval minden elemet végiglátogatva be kell járnia az objektumszerkezetet.
- Amikor egy elemhez látogató érkezik, az elem meghívja az osztályának megfelelő Látogató műveletet, és átadja magát argumentumként, hogy a látogató megismerhesse állapotát, ha szükséges.
  Az alábbi együttműködési diagram egy objektumszerkezet, egy látogató, valamint két elem együttműködését mutatja:



Következmények

A Látogató minta előnyei és hátrányai többek között a következők:

1. *Megkönnyíti új műveletek hozzáadását.* A látogatók megkönnyítik az olyan műveletek hozzáadását, amelyek összetett objektumok elemeitől függnek. Az objektumszerkezet új művelettel való bővítéséhez így elég, ha új látogatót készítünk. Ha a szolgáltatásokat több osztályban szórnánk szét, új művelet felvételéhez az összes osztályt meg kellene változtatnunk.
2. *A látogató összegyűjti a rokon műveleteket, és elválasztja a kapcsolatban nem állókat.* A rokon szolgáltatások nem az objektumszerkezetet meghatározó osztályokban oszlanak el; egy látogató gyűjti össze azokat. A kapcsolatban nem álló műveletek saját látogató alosztályaikban kapnak helyet. Ez a rendszer egyszerűsíti mind az elemeket leíró osztályokat, mind a látogatókban meghatározott algoritmusokat. A látogatókban bármilyen algoritmusfüggő adatszerkezet elrejthető.
3. *Új KonkrétElem osztályt nehéz hozzáadni.* A Látogató minta megnehezíti az Elemből új alosztályok származtatását. Minden új KonkrétElem új elvont műveletet von maga után a Látogatóban, illetve ezek megvalósítását a KonkrétLátogató osztályokban. Egyes esetekben a Látogató osztály alapértelmezett megvalósítást nyújthat, amelyet a KonkrétLátogatók többsége örökölhet, de ez inkább kivételnek, mintsem szabálynak számít.
   A fentiek miatt a Látogató minta alkalmazásának kulcsfontosságú kérdése, hogy az objektumszerkezetben használt algoritmus módosítása valószínűbb-e, vagy a szerkezetet felépítő objektumosztályoké. Amennyiben gyakran veszünk fel új KonkrétElem osztályokat, a Látogató osztályhierarchia fenntartása nehézzé válhat. Ilyen esetben valószínűleg egyszerűbb, ha csak a szerkezetet alkotó osztályokban határozzuk meg a műveleteket. Ha viszont az Elem osztályhierarchia stabil, de a programot rendszeresen bővítjük új műveletekkel, vagy gyakran módosítjuk az algoritmusokat, a Látogató minta jó szolgálatot tehet a változások kezelésében.
4. *A látogatás átívelhet az osztályhierarchiákon.* A bejárók (lásd a Bejáró mintát a fejezet korábbi részében) is meglátogathatják egy szerkezet objektumait, miközben bejárják azokat műveleteik meghívásával, de különböző típusú elemekből álló objektumszerkezeteken nem ívelhetnek keresztül. A fejezetben korábban bemutatott Iterator (Bejáró) felület például csak az `Item` (Elem) típusú objektumokat képes elérni:

```
template<class Item>
class Iterator {
        //...
        Item CurrentItem() const;
};
```

   Ez a kód azt jelenti, hogy a bejáró kizárólag olyan elemeket látogathat meg, amelyek szülője az `Item` osztály. A Látogató mintában ilyen korlátozás nincs. Bármilyen szülővel rendelkező objektumok meglátogathatók, és a Látogató felületekhez bármilyen típusú objektum hozzáadható.

```
class Visitor {
public:
        //...
```

```
        void VisitMyType(MyType*);
        void VisitYourType(YourType*);
};
```

A fenti kódban például a `MyType` (EnyémTípus) és a `YourType` (TiédTípus) egyáltalán nem kell, hogy öröklés révén rokonságban álljanak.

5. *Az állapot tárolható.* A látogatók összegyűjthetik az állapotinformációkat, ahogy az objektumszerkezetben végiglátogatják az elemeket. Látogató nélkül az állapotot a bejárást végző műveletnek kiegészítő argumentumként adnánk át, esetleg erre a célra globális változókat használnánk.
6. *Megsértheti az egységbe zárást.* A Látogató minta által alkalmazott megközelítés feltételezi, hogy a KonkrétElem felület elég „erős" ahhoz, hogy a látogatók munkáját támogassa. Ebből következik, hogy a minta gyakran arra kényszerít, hogy olyan nyilvános műveleteket adjunk meg, amelyek hozzáférnek egy elem belső állapotához, ami megsértheti az egységbe zárás elvét.

Megvalósítás

Minden objektumszerkezethez tartozik egy kapcsolódó Látogató osztály. Ez az elvont osztály a szerkezetet felépítő valamennyi KonkrétElem osztály számára bevezet egy-egy LátogatKonkrétElem (VisitConcreteElement) műveletet. A Látogató minden Látogat (Visit) művelete argumentumként egy adott KonkrétElemet ad meg, így a látogató a konkrét elemek felületét közvetlenül elérheti. A KonkrétLátogató osztályok a Látogat műveletek felülbírálásával valósítják meg a nekik megfelelő KonkrétElem osztályok látogatófüggő viselkedését.

A Látogató (Visitor) osztályt a C++-ban a következőképpen vezethetjük be:

```
class Visitor {
public:
   virtual void VisitElementA(ElementA*);
   virtual void VisitElementB(ElementB*);

   // és így tovább a többi konkrét elem esetében is
protected:
   Visitor();
};
```

Minden KonkrétElem osztály megvalósít egy `Accept` (Fogad) műveletet, amely az osztályhoz tartozó látogató megfelelő `Visit...` (Látogat...) műveletét hívja meg. Így az, hogy melyik lesz a meghívott művelet, mind az elem, mind a látogató osztályától függ.[10]

[10] Ha függvénytúlterhelést alkalmaznánk, a műveleteknek ugyanazt az egyszerű nevet (pl. `Visit`) adhatnánk, mivel az átadott paraméter kellőképpen megkülönbözteti őket. E megoldás mellett és ellen egyaránt szólnak érvek. A túlterhelés egyrészt megerősíti, hogy a műveletek valójában ugyanazt az elemzést végzik, csak más-más argumentummal, másrészt viszont homályosabbá teheti a kódot olvasó számára, hogy mi is történik a hívás helyén. A döntést csak az befolyásolja, hogy általában jó megoldásnak tartjuk-e a függvénytúlterhelést, vagy sem.

A konkrét elemeket az alábbi módon vezetjük be:

```
class Element {
public:
   virtual ~Element();
   virtual void Accept(Visitor&) = 0;
protected:
   Element();
};

class ElementA : public Element {
public:
   ElementA();
   virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element {
public:
   ElementB();
   virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```

Egy `CompositeElement` (ÖsszetettElem) osztály az `Accept` műveletet a következőképpen valósítaná meg:

```
class CompositeElement : public Element {
public:
   virtual void Accept(Visitor&);
private:
   List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
   ListIterator<Element*> i(_children);

   for (i.First(); !i.IsDone(); i.Next()) {
         i.CurrentItem()->Accept(v);
}
v.VisitCompositeElement(this);
}
```

A Látogató minta alkalmazásakor két további megvalósítási kérdéssel kell foglalkoznunk:

1. *Kettős közvetítés.* A Látogató minta végeredményben azt teszi lehetővé, hogy az osztályokhoz azok megváltoztatása nélkül adhassunk új műveleteket. A minta ehhez az úgynevezett **kettős közvetítés** (double-dispatch) megoldást használja. Ez jól ismert eljárás, né-

hány programozási nyelv (például a CLOS) közvetlenül támogatja. A C++, a Smalltalk és más hasonló nyelvek az **egyszeres közvetítést** (single-dispatch) alkalmazzák.

Az egyszeres közvetítésű nyelvekben két tényező határozza meg, melyik művelet teljesít egy adott kérést: a kérelem neve és a fogadó típusa. Az például, hogy egy ElőállítKód (GenerateCode) kérelem melyik műveletet hívja meg, attól függ, milyen típusú a címzett csomópont objektum. A C++-ban a `GenerateCode` meghívása egy `VariableRefNode` (VáltozóHivCsomópont) példányra a `VariableRefNode::GenerateCode` hívást eredményezi (ami egy változóhivatkozás számára állít elő kódot), ha pedig egy `AssignmentNode`-ra (ÉrtékadásCsomópont) hívjuk meg, az eredmény az `AssignmentNode::GenerateCode` lesz (ezzel egy értékadás kódját állítjuk elő). A végrehajtott művelet a kérelem fajtájától és a fogadó típusától is függ.

A kettős közvetítés mindössze annyit jelent, hogy a végrehajtott művelet a kérelem fajtája mellett *két* fogadó típusától függ. Az `Accept` kettős közvetítésű művelet, jelentése két típustól, a látogatóétól és az elemétől függően változik. A kettős közvetítés révén a látogatók különböző műveleteket kérhetnek minden elemosztályra.[11]

Ez a Látogató minta kulcsa: a végrehajtott művelet mind a látogató, mind a meglátogatott elem típusától függ. A műveleteket nem szerkesztjük be statikusan az Elem felületbe, hanem a Látogatóba helyezzük azokat, és az `Accept` használatával futásidőben hozzuk létre a kötést. Így az Elem felület bővítése csupán egyetlen új Látogató alosztály létrehozásával jár, nem pedig számos Elem alosztály meghatározásával.

2. *Ki felel az objektumszerkezet bejárásáért?* A látogatónak végig kell járnia az objektumszerkezet valamennyi elemét. A kérdés csak az, hogyan éri ezt el? A bejárás feladatát három helyen helyezhetjük el: az objektumszerkezetben, a látogatóban, illetve egy önálló bejáró objektumban (lásd a Bejáró tervezési mintát).

   Gyakran az objektumszerkezet felel a bejárásért. Egy gyűjtemény például egyszerűen a Fogad művelet többszöri meghívásával járja végig az elemeit. Az összetételek bejárása során a Fogad ismétlődő önhívással megy végig az elem gyermekein.

   Egy másik megoldás, ha az elemek meglátogatására egy bejárót használunk. A C++ nyelvben belső és külső bejárót is használhatunk, attól függően, hogy melyik elérhető, illetve melyik a hatékonyabb. A Smalltalkban általában belső bejárókat alkalmaznak, a `do:` és egy programblokk segítségével. Miután a belső bejárókat az objektumszerkezet valósítja meg, a belső bejárók használata nem sokban különbözik attól, mintha az objektumszerkezetet tennénk felelőssé a bejárásért. A fő különbség az, hogy a belső bejárók nem járnak kettős közvetítéssel: a műveletet a *látogatóra* hívják meg, és a művelet argumentuma egy adott *elem* lesz, nem pedig egy *elemre*, a *látogatóval* mint argumentummal. Mindazonáltal a Látogató minta könnyen használható belső bejáróval is, ha a látogató művelete egyszerű, önhívás nélküli művelethívást intéz az elemhez.

---

[11] Ha létezik *kettős* közvetítés, lehetséges *hármas*, *négyes* vagy még „többes" is? Nos, a kettős közvetítés valójában csak a **többszörös közvetítés** (multiple dispatch) egyik esete, amelyben a műveletet típusok *valamilyen* száma alapján választjuk ki. (A CLOS ténylegesen a többszörös közvetítést támogatja.) A kettős vagy többszörös közvetítést támogató nyelvekben kevésbé szükséges a Látogató minta alkalmazása.

A bejáró algoritmus a látogatóba is helyezhető, bár így a bejárás kódját minden összesített (aggregát) KonkrétElem minden KonkrétLátogató osztályában újra és újra megkettőzzük. A bejárási stratégiának a látogatóba helyezésére abban az esetben lehet jó okunk, ha különösen bonyolult bejárást valósítunk meg, amely az objektumszerkezeten végzett műveletek eredményétől függ. A Példakód részben látunk majd egy ilyen esetet.

Példakód

Mivel a látogatók általában az összetételekhez kapcsolódnak, a Látogató minta illusztrálásánál az Összetétel minta Példakód részében bemutatott `Equipment` (Eszközök) osztályokra fogunk támaszkodni. A Látogató mintát arra használjuk majd, hogy műveleteket határozzunk meg egy eszközleltár összeállításához, illetve az egyes eszközök teljes költségének kiszámításához. Az `Equipment` osztályok annyira egyszerűek, hogy a Látogató minta alkalmazása tulajdonképpen szükségtelen, de a példán könnyen bemutatható, mire van szükség a minta megvalósításához.

Álljon itt ismét az Összetétel mintánál (4. fejezet) megismert `Equipment` osztály, amelyet kibővítettünk egy `Accept` művelettel, hogy képes legyen együttműködni a látogatókkal:

```
class Equipment {
public:
   virtual ~Equipment();

   const char* Name() { return _name; }

   virtual Watt Power();
   virtual Currency NetPrice();
   virtual Currency DiscountPrice();

   virtual void Accept(EquipmentVisitor&);
protected:
   Equipment(const char*);
private:
   const char* _name;
};
```

Az `Equipment` műveletei egy adott eszköz tulajdonságait adják vissza, például a fogyasztását és a költségét. Az alosztályok a különböző eszköztípusoknak (ház, meghajtók, kártyák stb.) megfelelően felülírják a műveleteket.

Az eszközök látogatóinak elvont osztálya minden eszköz-alosztály számára tartalmaz egy elvont függvényt, amint azt az alábbi kódban láthatjuk. Alapértelmezés szerint a virtuális függvények egyike sem csinál semmit.

```
class EquipmentVisitor {
public:
   virtual ~EquipmentVisitor();

   virtual void VisitFloppyDisk(FloppyDisk*);
   virtual void VisitCard(Card*);
   virtual void VisitChassis(Chassis*);
   virtual void VisitBus(Bus*);

   // és így tovább az Equipment minden konkrét alosztályára
protected:
   EquipmentVisitor();
};
```

Az `Equipment` alosztályai az `Accept` műveletet lényegében ugyanúgy határozzák meg: a művelet meghívja azt az `EquipmentVisitor` műveletet, amelyik megfelel az `Accept` kérelmet kapó osztálynak:

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
   visitor.VisitFloppyDisk(this);
}
```

Azok az eszközök, amelyek más eszközöket foglalnak magukba (az Összetétel mintánál ezek a `CompositeEquipment` alosztályai), úgy valósítják meg az `Accept` műveletet, hogy bejárják gyermekeiket és mindegyikre meghívják. Ezután a szokásos módon meghívják a `Visit` műveletet. A `Chassis::Accept` például a házban található eszközöket a következőképpen járja be:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
   for (
         ListIterator<Equipment*> i(_parts);
         !i.Is.Done();
         i.Next()
) {
         i.CurrentItem()->Accept(visitor);
}
   visitor.VisitChassis(this);
}
```

Az `EquipmentVisitor` alosztályai a szerkezet egyes részeire vonatkozó egyedi algoritmusokat határozzák meg. A `PricingVisitor` (ÁrazóLátogató) az összes eszköz költségét számítja ki. Kiszámolja az egyszerű eszközök (pl. hajlékonylemezek) nettó árát, illetve az összetett eszközök (pl. ház) leszállított árát.

```
class PricingVisitor : public EquipmentVisitor {
public:
   PricingVisitor();
```

```
    Currency& GetTotalPrice();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    //...
private:
    Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
    _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

A `PricingVisitor` a szerkezet minden csomópontjának teljes költségét kiszámítja. Megfigyelhetjük, hogy az egyes eszközosztályok árazási módját úgy választja ki, hogy a megfelelő tagfüggvényhez fordul. Az árazási módszer így meg is változtatható, ehhez elég a `PricingVisitor` osztályt módosítani.

A leltárkészítő látogatót valahogy így határozhatjuk meg:

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor();

    Inventory& GetInventory();

    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
    //...

private:
    Inventory _inventory;
};
```

Az `InventoryVisitor` (LeltárLátogató) összegyűjti az egyes eszköztípusokra vonatkozó összegeket, emellett pedig az `Inventory` (Leltár) osztály segítségével felületet határoz meg az eszközök hozzáadására (ezt itt nem részletezzük).

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
   _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
   _inventory.Accumulate(e);
}
```

Az `InventoryVisitor`-t a következőképpen használhatjuk a teljes elemkészletre:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
   << component->Name()
   << visitor.GetInventory();
```

Most pedig megnézzük, hogyan valósítható meg az Értelmező mintánál bemutatott Smalltalk példa a Látogató minta használatával. Az előzőhöz hasonlóan ez a példa is olyan egyszerű, hogy a Látogató mintával valószínűleg nem nyerünk sokat, de illusztrációnak tökéletes, mert a minta használata mellett egy olyan helyzetet is bemutat, ahol a bejárás a látogató felelőssége.

Az objektumszerkezet (szabályos kifejezések) négy osztályból áll, melyek mindegyike rendelkezik egy `accept:` metódussal, amelynek argumentuma a látogató. A `SequenceExpression` (SorozatKifejezés) osztályban ennek alakja a következő:

```
accept: aVisitor
   ^ aVisitor visitSequence: self
```

Az `accept:` a `RepeatExpression` (IsmétlésKifejezés) osztályban a `visitRepeat:`, az `AlternationExpression` (VálasztásKifejezés) osztályban a `visitAlternation:`, a `LiteralExpression` (LiterálKifejezés) osztályban a `visitLiteral:` üzenetet küldi.

A négy osztálynak elérő függvényekkel is kell rendelkeznie, amelyeket a látogató használhat. A `SequenceExpression` esetében ezek az `expression1` és `expression2`, az `AlternationExpression` esetében az `alternative1` és `alternative2`, a `RepeatExpression` esetében a `repetition`, a `LiteralExpression`-nél pedig a `components`.

A KonkrétLátogató osztály a `REMatchingVisitor` (IsmIllesztőLátogató). Ez az osztály felel a bejárásért, mert a bejáró algoritmus nem szabályos, ami leginkább abban nyilvánul meg, hogy a `RepeatExpression` újra és újra bejárja az összetevőjét. A `REMatchingVisitor` osztály egy `inputState` nevű példányváltozóval rendelkezik, metódusai pedig

lényegében megegyeznek az Értelmező minta kifejezésosztályainak `match:` (illeszt) metódusaival, csak ezekben az `inputState` argumentum helyén az illesztendő kifejezés-csomópont áll. Mindazonáltal, amit visszaadnak, az továbbra is azon adatfolyamok halmaza, amelyekre a kifejezés az aktuális állapot azonosításához illeszkedik.

```
visitSequence: sequenceExp
   inputState := sequenceExp expression1 accept: self.
   ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
   | finalState |
   finalState := inputState copy.
   [inputState isEmpty]
         whileFalse:
               [inputState := repeatExp repetition accept: self.
               finalState addAll: inputState].
   ^ finalState

visitAlternation: alternateExp
   | finalState originalState |
   originalState := inputState.
   finalState := alternateExp alternative1 accept: self.
   inputState := originalState.
   finalState addAll: (alternateExp alternative2 accept: self).
   ^ finalState

visitLiteral: literalExp
   | finalState tStream |
   finalState := Set new.
   inputState
         do:
               [:stream | tStream := stream copy.
                     (tStream nextAvailable:
                           literalExp components size
                     ) = literalExp components
                           ifTrue: [finalState add: tStream]
               ].
   ^ finalState
```

Ismert felhasználások

A Smalltalk-80 fordítóprogram tartalmaz egy látogató osztályt, a ProgramNodeEnumerator (ProgramCsomópontFelsoroló) nevűt, amelyet elsősorban a forráskódot elemző algoritmusok használnak. Kód-előállításra vagy kimenetformázásra nem használatos, pedig alkalmas volna rá.

Az IRIS Inventor [Str93] térbeli grafikus alkalmazások fejlesztésére szolgáló elemkészlet. A háromdimenziós helyszíneket (scene) csomópontok hierarchiájaként ábrázolja, amelyek mindegyike egy-egy mértani alakzatot jelöl, vagy annak egy tulajdonságát. Az olyan műveletek, mint a helyszínek leképezése vagy a bemeneti események hozzárendelése ezen hierarchia különböző módokon való bejárását igényli, amit az Inventor műveleteknek (action) nevezett látogatókon keresztül old meg. Külön látogatók vannak a leképezésre, az eseménykezelésre, a keresésre, illetve a befoglaló dobozok meghatározására.

Az új csomópontok felvételét megkönnyítendő az Inventor kettős közvetítési sémát valósít meg C++-ban. A séma a futásidejű típusinformációkra épül, illetve egy kétdimenziós táblázatra, amelynek sorai a látogatókat, oszlopai pedig a csomópont osztályokat jelképezik. A cellák a látogatóhoz, illetve a csomópont osztályhoz kapcsolódó függvényt címző mutatót tartalmazzák.

A Látogató (Visitor) elnevezést Mark Linton használta először, az X Consortium Fresco Application Toolkit leírásában [LP93].

Kapcsolódó minták

Összetétel: A látogatók segítségével egy, az Összetétel mintával meghatározott objektumszerkezet elemein végezhetünk el egy adott műveletet.

Értelmező: Az értelmezést látogatóval is végezhetjük.

A viselkedési mintákról

A változatok egységbe zárása

A változó elemek egységbe zárása számos viselkedési minta alapja. Amikor egy program valamelyik tényezője gyakran változik, ezen minták segítenek abban, hogy egy objektumban egységbe zárhassuk. A program többi része ezután együttműködhet ezzel az objektummal, ha működésük az említett tényezőtől függ. A viselkedési minták általában egy elvont osztályt határoznak meg, amely leírja az egységbe záró objektumot, és nevüket erről az objektumról kapják:[12]

- a Stratégia objektum egy algoritmust zár egységbe (Stratégia minta),
- az Állapot objektum egy állapotfüggő viselkedést (Állapot minta),
- a Közvetítő objektum objektumok közötti protokollt (Közvetítő minta),
- a Bejáró objektum pedig az összesített objektumok (aggregátumok) elemeinek elérésére és bejárására szolgáló módszert.

Az említett minták a program egy olyan részét írják le, amelyet működés közben valószínűleg cserélgetünk. A legtöbb minta kétféle objektumot tartalmaz: új objektumokat, amelyek egységbe zárják a kérdéses szolgáltatást, illetve már meglevőket, amelyek ezen új objektumokat használják. Az új objektumok nyújtotta szolgáltatások az adott minta használata nélkül általában a már létező objektumok szerves részei lennének. Egy stratégia kódját például a stratégia környezetébe (Környezet, Context) „drótoznánk", egy állapotobjektum kódját pedig közvetlenül az állapot környezetében valósítanánk meg.

Mindazonáltal nem minden objektumviselkedési minta a fenti felosztást alkalmazza. A Felelősséglánc mintában például tetszőleges számú (a láncot alkotó) objektummal dolgozhatunk, amelyek mindegyike lehet a rendszerben már létező objektum.

A Felelősséglánc minta még egy különbségre rávilágít a viselkedési minták között: nem mindegyik statikus kapcsolatokat határoz meg az osztályok között. A Felelősséglánc minta objektumok korlátlan száma közötti kommunikációt ír elő, míg más mintákban olyan objektumokat találunk, amelyeket argumentumként adunk át.

Argumentumként használt objektumok

Számos tervezési minta vezet be egy olyan objektumot, amelyet *mindig* argumentumként használunk. Az egyik ilyen minta a Látogató. A látogató objektum egy többalakú Fogad (Accept) művelet argumentuma, amely a meglátogatott objektumokon működik.

[12] Más minták is hasonlóan működnek. Az Elvont gyár, az Építő és a Prototípus minták mind objektumok létrehozásának módját zárják egységbe; a Díszítő minta olyan szolgáltatásokat, amelyekkel egy objektum kiegészíthető; a Híd minta az elvont ábrázolást választja el a megvalósítástól, hogy azok egymástól függetlenül változtathatók legyenek.

A látogatót soha nem kezeljük ezen objektumok részeként, pedig a minta alkalmazásának hagyományos alternatívája a látogató kódjának elosztása az objektumszerkezet osztályai között.

Más minták olyan objektumokat határoznak meg, amelyek mágikus jelként viselkednek, amiket körbeadunk és később meghívunk. Mind a Parancs, mind az Emlékeztető minta ebbe a kategórába esik. A Parancs mintában a mágikus jel (magic token) egy kérelmet jelöl, az Emlékeztetőben egy objektum belső állapotát egy adott pillanatban. A jel mindkét esetben bonyolult belső szerkezetű lehet, de erről az ügyfélnek nincs tudomása. Mindazonáltal itt is találunk különbségeket. A Parancs mintában lényeges szerepet játszik a többalakúság (polimorfizmus), hiszen a Parancs (Command) objektum végrehajtása többalakú művelet. Ezzel szemben az Emlékeztető (Memento) felület olyan „keskeny", hogy az emlékeztető csupán értékként adható át, így nem valószínű, hogy egyetlen többalakú műveletet is nyújtana az ügyfeleinek.

Egységbe zárás vagy elosztás?

A Közvetítő és a Megfigyelő egymással versengő tervezési minták. A különbség köztük az, hogy a Megfigyelő a Megfigyelő (Observer) és Alany (Subject) objektumok bevezetésével elosztja a kommunikációt, míg a Közvetítő (Mediator) objektumok éppen hogy egységbe zárják a többi objektum közötti kapcsolattartást.

A Megfigyelő mintában nem egyetlen objektum zár egységbe egy kötést; a Megfigyelőknek és Alanyoknak együtt kell működniük annak fenntartása érdekében. A kapcsolattartási mintákat a megfigyelők és alanyok összekapcsolásának módja határozza meg: az egyedülálló alanyoknak általában több megfigyelőjük van, de egy megfigyelő is lehet egy másik megfigyelő alanya. A Közvetítő minta nem eloszt, inkább központosít; a kötések fenntartásának felelősségét kifejezetten a közvetítőkre bízza. A szerzők könnyebben újrahasznosíthatónak találták a Megfigyelő és Alany objektumokat, mint a közvetítőket. A Megfigyelő minta a megfigyelők és alanyok közötti elosztást és laza csatolást részesíti előnyben, ami „finomabb" osztályszerkezetet eredményez, a kisebb osztályokat pedig könnyebb újrahasznosítani.

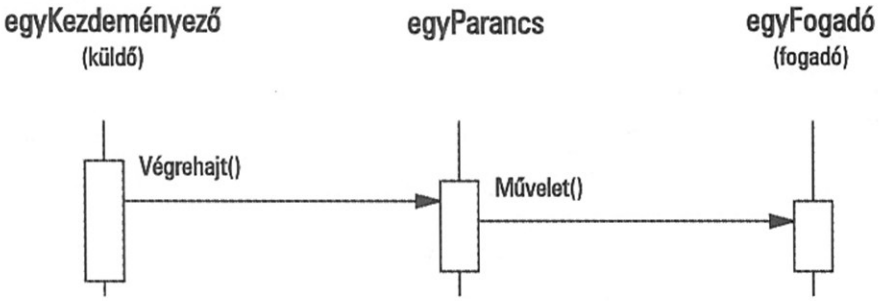
A Közvetítő mintában viszont átláthatóbb a kommunikáció folyása, mint a Megfigyelőben. A megfigyelők és alanyok általában röviddel létrehozásuk után összekapcsolódnak, így a programban később nehéz felderíteni kapcsolódásuk módját. Ha ismerjük a Megfigyelő mintát, tisztában vagyunk vele, hogy a kapcsolódás módjának ismerete lényeges, és azt is tudjuk, milyen kapcsolatokat keressünk, a minta által bevezetett közvetettség azonban ennek ellenére megnehezítheti egy rendszer megértését.

A megfigyelők a Smalltalkban üzenetparamétereket kaphatnak az alany állapotának eléréséhez, így újrahasznosításuk még egyszerűbb, mint a C++-ban. A Smalltalkban ezért vonzóbb a Megfigyelő minta, mint a Közvetítő, míg a C++ programozók inkább az utóbbit részesítik előnyben.

A küldő és a fogadó elválasztása

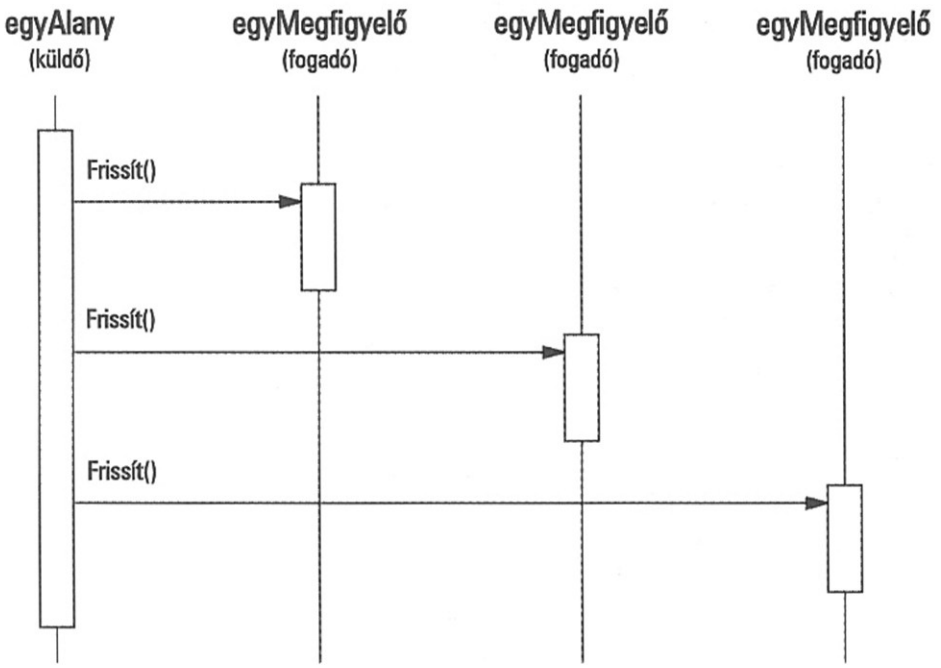
Amikor az együttműködő objektumok közvetlenül hivatkoznak egymásra, függővé válnak egymástól, ami nemkívánatos hatással van a rendszer rétegezettségére és újrahasznosíthatóságára. A Parancs, a Megfigyelő, a Közvetítő és a Felelősséglánc minták mind érintik a küldők (adók) és fogadók (vevők) elválasztásának kérdését, de más-más következményekkel.

A Parancs minta az elválasztást egy olyan Parancs (Command) objektummal támogatja, amelynek feladata a küldő és a fogadó kötésének meghatározása:

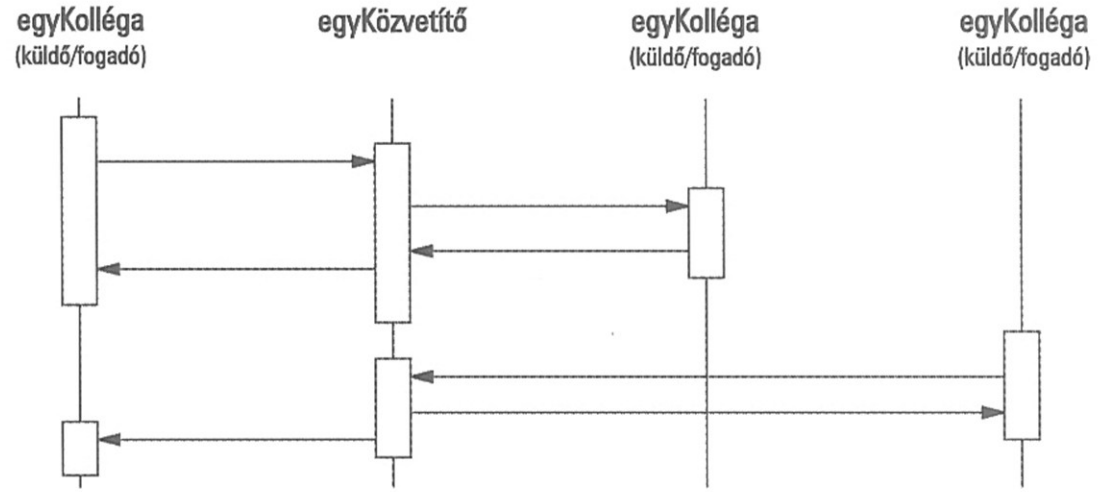


A Parancs objektum egyszerű felületet biztosít a kérelem kibocsátásához (vagyis a Végrehajt művelethez). A küldő–fogadó kapcsolat önálló objektumba helyezése lehetővé teszi a küldőnek, hogy különböző fogadókkal dolgozzon, emellett elválasztja a küldőt a fogadóktól, így megkönnyíti annak újrahasznosítását. A Parancs objektum is újrahasznosítható a fogadó különböző küldőkkel való paraméterezéséhez. A Parancs minta elvileg minden küldő–fogadó kapcsolathoz külön alosztályt igényel, de a minta leír olyan megvalósítási módszereket, amelyekkel az alosztályok származtatása elkerülhető.

A Megfigyelő minta az alanyok megváltozását jelző felület meghatározásával választja szét a küldőt (alany) a fogadótól (megfigyelő). A Megfigyelő minta lazább kötést alakít ki közöttük, mint a Parancs, mivel egy alanynak több megfigyelője lehet, és számuk futásidőben változhat.



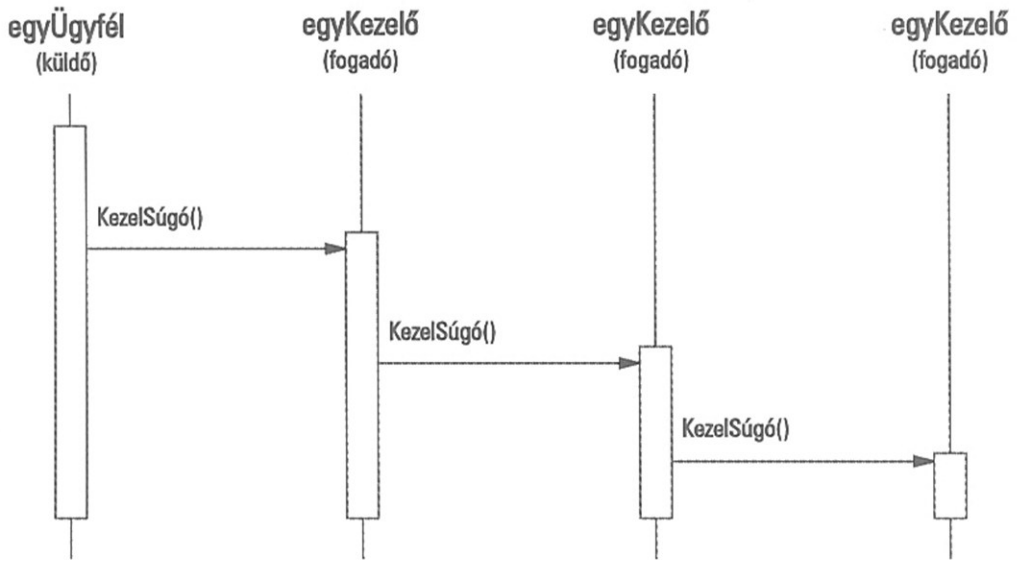
A Megfigyelő minta Alany és Megfigyelő felületeinek szerepe a változások közlése, ezért a minta akkor alkalmazható legjobban objektumok elválasztására, ha adatfüggőségek állnak fenn köztük. A Közvetítő minta az objektumokat úgy választja el, hogy megköveteli, hogy egy közvetítőn keresztül közvetetten hivatkozzanak egymásra.



A Közvetítő objektumok a kérelmeket Kolléga (Colleague) objektumokhoz továbbítják, és központosítják a közöttük folyó kommunikációt. Ennek következményeképpen a kollégák csak a közvetítő felületén keresztül társaloghatnak egymással. Mivel ez a felület rögzített, a nagyobb rugalmasság érdekében a közvetítőnek esetleg saját üzenetküldő sémát kell megvalósítania. A kérelmek kódolása és az argumentumok becsomagolása így oly módon történhet, hogy a kollégák korlátlan számú műveletet kérelmezhetnek.

A Közvetítő minta csökkenti az alosztályok létrehozásának szükségességét a rendszerben, mivel a kapcsolattartási viselkedést egyetlen osztályban egyesíti, nem pedig alosztályok között osztja szét. Mindazonáltal az *ad hoc* üzenetküldő sémák gyakran csökkentik a típusbiztonságot.

A Felelősséglánc minta a küldőt és a fogadót úgy választja el, hogy a kérelmeket a lehetséges fogadók láncán küldi át:



Miután a küldők és fogadók közötti felület rögzített, a Felelősséglánc minta szintén egyedi üzenetküldő sémát igényelhet, ami ugyanazokat a típusbiztonsággal kapcsolatos gondokat veti fel, mint a Közvetítő mintánál. A Felelősséglánc minta a küldő és a fogadó szétválasztására akkor jó megoldás, ha a lánc már a rendszer szerkezetének része, és valamelyik objektum képes kezelni a kérelmet. A minta emellett rugalmas, mert a lánc módosítható és könnyen bővíthető.

Összegzés

Néhány kivételtől eltekintve a viselkedési minták kiegészítik és erősítik egymást. Egy felelősséglánc egyik osztálya például valószínűleg legalább egy helyen alkalmaz Sablonfüggvényt; a sablonfüggvény alapműveletek segítségével megállapíthatja, hogy az objektum képes-e a kérelem kezelésére, illetve kiválaszthatja azt az objektumot, amelynek a kérelmet továbbítani kell. Ezenkívül a lánc a Parancs minta alkalmazásával a kérelmeket objektumokként ábrázolhatja. Az Értelmező minta az elemző környezetet az Állapot minta segítségével alakíthatja ki. Egy bejáró bejárhat egy összetételt, annak elemeire pedig egy látogató alkalmazhat műveleteket.

A viselkedési minták jól működnek együtt más mintákkal. Egy, az Összetétel mintát használó rendszer például egy látogató segítségével hajthat végre műveleteket az összetétel elemein, a Felelősséglánc minta révén lehetővé teheti az elemeknek, hogy szülőjükön keresztül globális tulajdonságokat érjenek el, sőt, a Díszítő mintával az összetétel egyes elemeiben felül is bírálhatja ezen tulajdonságokat. Emellett a Megfigyelőt mintát alkalmazva egy objekumszerkezetet egy másikhoz köthet, az Állapot mintával pedig megváltoztathatja egy elem viselkedését, amint az állapota megváltozik. Maga az összetétel létrehozható az Építő minta megközelítésével, a rendszer valamely más része pedig prototípusként is kezelheti az összetételt.

A jól megtervezett objektumközpontú rendszerek éppen ilyenek – számos minta ágyazódik beléjük, de nem feltétlenül azért, mert tervezőik erre törekedtek. Az osztályok vagy objektumok szintje helyett a *minták* szintjén történő tervezés számunkra is megkönnyíti, hogy hasonló szervezettséget érjünk el.

6

Tanulságok

Egyesek úgy gondolhatják, számukra ez a könyv nem nyújtott túl sokat. Tény, hogy nincsenek benne korábban ismeretlen algoritmusok és különféle programozási eljárások. Nem ad szigorú módszert a rendszerek tervezésére, nem dolgoz ki új tervezési elméletet, csupán leírja a létező tervezési mintákat. Így joggal adódhat a következtetés, hogy bármennyire jól összeállított oktatóanyag kezdőknek, az objektumközpontú tervezésben jártas programozók számára nem tartalmaz hasznosítható ismereteket.

Reméljük, az Olvasó másként gondolja és egyetért velünk abban, hogy a tervezési minták rendszerezése lényeges, mert szabványos meghatározást és nevet ad az általunk használt eljárásoknak. Ha nem tanulmányozzuk a különféle programokban fellelhető tervezési mintákat, továbbfejlesztésükre sem leszünk képesek, és nehezebben állunk majd elő újabbakkal.

Ez a könyv csupán a kezdőlökést adhatja meg. Azokat a leggyakrabban használt tervezési mintákat tartalmazza, amelyeket a tapasztalt objektumközpontú fejlesztők a gyakorlatban nap mint nap alkalmaznak, tudomást mégis csak szájhagyomány útján vagy a meglevő rendszerek tanulmányozásával szereznek róluk. A kötet korai vázlataiban arra buzdítottuk az olvasókat, hogy írják le az általuk használt tervezési mintákat: reményeink szerint a végleges változat még inkább erre sarkall. Reméljük, ez kezdete lesz egy mozgalomnak, amely a szoftverfejlesztők gyakorlati tapasztalatait végre írásban rögzíti.

Eme utolsó fejezetet annak szenteljük, hogy bemutassuk, szerintünk milyen hatást gyakorolnak a tervezési minták a szoftverfejlesztésre, hogyan kapcsolódnak a tervezéssel kapcsolatos egyéb részfeladatokhoz, hogyan találhatunk magunk is itt le nem írt mintákat, és hogyan rendszerezhetjük azokat.

6.1 Mit várjunk egy tervezési mintától?

Íme néhány terület, amit a kötetben bemutatott tervezési minták az objektumközpontú programok fejlesztése során érintenek (gyakorlati tapasztalataink alapján).

Közös tervezési szókincs

A programozó szakemberek tapasztalatai a hagyományos programnyelvek terén azt mutatják, hogy pusztán a nyelvtan ismerete nem nyújt elégséges tudást; szükség van a nagyobb fogalmi szerveződések, az algoritmusok, adatszerkezetek, nyelvjárások vagy idiómák [AS85, Cop92, Cur89, SS86], illetve az egy adott célhoz igazított tervek [SE84] elsajátítására is. A tervezők az adott programterv rögzítésére használt jelölési rendszerre kevesebb figyelmet fordítanak; inkább arra összpontosítanak, hogy a tervet a korábban megismert algoritmusokhoz, adatszerkezetekhez és idiómákhoz igazítsák.

A számítógép-tudomány igyekszik elnevezni és rendszerezni az algoritmusokat és adatszerkezeteket, de a tervezési minták esetében ez igen ritkán áll fenn. Pedig a rendszerezett tervezési minták közös nyelvet biztosíthatnak a különböző megoldások ismertetéséhez, leírásához vagy felfedezéséhez; emellett azáltal, hogy segítségükkel a programozási nyelveknél és más jelölési rendszereknél magasabb szinten elvonatkoztatva „beszélhetünk" egy adott tervről, a programot kevésbé bonyolultnak tüntetik fel. A tervezési minták tehát egyszerűsítik mind a tervezést, mind a tervről munkatársainkkal folytatott vitákat.

Ha elsajátítottuk a kötetben szereplő tervezési mintákat, tervezési szókincsünk is egészen biztosan meg fog változni, ahogy elkezdünk az ezek által használt fogalmakban gondolkodni. Fel sem fog tűnni, hogy máris olyanokat mondunk, hogy „használjuk itt a Megfigyelőt" vagy „csináljunk Stratégiát ezekből az osztályokból".

Dokumentáció és tanulási segédlet

A könyvben bemutatott tervezési minták megtanulása könnyebbé teszi a létező rendszerek megértését, hiszen a legtöbb nagy méretű objektumközpontú program alkalmazza őket. Az objektumközpontú programozást tanulók gyakran panaszkodnak, hogy a rendszer, amellyel dolgoznak, bonyolult öröklési viszonyokra épül, így nehéz követni a vezérlési folyamatot. Gondjuk nagyrészt abból adódik, hogy nem értik a rendszerben jelen levő tervezési mintákat, pedig ezek ismerete feltétlenül segít az objektumközpontú rendszerek működésének megértésében.

A tervezési minták segítségével jobb programtervezővé is válhatunk, hiszen e minták gyakran felbukkanó problémákra nyújtanak megoldást. Ha kellően hosszú ideig dolgozunk objektumközpontú rendszerekkel, a mintákat valószínűleg magunk is felfedezzük, de e könyvet elolvasva a tanulási idő jócskán lerövidíthető, így az újonc programozók hamarabb szakértővé válhatnak.

Emellett, ha egy rendszert az általa alkalmazott tervezési minták nevével írunk le, felépítését megérteni is jóval könnyebb lesz. Ha nem így teszünk, arra kényszerülünk, hogy a szer-

kezetet visszafelé „göngyölítve" jussunk el a használt minták kibányászásáig. A közös szókincs használata révén nem kell részletesen leírnunk a teljes szerkezetet; elég, ha megnevezzük, így mindenki tudni fogja, miről van szó. Ha mégsem, az illetőnek csak utána kell néznie egyszer, hogy milyen tervezési mintát is takar az adott név, ami még mindig egyszerűbb, mint a visszafejtés.

A szerzők saját munkáikban is alkalmazzák e mintákat, és felbecsülhetetlen értékűnek tartják azokat. Természetesen a felhasználás módja legtöbbször „naiv": a minták segítségével nevet keresünk az osztályoknak, a helyes tervezés oktatásában alkalmazzuk őket, vagy a felhasznált tervezési mintákat sorba állítva leírunk egy adott programszerkezetet [BJ94]. Ennél kifinomultabb alkalmazás is elképzelhető; a tervezési minták alapján például CASE eszközök vagy hiperszöveges dokumentumok is készíthetők, de a minták alapszinten is nagy segítséget nyújtanak.

A létező módszerek kiegészítője

Az objektumközpontú tervezési módszerek célja, hogy jó tervezésre ösztökéljenek, hogy megtanítsák a kezdő programozóknak a helyes tervezés mikéntjét, illetve hogy szabványosítsák a programfejlesztés módját. Egy tervezési módszer jellemzően azt írja le, hogy a programszerkezet különböző elemeinek modellezésére milyen (általában grafikus) jelölésrendszert használunk, illetve azon szabályokat, amelyek arra vonatkoznak, hogy az egyes szimbólumokat mikor és hogyan alkalmazzuk. A tervezési módszerek rendszerint lehetséges problémákat vázolnak, megoldást adnak azokra, és leírják, hogyan mérhetjük fel a tervezés helyességét, a szakértő programozók tapasztalatainak megragadására azonban eddig alkalmatlannak bizonyultak.

Úgy véljük, a bemutatott tervezési minták azt a láncszemet jelenthetik, ami eddig hiányzott az objektumközpontú tervezési módszerekből. Megmutatják, hogyan használhatjuk az alapvető programelemeket – az objektumokat, az öröklést vagy a többalakúságot –, illetve azt, hogyan paraméterezhetünk egy rendszert algoritmusokkal, viselkedésekkel, állapotokkal vagy azokkal az objektumokkal, amelyeket létre kell hoznia. Nem csupán döntéseink eredményét rögzítik, hanem azt írják le, „miért" legyen olyan a programszerkezet, amilyen. A döntések meghozatalában az egyes tervezési minták ismertetésében szereplő Alkalmazhatóság, Következmények és Megvalósítás részek segíthetnek minket.

A tervezési minták különösen hasznosak, ha egy elemző modellből szeretnénk megvalósítási modellt létrehozni. Bár egyesek azt állítják, hogy az objektumközpontú elemzésből zökkenőmentes az átmenet az objektumközpontú tervezésbe, a gyakorlatban ez az átmenet minden, csak nem sima. Egy rugalmas és újrahasznosítható programszerkezet olyan objektumokat is tartalmaz, amelyek az elemző modellben nem szerepelnek, a tervet pedig a használt programozási nyelv és osztálykönyvtár is befolyásolja. Az elemző modelleket gyakran át kell dolgozni, hogy újrahasznosíthatóvá váljanak. Számos, a gyűjteményünkben szereplő minta foglalkozik e kérdéskörrel, ezért is hívjuk őket *tervezési* mintáknak.

Egy teljes tervezési módszer nem csupán tervezési, hanem más típusú mintákat is igényel, például elemzési, felhasználóifelület-tervezési vagy teljesítményfokozási mintákat. A tervezési minták azonban olyan lényeges területet jelentenek, amelyről eddig nem sok szó esett.

Az újraépítés célja

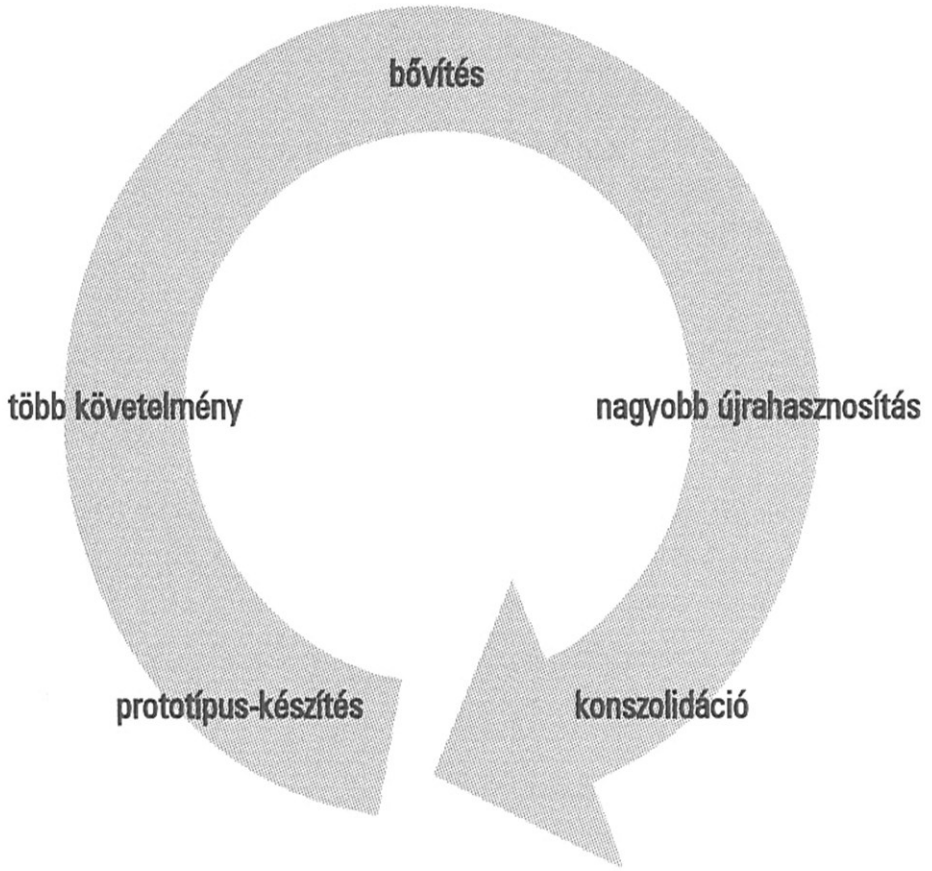
Az újrahasznosítható programok fejlesztésének egyik akadálya, hogy gyakran újjászervezésükre vagy újraépítésükre van szükség. (Ezt hívják idegen szóval **refaktorizációnak** [OJ90].) A tervezési minták segítenek abban, hogy megállapítsuk, milyen újjászervezés szükségeltetik, és a későbbi újraépítés igényét is csökkentik.

Az objektumközpontú programok élete több szakaszra osztható. Brian Foote ezeket a szakaszokat a **prototípus-készítési**, **bővítési**, illetve **konszolidációs** jelzőkkel illeti [Foo92].

A prototípus-készítési szakasz számos tevékenységet foglal magába; mindazokat a tevékenységeket, amelyek révén a szoftver az első változat fokozatos bővítésével, módosításával lassan eléri a „nagykorúságot", vagyis képessé válik arra, hogy megfeleljen az előzetesen támasztott követelményeknek. A program ekkor általában olyan osztályhierarchiákból épül fel, amelyek szorosan illeszkednek az előzetes terv által meghatározott fogalmakhoz. Az ebben a szakaszban alkalmazott újrahasznosítás többnyire öröklés útján megvalósított „fehér dobozos" újrahasznosítás.

Amikor az immár nagykorú szoftvert működésbe helyezik, további fejlesztését két egymásnak ellentmondó szükséglet határozza meg: (1) a szoftvernek további követelményeket kell kielégítenie, miközben (2) még inkább újrahasznosíthatóvá kell válnia. Az új követelmények általában új osztályok és műveletek, esetleg teljes osztályhierarchiák hozzáadását igénylik, vagyis a szoftver egy bővítési szakaszon megy át. E szakasz ugyanakkor nem nyúlhat túl hosszúra, mert különben a szoftver túlságosan rugalmatlanná, a későbbi változtatásokkal szemben túl nehézkessé válhat. Az osztályhierarchiák e szakaszban áttörik az előzetes terv által támasztott korlátokat, más fogalomkörökre is kiterjednek, az osztályok pedig számos önálló műveletet és példányváltozót is meghatároznak.

Ahhoz, hogy továbbfejlődhessen, a program újjászervezésére van szükség: e folyamatot nevezik *újraépítésnek* (refaktorizáció). A keretrendszerek rendszerint ebben a szakaszban lépnek a képbe. Az újraépítés magába foglalja osztályok általános és szakosított célú összetevőkre bontását, műveletek feljebb vagy lejjebb léptetését az osztályhierarchiában, illetve osztályfelületek ésszerűsítését is. Ezen konszolidációs szakaszban számos új objektum jön létre, gyakran a meglevők felbontásával, illetve öröklés helyett objektum-összetétel használatával, tehát a „fehér dobozos" újrahasznosítást felváltja a „fekete dobozos" újrahasznosítás. Az a folyamatosan fennálló igény, hogy újabb és újabb követelményeknek kell megfelelni, illetve még több kódrészletet lehessen újrahasznosítani, az objektumközpontú szoftvert a bővítés és konszolidáció körforgásába helyezi.



Az ábrán is látható ciklus kialakulása elkerülhetetlen, de a jó tervezők tisztában vannak azzal, milyen változások kényszeríthetik ki az újraépítést, és ismerik azokat az osztály- és objektumszerkezeteket is, amelyek segíthetnek az újraépítés elkerülésében. A követelmények alapos elemzése fényt deríthet arra, mely igények változnak majd meg nagy valószínűséggel a szoftver élete során; a jó programszerkezet pedig felkészül ezekre a változásokra.

Tervezési mintáink számos, az újraépítés következményeként előálló szerkezetet megragadnak; ha a mintákat a fejlesztés korai szakaszában alkalmazzuk, segítségükkel elkerülhető a későbbi újraépítés, de még abban az esetben is hasznosak, ha a rendszer felépítésének befejezéséig nem tudjuk, hogyan alkalmazzuk őket – ekkor a rendszer módosításának módját mutathatják meg.

6.2 Egy kis történelem

A gyűjtemény összeállítását Erich kezdte el, Ph.D. szakdolgozatához [Gam91, Gam92], melyben az itt bemutatott minták mintegy fele kapott helyet. Az OOPSLA '91 idejére már hivatalosan is önálló katalógussá vált, Richard pedig csatlakozott Erich-hez, hogy együtt dolgozzanak rajta. John nem sokkal később csatlakozott. Ralph az OOPSLA '92 idejére lett a csoport tagja. Keményen dolgoztunk, hogy az ECOOP '93-ra közzétehető állapotba hozzuk a gyűjteményt, de hamar rájöttünk, hogy egy kilencven oldalas dolgozatot ott nem mutathatunk be, ezért kivonatot készítettünk belőle, és azt nyújtottuk be. El is fogadták. Röviddel ezután úgy döntöttünk, a katalógust könyvvé alakítjuk.

A tervezési minták neve az idők során némileg változott: a Burkolóból (Wrapper) Díszítő (Decorator) lett, a Ragasztóból (Glue) Homlokzat (Facade), a Pasziánszból (Solitaire) Egyke (Singleton), a Sétálóból (Walker) Látogató (Visitor). Néhány mintát elvetettünk, mert nem tűntek eléggé fontosnak. Ezektől eltekintve azonban a gyűjtemény többé-kevésbé változatlan maradt 1992 vége óta, bár maguk a minták rengeteget fejlődtek.

Észrevenni, hogy valami mintát alkot, könnyű. Mind a négyen régóta dolgozunk objektumközpontú rendszerekkel, így tudjuk, hogy ha valaki elég sok rendszert látott már, rögtön kiszúrhatja a tervezési mintákat. A minták *leírása* azonban sokkal nehezebb, mint *megtalálásuk*.

Ha rendszereket építünk, majd visszatekintünk rájuk, látjuk a mintákat munkánkban, de úgy leírni azokat olyanok számára, akik nem ismerik őket, hogy megértsék működésüket és fontosságukat, nagyon nehéz. A szakemberek már korai formájában felismerték a gyűjtemény jelentőségét, de csak azok értették meg a mintákat, akik már alkalmazták őket.

Miután a könyv egyik fő célkitűzése az volt, hogy a kezdő fejlesztőknek megtanítsuk az objektumközpontú tervezés mikéntjét, tudtuk, hogy érdekesebbé kell tennünk a katalógust. Az egyes mintákat leíró átlagosan két oldalt egy-egy részletes példával és saját munkára ösztönző mintakóddal tíznél is többre bővítettük. Azt is belevettük a kötetbe, hogy az egyes minták alkalmazásának milyen területei, illetve milyen hátrányai vannak. Mindez a könnyebb tanulást segíti.

Egy másik fontos szempont, amely felé a könyv megírása során eltolódott a hangsúly, az volt, hogy bemutassuk a problémát, amit az adott minta megold. Egy probléma megoldásaként gondolni rá, vagy olyan módszerként, amelyet újra és újra felhasználva egy adott feladathoz igazíthatunk, megkönnyíti a minta megtanulását. A problémakör, illetve annak leírása, hogy milyen összefüggésben bizonyul az adott minta a legjobb megoldásnak – vagyis hogy mikor *megfelelő* –, persze jóval nehezebb. Általában véve mindig könnyebb tudni, *mit* csinál valaki, mint tudni, hogy *miért* – a „miért" pedig a tervezési minták esetében a megoldandó problémát jelenti. A minta alkalmazási céljának ismerete igen fontos, hiszen ennek alapján választhatjuk ki, melyik mintát is kell felhasználnunk, de a már létező rendszerek felépítésének megértésében is segít. A tervezési minta kidolgozója tehát meg kell, hogy határozza, és le kell, hogy írja a problémát, amit az adott minta megold – még ha a megoldás felfedezése után is.

6.3 A tervezési minták közössége

Nem mi vagyunk az egyetlenek, akik szakemberek által használt tervezési mintákat leíró gyűjteményeket állítanak össze: egy nagyobb közösséghez tartozunk, melynek tagjait általában véve a minták, különösképpen pedig a szoftverminták érdeklik. Christopher Alexander például az első építész volt, aki tanulmányozta az épületekben és közösségekben fel-

lelhető mintákat, és az ilyen minták alkotására kifejlesztett egy „mintanyelvet". Munkája mindnyájukra nagy hatással volt, ezért érdemes röviden összevetni a mi eredményeinket az övéivel. Ezután másoknak a szoftverminták terén végzett kutatásaira is kitérünk.

Alexander mintanyelvei

Munkánk sok szempontból hasonlít Alexanderére. Mindkettő meglevő rendszerek vizsgálatára és bennük minták keresésére épül. Mindkettő sablonokat alkalmaz a minták leírására (bár a mi sablonjaink teljesen más jellegűek). Mindkettő hétköznapi nyelven, sok-sok példával mutatja be a mintákat, nem formális nyelvek segítségével, és mindkettő leírja, mire adnak választ az egyes minták.

Mindazonáltal eltérések is jócskán adódnak:

1. Házakat több ezer éve épít az emberiség, így rengeteg klasszikus példa áll előttünk, amelyekből meríthetünk. A szoftverrendszerek építése azonban viszonylag rövid időre tekint vissza, és „klasszikussá" sem túl sok vált közülük.
2. Alexander sorrendet állít fel a minták alkalmazására; mi nem tettük.
3. Alexander mintái a hangsúlyt a megoldandó problémára fektetik, míg a mi tervezési mintáink a megoldást írják le részletesebben.
4. Alexander állítása szerint mintáiból teljes épületek építhetők fel. Mi nem állítjuk, hogy mintáink teljes programokat adnak ki.

Amikor Alexander azt állítja, hogy pusztán mintái egymás utáni alkalmazásával megtervezhetünk egy házat, ugyanaz a cél lebeg a szeme előtt, mint azoknak az objektumközpontú tervezési módszertant oktató szakembereknek, akik szigorú szabályokat állítanak fel a tervezés lépéseire. Alexander persze nem tagadja a kreativitás szükségességét. Egyes mintái megkövetelik, hogy ismerjük azoknak az embereknek a szokásait, akik majd az épületet lakják, a tervezés „költészetébe" vetett hite pedig arról árulkodik, hogy a mintanyelv ismerete önmagában nem jelenti a tudás teljességét[1]. Mindazonáltal leírása arról, hogyan építik fel a minták a teljes szerkezetet, világossá teszi, hogy a mintanyelvek a tervezési folyamatot körülhatárolhatóbbá és egyszerűen megismételhetővé tehetik.

Alexander nézőpontját alapul véve a tervezést befolyásoló „erőkre" összpontosítottunk. Hatására igyekeztünk jobban megérteni, hogyan és milyen következményekkel alkalmazhatók tervezési mintáink, formális ábrázolásuk miatt viszont nem aggódtunk. A formális ábrázolás természetesen segíthetné a minták alkalmazásának automatizálását, de jelen pillanatban sokkal fontosabb, hogy feltárjuk a tervezési minták működését, mint hogy formalizáljuk őket.

Alexander meghatározása szerint az e könyvben bemutatott minták nem alkotnak mintanyelvet. Ha a manapság készített szoftverrendszerek változatosságára gondolunk, beláthat-

[1] Lásd: *The poetry of the language* [AIS+77].

juk, hogy nehéz is lenne egy „teljes" mintakészletet összeállítani, ami az alkalmazások tervezéséhez lépésről lépésre vezető utasításokat ad. Egyes programfajták – például a jelentéskészítő vagy űrlapos programok – esetében persze ezt is megtehetjük, de ez a katalógus csak egymáshoz kapcsolódó minták gyűjteménye, nem teljes mintanyelv.

Igazság szerint úgy gondoljuk, valószínűleg *soha* nem is lesz teljes mintanyelv szoftverrendszerek számára, bár olyat mindig lehet készíteni, ami *teljesebb* a korábbiaknál. Ilyen kiegészítést jelenthetnek például a keretrendszerek és használatuk [Joh92], a felhasználói felület tervezési mintái [BJ94], az elemzési minták [Coa92], illetve a szoftverfejlesztés más szempontjai. A tervezési minták csupán egy szeletét jelentik egy nagyobb „szoftver-mintanyelvnek".

Szoftverminták

Első közös élményünket a szoftverépítés kutatásának területén az OOPSLA '91-en, a Bruce Anderson által vezetett műhelyben szereztük, amely egy szoftverépítőknek szánt kézikönyv kidolgozásán fáradozott. (Bár a „Szoftverépítők enciklopédiája" név jobban illett volna rá.) A műhelynek folytatása is lett; számos találkozó követte, például 1994 augusztusában az első, a programok mintanyelveivel foglalkozó konferencia. Ennek hatására jött létre a szoftvertervezési tapasztalatokat dokumentálni kívánó közösség.

Természetesen mások is tűztek ki hasonló célokat. Donald Knuth munkája, a *The Art of Computer Programming* (A számítógép-programozás művészete) [Knu73] – bár az algoritmusok leírására összpontosított – egyike volt az első kísérleteknek, amelyek a tapasztalatok egybegyűjtésére irányultak. A feladat azonban túlságosan hatalmasnak bizonyult ahhoz, hogy a végére lehessen érni. Egy másik, szintén az algoritmusokat középpontba helyező tervezésmódszertani gyűjtemény volt a *Graphics Gems* sorozat [Gla90, Arv91, Kir92], míg az Egyesült Államok hadügyminisztériuma által támogatott Domain Specific Software Architecture program [GM92] vizsgálatának tárgyát az architektúrákkal kapcsolatos információk képezték. A tudás alapú szoftverfejlesztéssel foglalkozó közösség a szoftverrendszerekkel általánosságban kapcsolatos tudást gyűjti össze, és még számos csoport célja hasonlít legalább egy kicsit a miénkre.

James Coplien *Advanced C++: Programming Styles and Idioms* (C++ haladóknak: programozási stílusok és nyelvjárások) [Cop92] című könyve ugyancsak hatással volt ránk. Az ő mintái a mieinknél sokkal jobban kötődnek a C++ nyelvhez, és a kötet több alacsonyszintű mintát is tartalmaz, de számos átfedés található, amit könyvünkben jeleztünk is. Jim a mintákat kutató közösség munkájában aktívan részt vesz; jelenleg a szoftverfejlesztő szervezetekben dolgozó emberek szerepének leírásával foglalkozik.

Számos más helyen is találhatunk mintaleírásokat. Kent Beck volt az első, aki felhívta a szoftverfejlesztők figyelmét Christopher Alexander munkájára. 1993-ban cikksorozatot indított a *The Smalltalk Report*-ban a Smalltalk mintáiról. Peter Coad is jó ideje gyűjti a mintákat; a róluk írt dolgozata [Coa92] – számunkra legalábbis úgy tűnik – leginkább elemzési

mintákat tartalmaz. Legutóbbi munkáját e könyv írásakor még nem láttuk, de tudjuk, hogy újabb mintákon dolgozik. Tudomásunk van több folyamatban levő könyvről is, amelyek a mintákkal foglalkoznak (többek között a Pattern Languages of Programs konferenciáról is megjelenik majd egy), de még ezeket sem volt alkalmunk szemügyre venni, így csak annyit tehetünk, hogy tudatjuk az Olvasóval, újabb munkák várhatók e témában.

6.4 Meghívó

Mit tehetünk, ha érdeklődünk a minták iránt? Először is használjuk őket, és keressünk magunk is a munkánkhoz illő mintákat. Az elkövetkezendő években rengeteg könyv és cikk lát majd napvilágot e témában, így a keresést számos forrás segítheti. Emellett célszerű elsajátítani a tervezési minták nyelvezetét, és a tervezéssel kapcsolatos viták során munkatársainkkal használni. Gondolkodjunk a minták fogalmaiban.

Másodszor, legyünk kritikusak. Ez a tervezésiminta-gyűjtemény nem csupán a mi kemény munkánk eredménye, hanem azon több tucatnyi kritikusé is, akik észrevételeikkel segítettek minket. Ha az Olvasó észrevesz egy hibát, vagy úgy érzi, egy ponton részletesebb magyarázatra lenne szükség, ne habozzon kapcsolatba lépni velünk. Ugyanez persze bármely más mintagyűjteményre is érvényes: mondjuk el véleményünket a szerzőknek! A tervezési minták egyik nagy előnye, hogy alkalmazásuk révén a tervezési döntések nem ösztönösen születnek, így pontosan megfogalmazhatjuk, mit miért javaslunk. Ebből következik, hogy a minták hibáira is könnyebb rámutatni és a szerzővel vitába szállni. Éljünk a lehetőséggel!

Harmadszor, fedezzük fel a saját magunk által használt mintákat, és írjuk le azokat. Legyen ez a programdokumentáció része, amit megmutatunk másoknak is. A minták felfedezéséhez nem kell kutatónak lennünk, sőt, a lényeges minták megtalálása szinte lehetetlen, ha nem rendelkezünk kellő gyakorlati tapasztalattal. Nyugodtan összeállíthatjuk saját mintagyűjteményünket is, ha van valaki, aki segít gatyába rázni őket.

6.5 Búcsúzóul

A legjobb tervek számos tervezési mintát használnak fel, amelyek egymással összefonódva segítik sikerre a terv egészét. Ahogy Christopher Alexander mondja:

Építkezhetünk úgy, hogy a mintákat lazán összefűzzük: ekkor az épület önálló mintákból áll majd. Nem lesz „sűrű", nem lesz „mély". De ha a mintákat úgy rakjuk össze, hogy több is átfedje egymást ugyanazon a teren belül, az apró tér számos jelentést rejt majd, és ezek sűrű szövedékéből az épület mélységet nyer.

A Pattern Language [AIS+77, *xli.* oldal]

A

Szószedet

aláírás (szignatúra) Egy művelet neve, paraméterei és visszatérési értéke.

alosztály Másik osztálytól öröklő osztály. A C++-ban az alosztályt **származtatott osztálynak** (derived class) hívják.

alrendszer Adott feladatkört együttműködve ellátó osztályok független csoportja.

altípus Egy típus akkor altípusa egy másiknak, ha felülete tartalmazza a másik típus felületét.

átruházás (képviselet, delegáció) Megvalósítási módszer, amelyben az objektumok más objektumokhoz továbbítják vagy más objektumokra ruházzák át egyes kérelmek végrehajtását. A megbízott az eredeti objektum nevében végzi el a kért műveletet.

barát osztály A C++ nyelvben olyan osztály, amely egy másik osztály adataihoz és műveleteihez ugyanolyan jogosultságokkal rendelkezik, mint maga a tulajdonos osztály.

csatolás Annak foka, hogy az egyes szoftverösszetevők mennyire függnek egymástól.

destruktor (megsemmisítő függvény) A C++ nyelvben olyan művelet, ami minden esetben automatikusan meghívódik,amikor egy objektum törlésére kerül sor.

dinamikus kötés vagy **késői kötés** Kérelem futás közbeni összekapcsolása egy objektummal, illetve annak egy műveletével. A C++-ban csak a virtuális függvények késői kötésűek.

egységbe zárás vagy **betokozás** Ábrázolás és megvalósítás objektumba rejtésének eredménye. Az ábrázolás így nem látható és az objektumon kívülről közvetlenül nem elérhető; elérése és módosítása kizárólag műveleteken keresztül lehetséges.

együttműködési diagram (interakció-diagram) Diagram, amely a kérelmek útját ábrázolja az objektumok között.

elemkészlet Osztálygyűjtemény, amely hasznos szolgáltatásokat biztosít, de nem határozza meg egy alkalmazás szerkezetét.

elvont csatolás Ha adott egy *A* osztály, amely a *B* elvont osztályra hivatkozik, azt mondjuk, *A* elvontan csatolt *B*-hez. Ezt azért hívjuk elvont csatolásnak, mert az *A* egy bizonyos *objektumtípusra*, és nem konkrét objektumra hivatkozik.

elvont művelet Olyan művelet, amely megad egy adott aláírást (szignatúrát), de annak megvalósítását nem tartalmazza. A C++ nyelvben az elvont művelet megfelelője a **tisztán virtuális tagfüggvény**.

elvont osztály Olyan osztály, amelynek elsődleges feladata egy felület meghatározása. Az elvont vagy absztrakt osztály megvalósításának egy részét vagy egészét alosztályokra bízza. Nem példányosítható.

fehér dobozos újrahasznosítás Osztályöröklésen alapuló újrahasznosítási módszer, melynek során egy alosztály felhasználja szülőosztályának felületét és megvalósítását, de hozzáférhet a szülő egyébként rejtett adataihoz is.

fekete dobozos újrahasznosítás Objektum-összetételen alapuló újrahasznosítási módszer. Az összetett objektumok nem fedik fel belső felépítésüket egymás számára, ezért hasonlítják őket a repülők fekete dobozához.

felület (interfész) Egy objektum műveletei által meghatározott aláírások (szignatúrák) összessége. A felület azon kérelmek halmazát írja le, amelyeket az objektum teljesíthet.

felülírás vagy **felülbírálás** (Szülőosztálytól örökölt) művelet új meghatározása egy alosztályban.

fogadó (vevő) Kérelem célobjektuma.

ismeretség (ismeretségi viszony) Az az osztály, amely egy másik osztályra hivatkozik, ismeretségi viszonyban áll a másik osztállyal, vagyis *ismerőse* annak.

kérelem Az objektumok akkor hajtanak végre egy műveletet, amikor egy másik objektumtól a műveletnek megfelelő kérelem érkezik. A kérelem gyakran használt másik neve az **üzenet**.

keretrendszer Együttműködő osztályok halmaza, amelyekből egy bizonyos szoftvertípus számára újrahasznosítható terv készíthető. A keretrendszer a szoftver szerkezetét elvont osztályokra bontja, meghatározza azok felelősségi körét, illetve a közöttük levő kapcsolatokat, ezáltal útmutatást ad a program felépítéséhez. A fejlesztő a keretrendszert úgy szabhatja egy adott alkalmazásra, hogy a keretosztályokból példányokat és alosztályokat hoz létre.

konkrét osztály Elvont műveletekkel nem rendelkező osztály, amely példányosítható.

konstruktor (létrehozó függvény) A C++ nyelvben olyan művelet, amelynek meghívására új példányok létrehozásakor automatikusan sor kerül.

metaosztály Az osztályok a Smalltalk nyelvben objektumok. A metaosztály az osztályobjektumok osztálya.

mixin osztály vagy **bekeveredő (bekevert) osztály** Olyan osztály, amelyet más osztályokkal való, öröklésen keresztüli együttműködésre terveztek. A mixin osztályok általában elvontak.

művelet Az objektumokban tárolt adatokat csak az objektum műveletei érhetik el. E műveletek végrehajtására akkor kerül sor, amikor az objektumhoz kérelem érkezik. A C++-ban a műveleteket **tagfüggvényeknek** hívják, a Smalltalk a **metódus** kifejezést használja.

objektum Futásidejű egyed, amely mind az adatokat, mind az adatokon műveleteket végző eljárásokat tartalmazza.

objektumdiagram Diagram, amely egy adott objektumszerkezetet ábrázol futásidőben.

objektumhivatkozás Más objektumot azonosító érték.

objektum-összetétel Objektumok „*összeszerelése*", melynek célja összetettebb viselkedés kialakítása.

osztály Az osztály határozza meg egy objektum felületét és megvalósítását, valamint az osztály adja meg az objektum belső ábrázolását, illetve azokat a műveleteket, amelyeket az objektum végrehajthat.

osztálydiagram Osztályokat, azok belső szerkezetét és műveleteit, illetve az osztályok közötti statikus kapcsolatokat ábrázoló diagram.

osztályművelet Olyan művelet, amely nem egy önálló objektumra, hanem egy osztályra irányul. A C++-ban az osztályműveletek megfelelői a **statikus tagfüggvények**.

öröklés Olyan kapcsolat, amely egy egyedet egy másikhoz viszonyítva határoz meg. Az **osztályöröklés** egy vagy több szülőosztályból állít elő egy új osztályt, amely felületét és megvalósítását a szülőktől örökli. Az új osztályt **alosztálynak** vagy (a C++-ban) **származtatott osztálynak** hívjuk. Az osztályöröklés a **felületöröklést** és a **megvalósítás-öröklést** egyesíti: az előbbi egy új felületet ír le egy vagy több már létező felület segítségével, míg az utóbbi egy új megvalósítást, már létező megvalósítások alapján.

őstípus Szülőtípus, amelytől egy másik típus örököl.

összesítő kapcsolat Az összesítő objektum viszonya az őt alkotó részekhez. A példányok (az összesítő objektumok) részére ezt a kapcsolatot az osztály határozza meg.

összesítő objektum (aggregát objektum, aggregátum) Olyan objektum, amely alobjektumokból épül fel. Az alobjektumok az összesítő objektum *részei*, melyekért az aggregátum felelős.

paraméterezett típus Olyan típus, amely egyes összetevő típusait nem határozza meg pontosan; ezeket használatkor paraméterként kapja meg. A C++-ban a paraméterezett típusokat **sablonoknak** (template) hívják.

példányváltozó Egy objektum ábrázolásának egy részét meghatározó adat. A C++ az **adattag** kifejezést használja rá.

privát öröklés A C++ nyelvben olyan osztály öröklése, amelynek kizárólag a megvalósítására van szükség.

protokoll A felület fogalmának bővítése a feldolgozható kérelmek sorozatával.

szülőosztály Olyan osztály, amelytől egy másik osztály örököl. Szinonimái a **szuperosztály** (Smalltalk), az **alaposztály** (C++) és az **ősosztály**.

tervezési minta A tervezési minták rendszerezik, elnevezik, és megmagyarázzák az objektumközpontú rendszereken belül gyakran ismétlődő megoldásokat. Leírják a problémákat, a rájuk adott válaszokat, illetve hogy a megoldás mikor alkalmazható, milyen következményekkel és mellékhatásokkal kell számolnunk, valamint ötleteket adnak és példát mutatnak a megvalósításra is. A megoldás a feladathoz illeszkedő osztályok és objektumok általános érvényű elrendezése, amit a megvalósítással együtt az adott környezethez kell igazítani.

típus Egy adott felület neve.

többalakúság (polimorfizmus) Az összeegyeztethető felületű objektumok futásidőben egymással való helyettesítésének képessége.

B

Útmutató a jelölésekhez

A könyv során a fontosabb fogalmak illusztrálására diagramokat használtunk. Egyesek „informálisak" voltak, például egy párbeszédablak képernyőképének vagy egy objektumfa sematikus rajzának formáját öltötték. Maguk a tervezési minták azonban ennél formálisabb jelölésrendszert alkalmaznak az osztályok és objektumok közötti kapcsolatok bemutatására: ez a függelék ezt mutatja be részletesebben.

Három különböző diagramtípust használtunk:

1. Az **osztálydiagramok** az osztályokat, azok szerkezetét, valamint a közöttük levő statikus kapcsolatokat ábrázolják.
2. Az **objektumdiagramok** egy adott objektumszerkezetet mutatnak be futásidőben.
3. Az **együttműködési diagramok** (interakció-diagramok) a kérelmek útját mutatják az objektumok között.

Minden tervezési minta legalább egy osztálydiagramot tartalmaz, a többi alkalmazására akkor került sor, amikor a tárgyalás ezt megkívánta. Az osztály- és objektumdiagramok az OMT (Object Modeling Technique) modellen alapulnak [RBP+91, Rum94][1], az együttműködési diagramok az Objectory [JCJO92], illetve a Booch módszeren [Boo94].

B.1 Osztálydiagramok

A B.1 ábra az elvont és konkrét osztályok OMT jelölését mutatja. Az osztályokat egy doboz jelöli, amelynek felső részében az osztály vastag betűs neve olvasható. Az osztály kulcsműveletei az osztály neve alatt jelennek meg, ezek alatt pedig az esetleges példányváltozók.

[1] Az OMT az osztálydiagramokra is objektumdiagram néven hivatkozik, mi ezt kizárólag az objektumszerkezeteket ábrázoló diagramok részére tartottuk fenn.

A típusinformációk nem kötelezőek; mi a C++ programozók szokásait követjük, akik (a visszatérési típus jelzéséhez) a típus nevét kiteszik a művelet, a példányváltozó vagy a tényleges paraméter neve elé. A dőlt betű arra utal, hogy az adott osztály vagy művelet elvont.

Egyes tervezési mintákban segíthet, ha látjuk, hol hivatkoznak az ügyfélosztályok résztvevő osztályokra. Amikor a minta résztvevőként tartalmaz egy Ügyfél osztályt (vagyis annak a mintában felelősségi köre van), az Ügyfél szokványos osztályként jelenik meg. Ilyen például a Pehelysúlyú minta. Ha a mintának azonban nincsenek Ügyfél résztvevői (vagyis az ügyfeleknek nincs feladatuk a mintában), de feltüntetésük világosabbá teszi, mely résztvevők lépnek kapcsolatba ügyfelekkel, az Ügyfél osztályokat a B.1b ábrához hasonlóan szürke betűkkel jeleztük. Erre a Helyettes mintánál láthatunk példát. A szürke jelzés azt is egyértelművé teszi, hogy nem véletlenül maradt ki az Ügyfél a résztvevők tárgyalásából.

A B.1c ábra különböző osztályok közötti kapcsolatokat mutat. Az osztályöröklés OMT jelölése az alosztályt (az ábrán VonalAlakzat) és szülőosztályát (Alakzat) összekötő háromszög. A része, illetve összesítés kapcsolatot ábrázoló objektumhivatkozást rombusz alapú nyílhegyű vonal jelzi, ami az összesített osztály (például az Alakzat) felé mutat. Ha a nyílhegyű vonal végén nincs rombusz, akkor ismeretségi viszonyról van szó (például a VonalAlakzat egy Szin objektumra hivatkozik, amelyen a különböző alakzatok osztozkodhatnak). A hivatkozás nevét a kezdőpont közelében néhol feltüntettük, hogy megkülönböztessük más hivatkozásoktól.[2]

Egy másik fontos dolog annak jelzése, mely osztályok példányosítanak más osztályokat. Ezt szaggatott nyílhegyű vonallal ábrázoltuk (az OMT nem támogatja). Ezt hívjuk „létrehozza" kapcsolatnak; a nyíl a példányosítás céljául szolgáló osztályra mutat. A B.1c ábrán a LétrehozóEszköz VonalAlakzat objektumokat hoz létre.

Az OMT jelölése az „egynél több" fogalmára a teli kör. Ha ilyen kört látunk egy hivatkozás előtt, az azt jelenti, hogy több objektumra hivatkozunk, illetve több objektumot összesítünk. A B.1c ábra azt mutatja, hogy a Rajz több Alakzat típusú objektumot összesít.

Végezetül, az OMT-t álkódos (pszeudokódos) jelzésekkel egészítettük ki, hogy felvázolhassuk a műveletek megvalósítását. A B.1d ábra a Rajz osztály Rajzol műveletének álkódját tartalmazza.

---

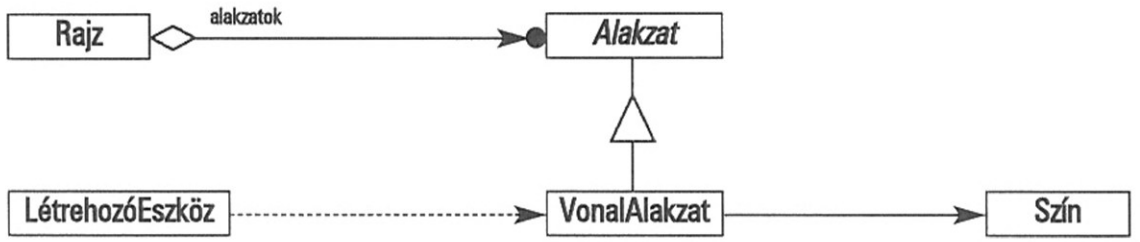
[2] Az OMT az osztályok között „asszociációkat" is meghatároz, amelyeket az osztályok dobozai között egyszerű vonalakkal jelez (az asszociációk kétirányúak). Bár elemzés közben hasznosak lehetnek, úgy éreztük, az asszociációk túlságosan elvontak ahhoz, hogy a tervezési mintákban kapcsolatokat fejezzünk ki velük, hiszen a tervezés során úgyis le kell majd őket fordítanunk objektumhivatkozásokra vagy mutatókra. Az objektumhivatkozások emellett befelé irányulnak, így jobban megfelelnek a minket érdeklő kapcsolatok számára. (A Rajz például ismeri az Alakzat objektumokat, az Alakzatok azonban nem ismerik az őket tartalmazó Rajzot. Ezt a viszonyt nem lehet pusztán asszociációkkal kifejezni.)

B.2 Objektumdiagramok

Az objektumdiagramok kizárólag példányokat ábrázolnak; a tervezési minta objektumainak pillanatfelvételét. Az objektumnak az „egy *Valami*" nevet adtuk, ahol a „*Valami*" az objektum osztálya. Az objektumok jele a könyvben (némileg eltérően a szabványos OMT-től) a lekerekített sarkú „doboz" (négyszög), amelyben az objektum nevét vonal választja el az esetleges objektumhivatkozásoktól. A hivatkozott objektumokra nyilak mutatnak. Az objektumdiagramra a B.2 ábrán láthatunk példát.

(a) Elvont és konkrét osztályok

(b) Résztvevő Ügyfél osztály (balra) és rejtett Ügyfél osztály (jobbra)

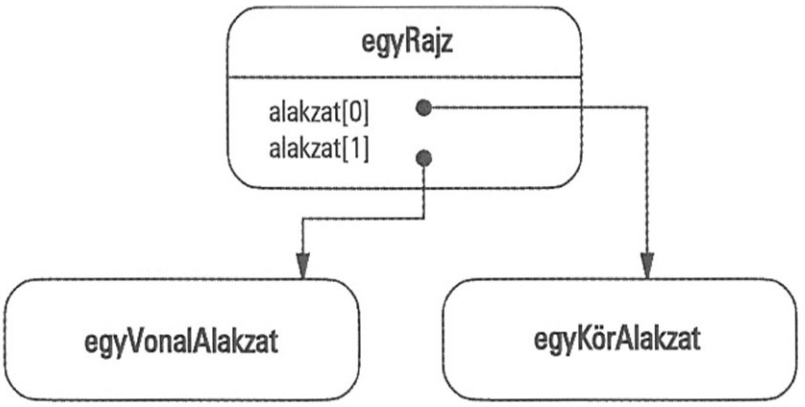


(c) Osztálykapcsolatok

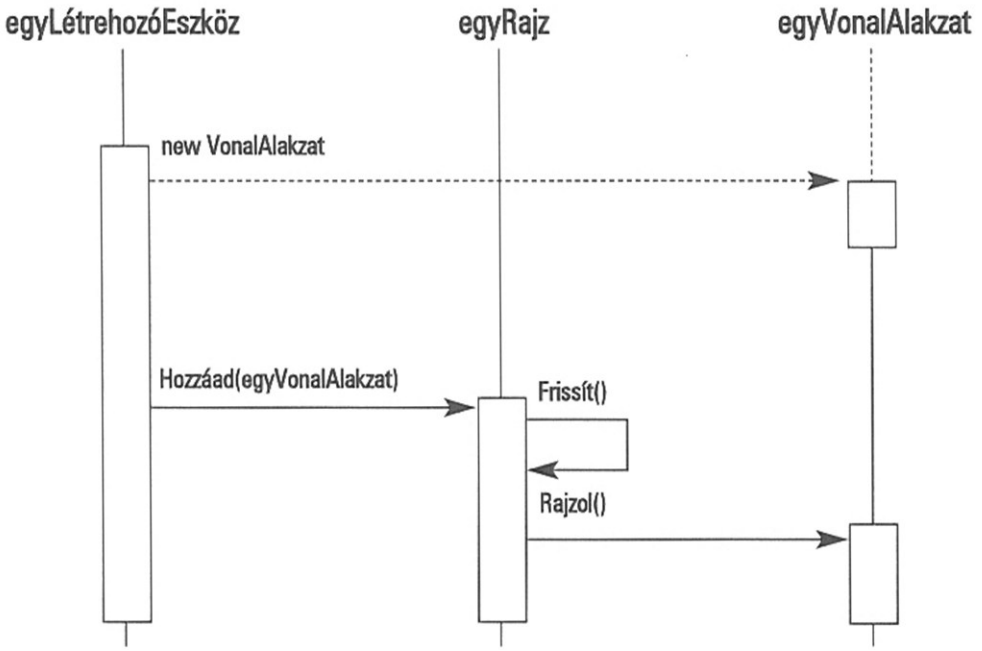
(d) Álkódos jelölés

B.1 ábra

Az osztálydiagramok jelölései.



B.2 ábra
Az objektumdiagramok jelölései.



B.3 ábra
Az együttműködési diagramok jelölései.

B.3 Együttműködési diagramok

Az együttműködési diagramok a kérelmek végrehajtásának sorrendjét mutatják az objektumok között. A B.3 ábrán levő diagram azt mutatja, hogyan adunk egy alakzatot egy rajzhoz.

Az együttműködési diagramon az idő fentről lefelé „folyik"; egy-egy objektum élettartamát folytonos függőleges vonal jelzi. Az objektumok elnevezésére ugyanazok a szabályok vonatkoznak, mint az objektumdiagramoknál, vagyis az osztály neve elé az „egy" kerül (pl. egyAlakzat). Ha egy objektum példányosítására csak a diagram által rögzített időpont után kerül sor, ezt szaggatott függőleges vonallal jelezzük, ami a létrehozási pontig tart.

Azt, hogy egy objektum aktív, vagyis kérelmet szolgál ki, a rövidebb oldalára állított („függőleges") téglalappal jelöljük. A művelet más objektumokat is felkérhet, ezeket a kérelmeket a fogadó objektumra mutató vízszintes nyilak mutatják; a kérelem nevét a nyíl felett tüntetjük fel. Az objektum létrehozására irányuló kérelmeket szaggatott nyílhegyű vonallal jelezzük, az olyanokat pedig, amelyek magára a küldő objektumra vonatkoznak, visszakanyarodó nyíllal.

A B.3 ábra azt mutatja, hogy az első kérelem az egyLétrehozóEszköz-től érkezik, és az egyVonalAlakzat létrehozására irányul. Később az egyVonalAlakzat „Hozzáad"-ódik az egyRajz-hoz, e művelet pedig az egyRajz-ot arra kéri, hogy saját magának küldjön egy Frissít kérelmet. Megfigyelhetjük azt is, hogy az egyRajz a Rajzol kérelmet a Frissít művelet részeként küldi el az egyVonalAlakzat-nak.

C

Alaposztályok

Ebben a függelékben azokat az alaposztályokat ismertetjük, amelyeket az egyes tervezési mintákhoz mellékelt C++ példakódokban használtunk. Szándékosan egyszerűek és számukat is szándékosan tartottuk alacsonyan. A következő osztályokról van szó:

- `List` – objektumok rendezett listája.
- `Iterator` – egy összesítő objektum (aggregátum) objektumainak sorrendben történő elérésére szolgáló felület.
- `ListIterator` – bejáró a `List` bejárásához.
- `Point` – kétdimenziós pont.
- `Rect` – tengelyekhez igazított téglalap.

A C++ egyes újabb szabványos típusait nem minden fordítóprogram ismeri. Amennyiben a `bool` esetében ez a helyzet, határozzuk meg ezt a típust magunk:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

C.1 List

A `List` osztálysablon alapszintű tárolót biztosít objektumok rendezett listájának tárolására. Az elemeket érték szerint tárolja, ami azt jelenti, hogy a beépített típusokkal ugyanúgy működik, mint az osztálypéldányokkal. A `List<int>` például egészek (`int`) listáját adja meg. Mindazonáltal a legtöbb minta a `List`-et arra használja, hogy objektumokat címző mutatókat tároljon – ilyen például a `List<Glyph*>` –, ezért a `List` heterogén listákhoz nem alkalmazható.

A kényelem kedvéért a `List` a veremműveletekhez szinonimákat is biztosít, így a `List`-et vermekhez használó kódban nem kell új osztályt meghatároznunk, ami áttekinthetőbbé teszi a programot.

```
template <class Item>
class List {
public:
   List(long size = DEFAULT_LIST_CAPACITY);
    List(List&);
    ~List();
    List& operator=(const List&);

    long Count() const;
    Item& Get(long index) const;
    Item& First() const;
    Item& Last() const;
    bool Includes(const Item&) const;

    void Append(const Item&);
    void Prepend(const Item&);

    void Remove(const Item&);
    void RemoveLast();
    void RemoveFirst();
    void RemoveAll();

    Item& Top() const;
    void Push(const Item&);
    Item& Pop();
};
```

A következőkben a fenti műveleteket részletesebben is bemutatjuk.

Létrehozás, megsemmisítés, előkészítés és értékadás

```
List(long size)
```
Előkészíti (inicializálja) a listát. A `size` (méret) paraméter az elemek kezdeti számára utal.

```
List(List&)
```
Felülbírálja az alapértelmezett másoló konstruktort az adattagok megfelelő előkészítéséhez.

```
~List()
```
Felszabadítja a lista belső adatszerkezeteit, de a lista elemeit *nem*. Az osztályból nem származtathatók alosztályok, ezért a destruktor nem virtuális.

```
List& operator=(const List&)
```
Megvalósítja az értékadó műveletet az adattagoknak megfelelő értékadáshoz.

Elérés

Ezek a műveletek a lista elemeinek alapszintű elérését biztosítják.

```
long Count() const
```
A listában levő objektumok számát adja vissza.

```
Item& Get(long index) const
```
A megadott sorszámú (indexű) objektumot adja vissza.

```
Item& First() const
```
A lista első objektumát adja vissza.

```
Item& Last() const
```
A lista utolsó objektumát adja vissza.

Hozzáadás

```
void Append(const Item&)
```
Az argumentumot a listához adja, annak utolsó elemeként (append = hozzáfűz).

```
void Prepend(const Item&)
```
Az argumentumot a listához adja, annak első elemeként (prepend = eléfűz).

Eltávolítás

```
void Remove(const Item&)
```
A megadott elemet eltávolítja a listából (remove = eltávolít). A művelet megköveteli, hogy a listában levő elemek típusa támogassa az == összehasonlító operátort.

```
void RemoveFirst()
```
Eltávolítja az első elemet a listából.

```
void RemoveLast()
```
Eltávolítja az utolsó elemet a listából.

```
void RemoveAll()
```
Minden elemet eltávolít a listából.

Veremfelület

```
Item& Top() const
```
A legfelső elemet adja vissza (amikor a listát veremként kezeljük).

```
void Push(const Item&)
```
Az elemet a verem tetejére helyezi.

```
Item& Pop()
```
Az elemet leveszi a verem tetejéről.

C.2 Iterator

Az `Iterator` az összesítő objektumok bejárási felületét meghatározó elvont osztály.

```
template <class Item>
class Iterator {
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

A műveletek szerepe a következő:

`virtual void First()`
A bejárót az aggregátum első objektumára állítja.

`virtual void Next()`
A bejárót a sorozat következő objektumára állítja.

`virtual bool IsDone() const`
Igazat (`true`-t) ad vissza, ha a sorozatban már nincs több objektum.

`virtual Item CurrentItem() const`
A sorozat aktuális pozícióján álló objektumot adja vissza.

C.3 ListIterator

A `ListIterator` az `Iterator` felület megvalósítása listaobjektumok bejárásához. Konstruktora argumentumként a bejárandó listát várja.

```
template <class Item>
class ListIterator : public Iterator<Item> {
public:
   ListIterator(const List<Item>* aList);

    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
};
```

C.4 Point

A `Point` egy pontot jelképez egy kétdimenziós derékszögű koordináta-rendszerben. Alapszintű vektorszámítási képességekkel rendelkezik. Az egyes pontok koordinátáit a következőképpen határozza meg:

```
typedef float Coord;
```

A `Point` műveletei önmagukért beszélnek.

```
class Point {
public:
    static const Point Zero;
```

```
    Point(Coord x = 0.0 Coord y = 0.0);

    Coord X() const; void X(Coord x);
    Coord Y() const; void Y(Coord y);

    friend Point operator+(const Point&, const Point&);
    friend Point operator-(const Point&, const Point&);
    friend Point operator*(const Point&, const Point&);
    friend Point operator/(const Point&, const Point&);

    Point& operator+=(const Point&);
    Point& operator-=(const Point&);
    Point& operator*=(const Point&);
    Point& operator/=(const Point&);

    Point operator-();

    friend bool operator==(const Point&, const Point&);
    friend bool operator!=(const Point&, const Point&);

    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);

};
```

A `Zero` statikus tag jelentése `Point(0, 0)`.

C.5 Rect

A `Rect` egy tengelyekhez igazított téglalapot jelképez, amelyet kezdőpontja és kiterjedése (`extent`, vagyis a szélessége és a magassága, `Width` és `Height`) határoz meg. A `Rect` műveletei sem igényelnek külön magyarázatot.

```
class Rect {
public:
    static const Rect Zero;

    Rect(Coord x, Coord y, Coord w, Coord h);
    Rect(const Point& origin, const Point& extent);

    Coord Width() const; void Width(Coord);
    Coord Height() const; void Height(Coord);
    Coord Left() const; void Left(Coord);
    Coord Bottom() const; void Bottom(Coord);

    Point& Origin() const; void Origin(const Point&);
    Point& Extent() const; void Extent(const Point&);
```

```
    void MoveTo(const Point&);
    void MoveBy(const Point&);

    bool IsEmpty() const;
    bool Contains(const Point&) const;
};
```

A `Zero` statikus tag a következő négyszöggel egyenértékű:

```
Rect(Point(0, 0), Point(0, 0));
```

Irodalomjegyzék

[Add94] Addison-Wesley, Reading, MA. *NEXTSTEP General Reference* 3. kiadás, 1. és 2. kötet, 1994

[AG90] D.B. Anderson és S. Gossain: Hierarchy evolution and the software lifecycle. *TOOLS '90 Conference Proceedings*, 41–50. oldal, Párizs, 1990. június, Prentice Hall.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King és Shlomo Angel: *A Pattern Language*. Oxford University Press, New York, 1977.

[App89] Apple Computer, Inc., Cupertino, CA. *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.

[App92] Apple Computer, Inc., Cupertino, CA. Dylan. *An object-oriented dynamic language*, 1992.

[Arv91] James Arvo: *Graphics Gems II*. Academic Press, Boston, MA, 1991.

[AS85] B. Adelson és E. Soloway: The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, 1985.

[BE93] Andreas Birrer és Thomas Eggenschwiler: Frameworks in the financial engineering domain: An experience report. *European Conference on Object-Oriented Programming*, 21–35. oldal, Kaiserslautern, Németország, 1993. július, Springer-Verlag.

[BJ94] Kent Beck és Ralph Johnson: Patterns generate architectures. *European Conference on Object-Oriented Programming*, 139–149. oldal, Bologna, Olaszország, 1994. július, Springer-Verlag.

[Boo94] Grady Booch: *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1994. Második kiadás.

[Bor81] A. Borning: The programming language aspects of ThingLab – a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):343–387, 1981. október.

[Bor94] Borland International, Inc., Scotts Valley, CA. *A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5*, 1994.

[BV90] Grady Booch és Michael Vilot: The design of the C++ Booch components. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 1–11. oldal, Ottawa, Kanada, 1990. október, ACM Press.

[Cal93] Paul R. Calder: *Building User Interfaces with Lightweight Objects.* PhD szakdolgozat, Stanford University, 1993.

[Car89] J. Carolan: Construction bullet-proof classes. *Proceedings C++ at Work '89*, SIGS Publications, 1989.

[Car92] Tom Cargill: *C++ Programming Style.* Addison-Wesley, Reading, MA, 1992.

[CIRM93] Roy H. Campbell, Nayeem Islam, David Raila és Peter Madeany: Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, 1993. szeptember.

[CL90] Paul R. Calder és Mark A. Linton: Glyphs: Flyweight objects for user interfaces. *ACM User Interface Software Technologies Conference*, 92–101. oldal, Snowbird, UT, 1990. október.

[CL92] Paul R. Calder és Mark A. Linton: The object-oriented implementation of a document editor. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 154–165. oldal, Vancouver, British Columbia, Kanada, 1992. október, ACM Press.

[Coa92] Peter Coad: Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992. szeptember.

[Coo92] William R. Cook: Interfaces and specifications for the Smalltalk-80 collection classes. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 1–15. oldal, Vancouver, British Columbia, Kanada, 1992. október, ACM Press.

[Cop92] James O. Coplien: *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[Cur89] Bill Curtis: Cognitive issues in reusing software artifacts. Ted J. Biggerstaff és Alan J. Perlis (szerk.): *Software Reusability, Volume II: Applications and Experience*, 269–287. oldal, Addison-Wesley, Reading, MA, 1989.

[dCLF93] Dennis de Champeaux, Doug Lea és Penelope Faure: *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.

[Deu89] L. Peter Deutsch: Design reuse and frameworks in the Smalltalk-80 system. Ted J. Biggerstaff és Alan J. Perlis (szerk.): *Software Reusability, Volume II: Applications and Experience*, 57–71. oldal, Addison-Wesley, Reading, MA, 1989.

[Ede92] D.R. Edelson: Smart pointers: They're smart, but they're not pointers. *Proceedings of the 1992 USENIX C++ Conference*, 1–19. oldal, Portland, OR, 1992. augusztus, USENIX Association.

[EG92] Thomas Eggenschwiler és Erich Gamma: The ET++SwapsManager: Using object technology in the financial engineering domain. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 166–178. oldal, Vancouver, British Columbia, Kanada, 1992. október, ACM Press.

[ES90] Margaret A. Ellis és Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[Foo92] Brian Foote: A fractal model of the lifecycles of reusable objects. *OOPSLA '92 Workshop on Reuse*, 1992. október, Vancouver, British Columbia, Kanada.

[GA89] S. Gossain és D.B. Anderson: Designing a class hierarchy for domain representation and reusability. *TOOLS '89 Conference Proceedings*, 201–210. oldal, CNIT Paris-La Defense, Franciaország, 1989. november, Prentice Hall.

[Gam91] Erich Gamma: *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (német nyelven). PhD szakdolgozat, Zürichi Egyetem, Informatikai Intézet, 1991.

[Gam92] Erich Gamma: *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (német nyelven), Springer-Verlag, Berlin, 1992.

[Gla90] Andrew Glassner: *Graphics Gems*, Academic Press, Boston, MA, 1990.

[GM92] M. Graham és E. Mettala: The Domain-Specific Software Architecture Program. *Proceedings of DARPA Software Technology Conference, 1992*, 204–210. oldal, 1992. április. Újraközölve: *CrossTalk, The Journal of Defense Software Engineering*, 19–21. és 32. oldal, 1992. október.

[GR83] Adele J. Goldberg és David Robson: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[HHMV92] Richard Helm, Tien Huynh, Kim Marriott és John Vlissides: An object-oriented architecture for constraint-based graphical editing. *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, 1–22. oldal, Champéry, Svájc, 1992. október, illetve IBM Research Division Technical Report RC 18524 (79392).

[HO87] Daniel C. Halbert és Patrick D. O'Brien: Object-oriented development. *IEEE Software*, 4(5):71–79, 1987. szeptember.

[ION94] IONA Technologies, Ltd., Dublin, Írország. *Programmer's Guide for Orbix, Version 1.2*, 1994.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrick Jonsson és Gunnar Overgaard: *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley, Wokingham, Anglia, 1992.

[JF88] Ralph E. Johnson és Brian Foote: Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. június-július.

[JML92] Ralph E. Johnson, Carl McConnell és J. Michael Lake: The RTL system: A framework for code optimization. Robert Giegerich és Susan L. Graham (szerk.): *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, 255–274. oldal, Dagstuhl, Németország, 1992, Springer-Verlag.

[Joh92] Ralph E. Johnson: Documenting frameworks using patterns. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 63–76. oldal, Vancouver, British Columbia, Kanada, 1992. október, ACM Press.

[JZ91] Ralph E. Johnson és Jonathan Zweig: Delegation in C++. *Journal of Object-Oriented Programming*, 4(11):22–35, 1991. november.

[Kir92] David Kirk: *Graphics Gems III*, Harcourt, Brace, Jovanovich, Boston, MA, 1992.

[Knu73] Donald E. Knuth: *The Art of Computer Programming*, 1., 2. és 3. kötet, Addison-Wesley, Reading, MA, 1973.

[Knu84] Donald E. Knuth: *The TeXbook*, Addison-Wesley, Reading, MA, 1984.

[Kof93] Thomas Kofler: Robust iterators in ET++. *Structured Programming*, 14:62–85, 1993. március.

[KP88] Glenn E. Krasner és Stephen T. Pope: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988. augusztus-szeptember.

[LaL94] Wilf LaLonde: *Discovering Smalltalk*, Benjamin/Cummings, Redwood City, CA, 1994.

[LCI+92] Mark Linton, Paul Calder, John Interrante, Steven Tang és John Vlissides: *InterViews Reference Manual*, CSL, Stanford University, 3.1. kiadás, 1992.

[Lea88] Doug Lea: libg++, the GNU C++ library. *Proceedings of the 1988 USENIX C++ Conference*, 243–256. oldal, Denver, CO, 1988. október, USENIX Association.

[LG86] Barbara Liskov és John Guttag: *Abstraction and Specification in Program Development*, McGraw-Hill, New York, 1986.

[Lie85] Henry Lieberman: There's more to menu systems than meets the screen. *SIGGRAPH Computer Graphics*, 181–189. oldal, San Francisco, CA, 1985. július.

[Lie86] Henry Lieberman: Using prototypical objects to implement shared behavior in object-oriented systems. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 214–223. oldal, Portland, OR, 1986. november.

[Lin92] Mark A. Linton: Encapsulating a C++ library. *Proceedings of the 1992 USENIX C++ Conference*, 57–66. oldal, Portland, OR, 1992. augusztus, ACM Press.

[LP93] Mark Linton és Chuck Price: Building distributed user interfaces with Fresco. *Proceedings of the 7th X Technical Conference*, 77–87. oldal, Boston, MA, 1993. január.

[LR93] Daniel C. Lynch és Marshall T. Rose: *Internet System Handbook*, Addison-Wesley, Reading, MA, 1993.

[LVC89] Mark A. Linton, John M. Vlissides és Paul R. Calder: Composing user interfaces with InterViews. *Computer*, 22(2):8–22, 1989. február.

[Mar91] Bruce Martin: The separation of interface and implementation in C++. *Proceedings of the 1991 USENIX C++ Conference*, 51–63. oldal, Washington D.C., 1991. április, USENIX Association.

[McC87] Paul McCullogh: Transparent forwarding: First steps. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 331–341. oldal, Orlando, FL, 1987. október, ACM Press.

[Mey88] Bertrand Meyer: *Object-Oriented Software Construction*. Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Mur93] Robert B. Murray: *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA 1993.

[OJ90] William F. Opdyke és Ralph E. Johnson: Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *SOOPPA Conference Proceedings*, 145–161. oldal, Marist College, Poughkeepsie, NY, 1990. szeptember, ACM Press.

[OJ93] William F. Opdyke és Ralph E. Johnson: Creating abstract superclasses by refactoring. *Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93)*, 66–73. oldal, Indianapolis, IN, 1993. február.

[P+88] Andrew J. Palay és mások: The Andrew Toolkit: An overview. *Proceedings of the 1988 Winter USENIX Technical Conference*, 9–21. oldal, Dallas, TX, 1988. február, USENIX Association.

[Par90] ParPlace Systems, Mountain View, CA. *ObjectWorks\Smalltalk Release 4 Users Guide*, 1990.

[Pas86] Geoffrey A. Pascoe: Encapsulators: A new software paradigm in Smalltalk-80. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 341–346. oldal, Portland, OR, 1986. október, ACM Press.

[Pug90] William Pugh: Skiplists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. június.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy és William Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rum94] James Rumbaugh: The life of an object model: How the object model changes during development. *Journal of Object-Oriented Programming*, 7(1):24–32, 1994. március-április.

[SE84] Elliot Soloway és Kate Ehrlich: Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984. szeptember.

[Sha90] Yen-Ping Shan: MoDE: A UIMS for Smalltalk. *ACM OOPSLA/ECOOP '90 Conference Proceedings*, 258–268. oldal, Ottawa, Ontario, Kanada, 1990. október, ACM Press.

[Sny86] Alan Snyder: Encapsulation and inheritance in object-oriented languages. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 38–45. oldal, Portland, OR, 1986. november, ACM Press.

[SS86] James C. Spohrer és Elliot Soloway: Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986. július.

[SS94] Douglas C. Schmidt és Tatsuya Suda: The Service Configurator Framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons. *Proceedings of the Second International Workshop on Configurable Distributed Systems*, 190–201. oldal, Pittsburgh, PA, 1994. március, IEEE Computer Society.

[Str91] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1991. Második kiadás. (Magyarul: *A C++ programozási nyelv*, 2002, Kiskapu Kiadó.)

[Str93] Paul S. Strauss: IRIS Inventor, a 3D graphics toolkit. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 192–200. oldal, Washington D.C., 1993. szeptember, ACM Press.

[Str94] Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994.

[Sut63] I.E. Sutherland: *Sketchpad: A Man-Machine Graphical Communication System*. PhD szakdolgozat, MIT, 1963.

[Swe85] Richard E. Sweet: The Mesa programming environment. *SIGPLAN Notices*, 20(7):216–229, 1985. július.

[Sym93a] Symantec Corporation, Cupertino, CA. *Bedrock Developer's Architecture Kit*, 1993.

[Sym93b] Symantec Corporation, Cupertino, CA. *THINK Class Library Guide*, 1993.

[Sza92] Duane Szafron: SPECTalk: An object-oriented data specification language. *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, 123–138. oldal, Santa Barbara, CA, 1992. augusztus, Prentice Hall.

[US87] David Ungar és Randall B. Smith: Self: The power of simplicity. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, 227–242. oldal, Orlando, FL, 1987. október, ACM Press.

[VL88] John M. Vlissides és Mark A. Linton: Applying object-oriented design to structured graphics. *Proceedings of the 1988 USENIX C++ Conference*, 81–94. oldal, Denver, CO, 1988. október, USENIX Association.

[VL90] John M. Vlissides és Mark A. Linton: Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, 1990. július.

[WBJ90] Rebecca Wirfs-Brock és Ralph E. Johnson: A survey of current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson és Lauren Wiener: *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[WGM88] André Weinand, Erich Gamma és Rudolf Marty: ET++ – An object-oriented application framework in C++. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, 46–57. oldal, San Diego, CA, 1988. szeptember, ACM Press.

Tárgymutató

A, Á

B

C, Cs

D

E, É

F

G, Gy

H

J

K

L

M

N, Ny

O, Ó, Ö, Ő

P

Q

R

S, Sz

T, Ty

U, Ú, Ü, Ű

V

W

X

Y

Z, Zs

E kötetben az objektumközpontú szoftvertervezésben szerzett hatalmas tapasztalataikkal felvértezve négy elismert tervező mutatja be egyszerű, de nagyszerű megoldásait az általánosan felbukkanó tervezési problémákra. A korábban még le nem írt 23 tervezési minta lehetővé teszi, hogy a tervezők rugalmasabb, elegánsabb – és ami a legfontosabb –, újrahasznosítható programterveket készíthessenek, anélkül, hogy a megoldásokat maguknak kellene felfedezniük.

A szerzők először bemutatják a létező mintákat, illetve hogy ezek miként segítenek bennünket az objektumközpontú programok fejlesztésében, majd rendszerezve nevet adnak az objektumközpontú rendszerekben vissza-visszatérő mintáknak, elmagyarázzák és értékelik azokat. A *Programtervezési minták* segítségével megtanuljuk, hogyan illeszkednek ezek a minták a szoftverfejlesztés folyamatába, és hogyan oldhatók meg velük a leghatékonyabban saját egyéni tervezési gondjaink.

Minden mintánál leírják, milyen körülmények között alkalmazható, milyen más tervezési megkötéseket kell figyelembe venni, illetve hogy az adott minta nagyobb terv részeként való felhasználásánál milyen következményekkel és mellékhatásokkal kell számolnunk. Minden minta létező rendszeren, a valós életből vett példákon alapul. Mindegyikhez tartozik kód is, amely bemutatja, hogyan valósítható meg a minta az olyan objektumközpontú nyelveken, mint a C++ vagy a Smalltalk.

A szerzők nemzetközileg elismert szakemberek az objektumközpontú programozás területén. Dr. Erich Gamma a svájci Zürichben az Object Techology International szoftvertechnológiai központjának technikai igazgatója. Dr. Richard Helm az ausztráliai Sydneyben dolgozik, mint az IBM Consulting Group Object Technology Practice Group csoportjának tagja. Dr. Ralph Johnson az illionoisi egyetemen oktat, a számítógép-tudományok tanszékén. Dr. John Vlissides az IBM Thomas J. Watson kutatóközpontjában folytatja kutatásait a New York állambeli Hawthorne-ban.

KISKAPU

Addison-Wesley

Kategória: programozás, elmélet
Felhasználói szint: haladó

ISBN szám: 963 9301 77 9
Ára: 5980 Ft

ISBN: 963 9301 77 9
9 789639 301771