Tervezés és elemzés elmélete

A 28., 29., 31., 33., 34. fejezetek a TÁMOP-4.2.1.B-09-1-KMR-2010-0003 támogatásával készültek.

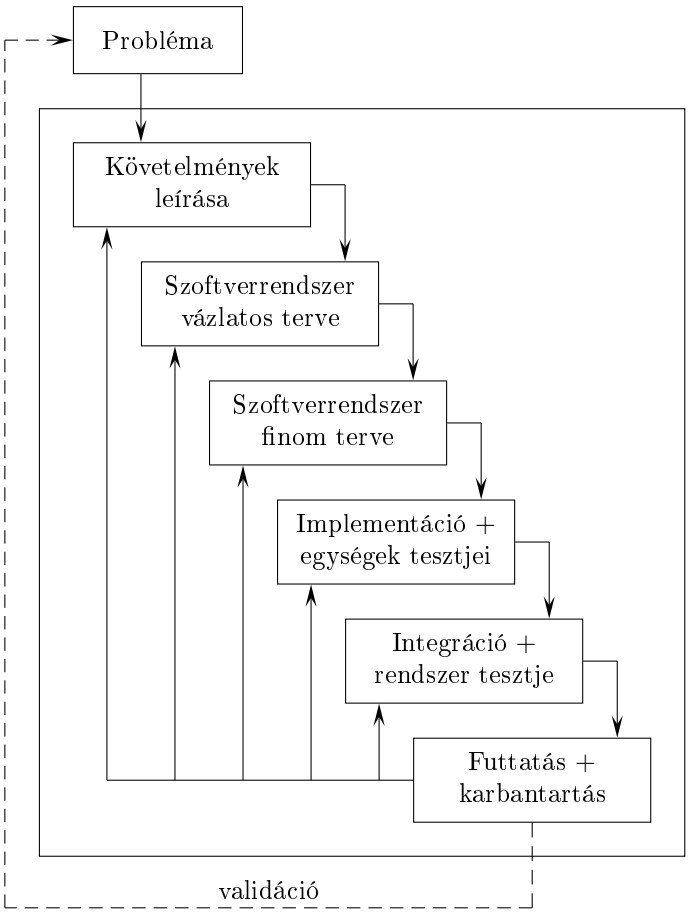
Tartalomjegyzék

1. Szoftverfejlesztési modellek

A gyakorlatban kialakultak különböző modellek, amelyek a szoftverfejlesztés fázisainak kapcsolatait adják meg.

1.1. Vízesés modell

A modell szerinti kapcsolatokat mutatja az 1.1 ábra. Az ábráról jól leolvasható a fejlesztés menete. Az egyes fázisok egymást követik, a módosítások a futtatási eredmények ismeretében történnek. Egy bizonyos fázisban elvégzett módosítás az összes rákövetkező fázist is



1.1. ábra. A vízesés modell

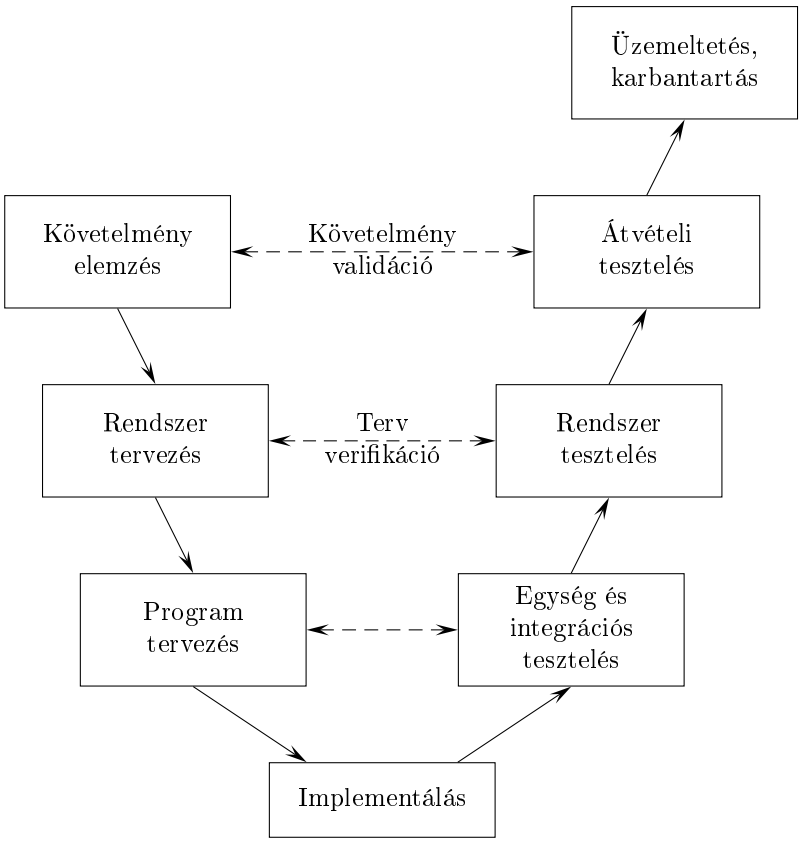
érinti. Ezt a modellt a gyakorlat szülte.

A vízesés modell hátrányai:

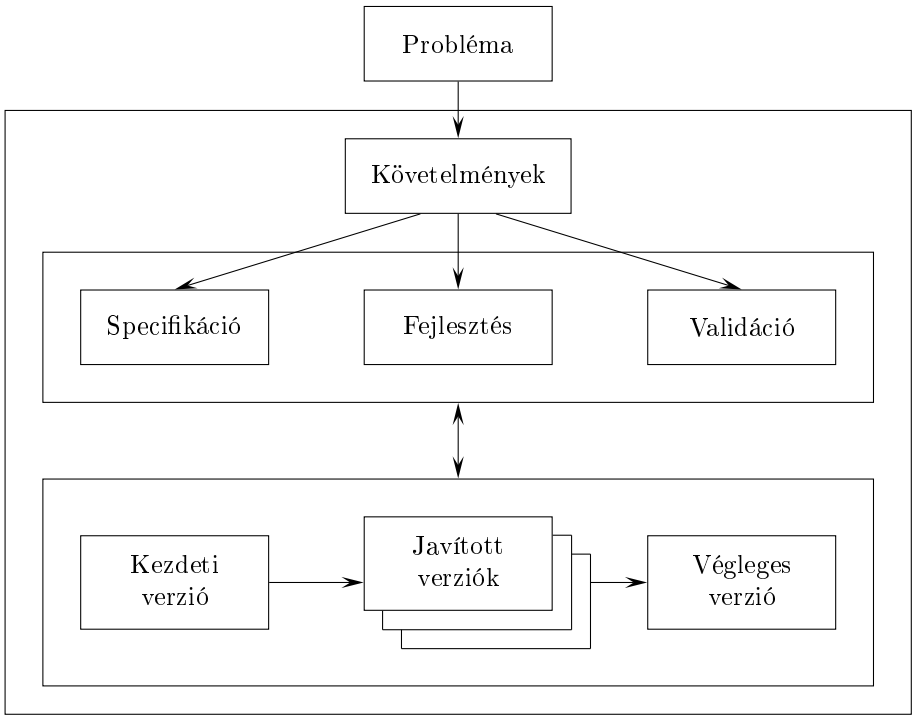
- A projekt munkájának megszervezése rendkívül nehézkes. (Párhuzamos munka, ellenőrzési pontok elhelyezése.)
- Új szolgáltatások utólagos bevezetése szinte minden fázison módosítást kíván meg.
- A validáció az egész életciklus megismétlését követelheti meg.

1.2. V-modell

A modell a vízesés modell egy módosított változata, amelyet a német védelmi minisztérium fejlesztett ki, és tett a német hadsereg szoftverfejlesztési szabványává 1992-ben. A módosítás lényege, hogy az egyes fázisok eredményét korábban ellenőrzik, így a visszacsatolások nagysága és hatása csökkenthető (1.2. ábra).



1.2. ábra. A V-modell



1.3. ábra. Az evolúciós modell

1.3. Evolúciós modell

Az evolúciós modellt szemlélteti az 1.3 ábra. A modell lényeges vonása, hogy a megoldás közelítő verzióinak, prototípusainak sorozatát állítjuk egymás után elő, és így haladunk lépésenként egészen a végleges megoldásig. Ennek során egy verzió elkészítésekor a specifikáció, a fejlesztés és a validáció párhuzamosan történik.

A programrendszerek fejlesztése során rendszerint nem áll rendelkezésünkre egy teljes követelményleírás. Nagyon sok részlet csak a felhasználó fejében létezik, így tehát nem is tudunk egy teljes és ellentmondásmentes specifikációt készíteni. Ennek alapján vagyunk kénytelenek megkezdeni a fejlesztést. Előállítjuk tehát a megoldás egy első verzióját. Ehhez elkészítjük a felhasználói felületet szimuláló prototípust. Ezt egyeztetjük a felhasználóval, és az elemzés alapján döntünk a következő verzió előállításáról. A verziók sorozatának fejlesztésével közelítünk a végső megoldáshoz.

A modell fő erénye a működés közben tanulmányozható prototípusok sorozatának megléte. Különösen előnyös ez olyan feladatok esetén, amikor nehézséget okoz a részletes specifikáció elkészítése.

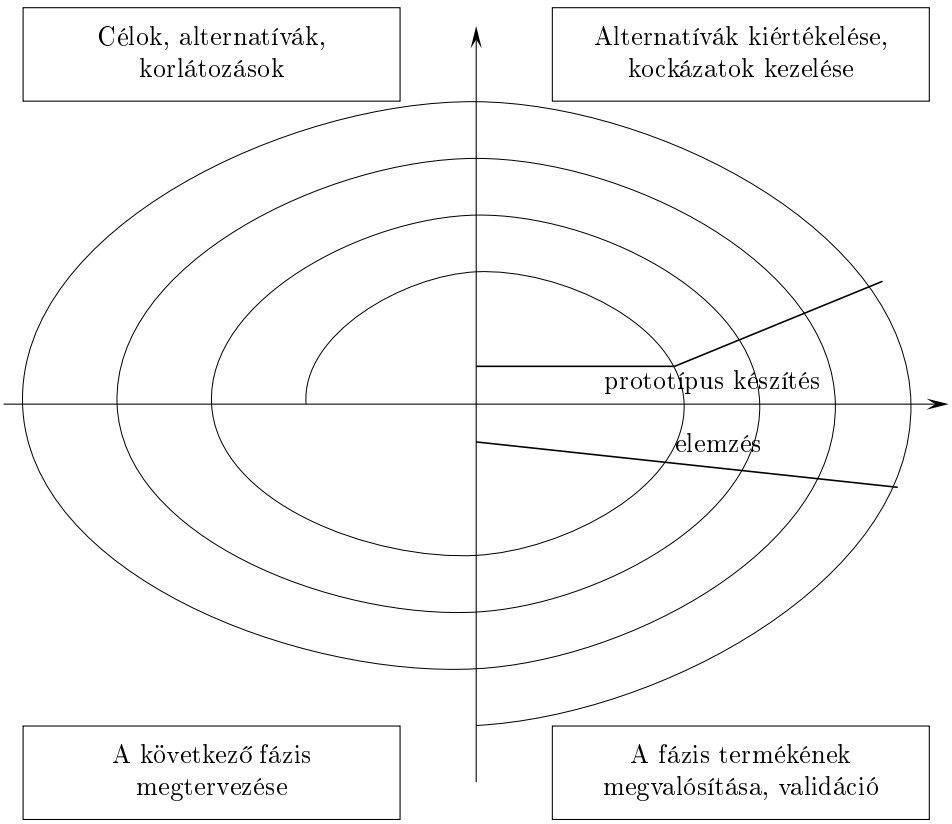
A modell hátrányai:

- Elsősorban csak rövidebb idő alatt megvalósítható rendszerek esetén ajánlható.
- A rendszer utólagos módosítása, bővítése nehézkes.
- Nehéz a projekt áttekintése.
- A gyors fejlesztés rendszerint a dokumentáltság rovására megy.

1.4. Boehm-féle spirális modell

Boehm 1988-ban javasolta programrendszerek kidolgozására ezt a modellt. A javaslat lényege az a felismerés, hogy a programok rendszerint iterációkon keresztül nyerik el végső formájukat. Javaslata szerint ezek az iterációk spirálisba olvaszthatók össze. Az iteráció a spirális egy fázisával modellezhető, amely négy szakaszra bontható (1.4 ábra). Ezek a következők:

1. Az adott iterációs lépés során elérendő célok és a megoldást korlátozó feltételek definiálása. A megoldás lehetséges útjainak, az alternatíváknak a meghatározása.
2. A különböző megoldások során felmerülő esetleges kockázati tényezők elemzése, stratégiák kidolgozása a kockázati tényezők hatásának csökkentésére, fellépésük elkerülésére; szükség szerint prototípusok készítése, és azok eredményeinek értékelése.
3. Az előző pontban javasolt elemzés alapján az iterációs lépés feladatának megoldása, a megoldással szemben támasztott követelmények teljesülésének igazolása, validáció.
4. A következő iterációs lépés megtervezése.



1.4. ábra. Boehm-féle spirális modell vázlata

A modell leírása során felhasznált néhány fogalom értelmezése:

Cél: az elemzés tárgya.

Korlátozás: ami a megvalósításban határt szab a lehetőségeknek.

Alternatívák: a célok megvalósításának különböző útjai.

Kockázatok: az egyes alternatívák nagy valószínűséggel hibát okozó forrásai.

Kockázat kezelése: stratégia a kockázat hatásának a csökkentésére.

Validáció: a terv előírt célok szerinti teljesülésének ellenőrzése.

A spirális modell a projektmunka szervezésének áttekinthető, szabványos modellje, amelynek előnyeit a következőkben foglalhatjuk össze:

- A projekt kidolgozásának jó dokumentáltságát eredményezi, mivel mind a négy szakaszban dokumentumok készülnek.
- A projekt strukturáltsága, az iterációs fázisok – a projektvezető elképzelésének megfelelően – szabadon megválaszthatók.
- A fázisokon belüli szakaszok tartalmának (a kockázati tényezők, azok kezelése stb.) meghatározása szintén iterációs lépésenként, a projektvezető elképzelése szerint történhet meg.
- A fejlesztést, az eredményt mindig validáció követi.
- Ciklusonként döntés születhet a projekt egy újabb fázisáról, ha az eredmény alapján az kívánatos.

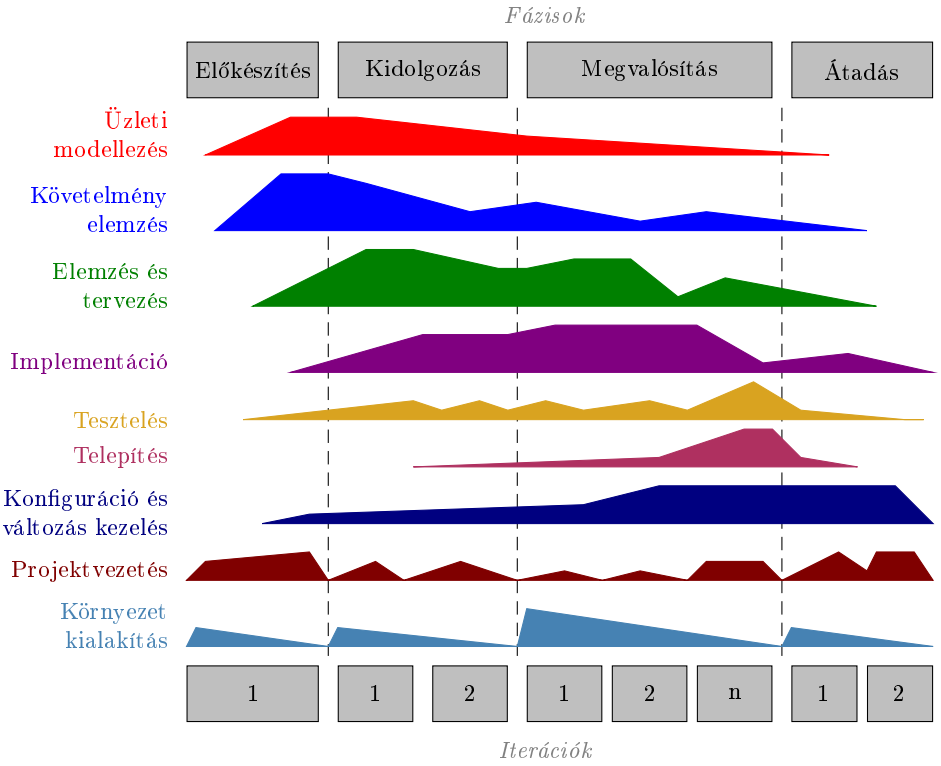
A spirális modell alkalmazásával kapcsolatban a következő problémákat szokták felvetni:

- A modell alkalmazása általában munkaigényes, bonyolult feladat. A programkészítés általunk választott minden fázisának fenti módon történő szakaszokra bontása és végrehajtása általában költséges és nehezen megoldható.
- A projekt kidolgozásához szükséges szakembereket nem könnyű gazdaságosan foglalkoztatni. A párhuzamos foglalkoztatás szinte csak a 3. szakasz során lehetséges. Ezért a modellel kapcsolatban gyakran elhangzik a gazdaságtalanság jelzője.

1.5. RUP

A RUP (Rational Unified Process) az UML alkotói által kidolgozott iteratív szoftverfejlesztési folyamat. A modellben fázisokat és munkafolyamatokat különböztetünk meg. A fejlesztés négy fázisból áll, és minden fázis egy vagy több iterációból. Az egyes fázisokon, illetve iterációkon belül a munkafolyamatok eltérő súllyal szerepelnek.

A RUP fázisai sorrendben a következők: előkészítés, kidolgozás, megvalósítás, átadás. A munkafolyamatok: üzleti modellezés, követelmény elemzés, elemzés és tervezés, implementáció, tesztelés, telepítés, konfiguráció és változás kezelés, projektvezetés, környezet kialakítás. Az egyes iterációkban a munkafolyamatok a megadott sorrendben követik egymást. Az 1.5. ábra egy átlagos projekt összes elvégzendő munkájának az egyes fázisokra és



1.5. ábra. RUP munkafolyamatok és fázisok

iterációkra eső hányadát mutatja. Az egyes alakzatok területe arányos a munka mennyiségével.

A RUP használati esetek által vezérelt, architektúra centrikus és inkrementális fejlesztési modell. A használati esetek által vezérelt modell jelentése, hogy az architektúrát az alkalmasan választott használati esetek tervezése, implementálása és tesztelése jellemzi, majd a megvalósítás fázis iterációiban a megfelelő sorrendben választott további használati eseteket valósítják meg. Az architektúra centrikusság jelentése, hogy a kidolgozás fázis végére az architektúrának stabilnak kell lennie. Az inkrementális modell úgy értendő, hogy a szoftver fokozatosan épül. Az architekturális alapverzióból indulnak ki, és minden iterációban az akkor megvalósított használati eseteket teljesen beépítik a rendszerbe.

1.5.1. Előkészítés

Ebben a fázisban a következő kérdések kerülnek a középpontba.

1. A rendszer milyen szolgáltatásokat nyújtson a legfontosabb felhasználók számára?
2. Milyen lehet a rendszer architektúrája?
3. Mi a terve és a költsége a rendszer fejlesztésének?

Az első kérdésre a használati esetek és aktivációs diagramok segítségével, valamint a funkcionális és nem funkcionális követelmények meghatározásával válaszolhatunk. A rendszer architektúrája ebben a szakaszban még csak kísérleti, és csak a fontosabb modulok,

alrendszerek vázlatát tartalmazza. A harmadik kérdésre a legjelentősebb kockázati tényezők felsorolása és sorrendbe állítása, a következő fázis részletes megtervezése, valamint az egész projekt előzetes terve és költségbecslése ad választ.

1.5.2. Kidolgozás

Ebben a fázisban a következő kérdésre kell választ adni.

- Kellően stabilok és kezelhetőek-e a használati esetek, az architektúra és a tervek ahhoz, hogy a teljes fejlesztési munkára szerződést lehessen kötni?

Ebben a fázisban elkészül a legtöbb használati eset specifikációja és a rendszer architektúrájának terve. A rendszer és az architektúra közötti összefüggés dominálja ezt a fázist. A RUP modell alkotói szerint az architektúra egy bőrrel fedett csontváz, amelyen ebben a fázisban csak minimális mennyiségű izom (programkód) található, ami csak arra elegendő, hogy a fontosabb mozgásokat elvégezhesse a test. A rendszer a teljes testnek felel meg, csontvázzal, izmokkal, bőrrel.

A rendszer architektúrájára úgy tekinthetünk, mint a rendszer összes modelljének nézeteire. Ennek értelmében a használati, az elemzési, a tervezési, az implementációs és a környezeti modellnek is van architekturális nézete. Az implementációs modell architekturális nézetéhez tartozó komponensek segítségével kimutatható, hogy az architektúra működőképes, azaz futtatható. Ezért ebben a fázisban a legfontosabb használati esetek azonosítása után, azokat implementálják és tesztelik is. Ez adja meg a szoftver vázát az előző hasonlatnak megfelelően.

A fázis végére a projektvezető kellő ismerettel rendelkezik ahhoz, hogy képes legyen a hátralévő egész fejlesztési munka megtervezésére, és a szükséges erőforrások felbecslésére.

A fázis munkáit addig kell folytatni, amíg a kellő stabilitás és főbb kockázatok kezelése nem biztosított. A fázis eredménye a rendszer *architekturális alapverziója*.

1.5.3. Megvalósítás

A megvalósítás alatt elkészül a termék, azaz a hasonlat szerint, az izmok rákerülnek a csontokra. Az architekturális alapverzió kész rendszerré fejlődik. Ebben a fázisban a fejlesztéshez szükséges erőforrások nagy részét felhasználja a projekt.

A rendszer architektúrája ugyan stabil, de előfordulhat, hogy a fejlesztők felfedeznek jobb lehetőségeket a rendszer szerkezetének kialakítására, így kisebb architekturális változásokat javasolhatnak, megvalósíthatnak. A fázis végére az összes használati esetet implementálják, amelynek kiadásáról megegyezett a megrendelő és a projektvezetés. Ekkor a rendszer még nem feltétlen hibátlan, az esetleges hibákat azonosítani és javítani lehet az átadási fázisban.

A fázis munkáit addig kell folytatni, amíg a következő kérdésre igenlő választ nem lehet adni.

- Megfelel-e a termék annyira a felhasználói igényeknek, hogy néhány kiválasztott felhasználó megkaphasson egy korai (béta) kiadást?

1.5.4. Átadás

Ebben a fázisban a termék először egy béta kiadás formájában kerül kisszámú gyakorlott felhasználóhoz. Azok kipróbálják a terméket, és jelentik az észlelt hibákat, hiányosságokat.

A fejlesztők kijavítják a hibákat, a javasolt javítások egy részét beépítik a rendszerbe, és előállítják az általános kiadást. Ez a teljes felhasználói társadalom elé kerül.

Itt kerül sor olyan tevékenységekre, mint kézikönyvek és CD-k gyártása, felhasználói személyzet képzése, támogatás létrehozása és fenntartása, a kiadás után észlelt hibák javítása.

A kiadás után észlelt hibákat a karbantartó csapat rendszerint két részre osztja.

- Olyan hibák, amelyek annyira befolyásolják a rendszer működését, hogy kijavításuk egy azonnali delta kiadást tesz indokolttá.
- Olyan hibák, amelyek kijavítására a következő, rendesen tervezett kiadásban kerül sor.

1.6. XP

Az XP az eXtreme Programming rövidítése. Ez egy könnyűsúlyú fejlesztési modell, amely határozatlan és változékony követelmények, valamint kis projektcsapatok esetén használható. A modell alapvető elemei, paradigmái a következők.

- A vezetni tanulás története: a projekt vezetésekor a látóhatárra tekintsünk, ne közvetlenül a lábunk elé.
- A négy alapérték:
  1. *Kommunikáció*, amelynek minden irányban működnie kell a megrendelő, a projektvezető és a fejlesztők között.
  2. *Egyszerűség*, azaz mindig a legegyszerűbb megoldást kell keresni, ami működik.
  3. *Visszajelzés*, amelynek három fontos esete a következő. A megrendelő megadja a rendszer tulajdonságainak leírását, amit a fejlesztők kiértékelnek. A kód és az egységteszt viszonya. A rendszerkövető követi a feladatokat, és a csapat visszajelzést kap a haladásról.
  4. *Bátorság*, amelynek értelmében nem kell félni a kód megváltoztatásától, illetve eldobásától.
- Az elvek, amelyeket a következőkben külön tárgyalunk.
- A négy alaptevékenység:
  1. *Kódolás*: A kód tisztán és tömören fejezi ki az elképzeléseket. A kód az a fizikai összetevő, amely nélkül nem képzelhető el szoftverfejlesztés.
  2. *Tesztelés*: A kód könnyebben változtatható és a programozás is szórakoztatóbb, ha rendelkezünk kész tesztesetekkel. Akkor tekintjük a kódot elkészültnek, ha nem tudunk elképzelni olyan tesztesetet, amely hibát eredményezne.
  3. *Meghallgatás*: A fejlesztők semmit sem tudnak a fejlesztendő rendszerről, ezért kérdezni kell, és a válaszokat meg kell hallgatni.
  4. *Tervezés*: Nem elég előállítani a teszteseteket, a kódot, futtatni a kódot a tesztesetekre, majd új teszteket készíteni és kódot írni, futtatni stb. Ez egy idő után nem működik. Szükséges a kód szerkezetének megtervezése.

1.6.1. Az XP elvei

Az elvek első csoportját alkotják az úgynevezett alapelvek, amelyek a következők.

- *Gyors visszajelzés*: Az akció és a visszajelzés között eltelt idő kritikus a tanulás szempontjából. Ez érinti a kód előállítás és tesztelés, valamint a programozó és a megrendelő viszonyát.
- *Az egyszerűség feltételezése*: A problémák 98 százalékát nagyon egyszerű megoldani. Ha ennek megfelelően járunk el, akkor rengeteg időnk marad a maradék 2% megoldására. (Ez a legnehezebben elfogadható elv, mert a programozók minden előre látható problémát azonnal meg akarnak oldani.)
- *Fokozatos változtatás*: Nagy változások rendszerint nem hajthatóak végre sikeresen. Ezért egy XP projekt egész ideje alatt csak kicsit változik a terv, a csapat és az XP alkalmazása.
- *Változtatások felkarolása*: A könnyű változtatás legjobb stratégiája az, amelyben a lehető legtöbbet megtartunk a már működő rendszerből, és egyben megoldjuk az aktuális problémát.
- *Minőségi munka*: A projekt négy változója a rendszer működési területe, a költség, az idő és a minőség. Ezek közül a minőség nem szabad változó, a lehetséges értékei kiváló vagy őrülten kiváló.

A további elvek a következők.

- *Taníts tanulni*: Helytelen például azt mondani, hogy neked is úgy kell tesztelni, mint X.Y-nak. Ezzel szemben stratégiát kell tanítani arra, hogy mennyi tesztre van szükség, mennyi tervezésre, stb.
- *Kismértékű kezdeti befektetések*: Túl sok kezdeti erőforrás rendszerint sikertelenséghez vezet. A szűkös lehetőségek ösztönöznek az erőforrások helyes felhasználására. Ugyanakkor túl kevés erőforrás, amelyekkel nem lehet a rendszer képességeit bemutatni, a projekt leállásához vezethet.
- *Dolgozz a győzelemért*: A pozitív motiváció nagyon fontos, a siker valószínűsége ilyenkor nagyobb. (Az UCLA kosárlabda csapata az utolsó pillanatig a győzelemért hajtott, és győzött. Az Oregon csapata 12 pontos előnyről vesztett, mert csak meg akarta tartani azt.)
- *Konkrét kísérletek*: Minden, tesztelés nélkül meghozott döntés magában hordozza a hiba valószínűségét.
- *Nyílt, becsületes kommunikáció*: Meg lehessen mondani a magunk vagy a mások által elkövetett hibákat. Baj van a projektcsapatban, ha valaki körülnéz, mielőtt beszélne.
- *Dolgozz az emberek ösztöneivel, ne azok ellen*: Az emberek szeretnek tanulni, győzni, együttműködni másokkal, egy csapathoz tartozni, jó munkát végezni és látni, hogy amit csinálnak működik. Ha az XP nem tud ezen emberi értékeknek megfelelni, akkor nem válhat elfogadott fejlesztési modellé.
- *Elfogadott felelősség*: Ha a felelősséget valakire ráruházzák, különösen ha az elvárás teljesíthetetlen, akkor az frusztrációhoz vezet. Felelősséget nem ráruházni kell, hanem az embereknek önként kell vállalni és elfogadni azt.

- *Helyi adaptáció*: Az XP-t a helyi adottságok szerint kell alkalmazni.
- *Mobilitás*: Ne bővelkedjünk a tárgyi feltételekben, de azok legyenek egyszerűek és értékesek. Egy XP csapat szellemi vándorokból áll, ahol a csorda a terv és a megrendelő, ami bármikor egy váratlan irányba mehet, illetve bármelyik csapattag bármikor elhagyhatja a csapatot.
- *Becsületes mérések*: A túlzásba vitt mérések eredménytelenek lehetnek. Jobb azt mondani, hogy egy feladat kidolgozása kb. két hetet vesz igénybe, mint azt, hogy 76,15 órát. A metrikákat a munkánknak megfelelően válasszuk meg. Például XP-ben a kódsorok száma nem alkalmas a termelékenység mérésére, mert az XP-ben érvényes egyszerűsítési elv alapján lehetőség szerint csökkenteni kell a kódsorok számát. (Ha így egyszerűbb programhoz jutunk.)

1.6.2. Az XP alkalmazása

Az XP modell alkalmazása során a következő főbb területeket szokás megkülönböztetni.

- A *tervezési játék*, ami három fázisból áll.
  1. Feltárás: A cél, hogy a megrendelő és a fejlesztők megértsék a rendszer képességeit, lehetőségeit. A megrendelő esemény kártyákra írja a rendszer funkcióit. Ezek megvalósíthatóságát a fejlesztők megvizsgálják, és megfelelő feladat kártyákat készítenek.
  2. Kötelezettség vállalás: A cél, hogy a megrendelő határozza meg a rendszer következő kiadásának funkcionalitását és a kiadás dátumát, illetve a fejlesztők magabiztosan kötelezzék el magukat annak teljesítésére.
  3. Irányítás. Célja a terv frissítése a megrendelő és a fejlesztők tapasztalatai alapján.
- *Kisméretű kiadások*: Egy kiadás elkészítésének ideje inkább 1 vagy 2 hónap legyen, mint 6 vagy 12, de mindenképpen egy teljesen felhasználható rendszerrészt kell átadni.
- *Metafora*: Egy XP projektet a rendszer jellemzőit összefogó egyszerű leírás irányít. Ez a gyakorlatban úgy valósul meg, hogy például egy szerződés menedzsment rendszerről a projekten belül a szerződések, ügyfelek, jóváhagyások szavakkal beszélnek, illetve, hogy egy számítógép íróasztalként jelenik meg, vagy, hogy a nyugdíj számítás egy táblázatkezelő programra hasonlít.
- *Egyszerű terv*: A rendszer terve a lehető legegyszerűbb legyen, minden felesleges elemet azonnal el kell távolítani.
- *Tesztelés*: A fejlesztők a kód megírása előtt egységteszteket állítanak elő. A teszteredményeknek teljesen meg kell felelni, mielőtt újabb egység fejlesztésébe kezdenének. A megrendelő átvételi teszteket ír.
- *Átstrukturálás* (Refactoring): A fejlesztők átalakítják a rendszer szerkezetét úgy, hogy annak viselkedése ne változzon. A cél az ismétlések eltávolítása, egyszerűsítések végrehajtása és a rugalmasság növelése.
- *Párban programozás*: Minden kódot két programozó készít egy gépen.

- *Közös tulajdon*: Mindenki megváltoztathat akármilyen kódot, bárhol és bármikor a rendszerben.
- *Folyamatos integrálás*: A rendszer integrálása és építése naponta akár többször is lehetséges. Erre minden részfeladat elvégzése után sor kerül.
- *40 órás munkahét*: Lehetőség szerint senki ne dolgozzon többet egy héten 40 óránál. Ha mégis előfordul túlóra, akkor senki se túlórázzon két egymást követő héten.
- *Helyben tartózkodó megrendelő*: Egy valódi megrendelő vagy felhasználó legyen teljes munkaidőben a csapat tagja.
- *Kódolási követelmények*: A kód is a kommunikáció eszköze, ezért annak egységesnek kell lennie. A szabványt mindenkinek önként el kell fogadnia. A szabvány törekedjen a minimális munkabefektetés elvére, és tartalmazza az „egyszer és csak egyszer" szabályt, azaz nem lehet kétszeres kód.

1.6.3. Egy XP projekt életciklusa

Egy ideális XP projekt a következő fázisokat tartalmazza.

1. *Feltárás*: A lehető legrövidebb ideig tartson. Akkor ér véget, amikor a megrendelőnek elegendő esemény kártyája van az első kiadáshoz, amely a legkisebb és legfontosabb cselekmény halmazból áll, továbbá a fejlesztők biztosak abban, hogy nem tudnak jobb rendszerarchitektúrát kialakítani. (Ezt megelőzően három-négy architektúrát is kipróbáltak a rendszer főbb aspektusait implementálva.) A fázis időtartama gyakorlott csapat esetén néhány hét, kezdő esetén néhány hónap.
2. *Tervezés*: Itt kell megegyezni a megrendelővel az első kiadás implementálásának idejében, ami általában 2 és 6 hónap között változik. Időtartam: 1–2 nap.
3. *Iterációk az első kiadásig*: A vállalt implementációs időtartamot 1–4 hetes iterációkra kell bontani. Minden iteráció minden cselekményéhez teszthalmaz tartozik. Az első iteráció az architektúrát stabilizálja. Időtartam: 2–6 hónap.
4. *Termelésbe helyezés*: Ez rövidebb iterációkból (1 hét) áll. A termelési hardver már a helyén van. A fázis feladata a teljesítmény beállítása, hangolása, illetve új tesztesetek létrehozása a termelés tesztelésére. A kockázatok csökkentése érdekében minimális változtatásokat kell elvégezni. Ebben a fázisban bekövetkezik valamilyen formában a szoftver átvételének igazolása, amit meg kell ünnepelni. Időtartam: néhány hét.
5. *Karbantartás*: Az XP projekt normális állapota. A meglevő rendszer működése közben azt új funkciókkal egészítik ki, új munkatársak kerülnek a csapatba, régi munkatársak elmennek. Minden új kiadás egy feltárási fázissal kezdődik, egy új cikluson megy keresztül, és a termelésbe vitellel zárul. A fejlesztők megpróbálják azokat az átstrukturálásokat megvalósítani, és azokat az új technológiákat bevezetni, amelyeket nem tettek meg az előző kiadásban. A megrendelő olyan új cselekményeket dolgoz ki, amelyek üzleti előnyöket adnak vállalatának. Egy lezárult kiadást egy új követ. Időtartam: a következő fázisig.
6. *Befejezés*: Amikor a megrendelő már nem tud új cselekményeket megadni, azaz teljesen elégedett a rendszerrel, akkor a rendszer tartalékba kerül. Egy 4–5 oldalas

dokumentumban meg kell fogalmazni a szükséges tudnivalókat egy 4–5 éven belül bekövetkező esetleges változtatáshoz.

Egy másik lehetséges befejezés, ha a megrendelő olyan módon szeretné bővíteni a rendszert, amit a fejlesztők a megrendelő által megadott áron belül nem tudnak megvalósítani. Remélhetőleg ez elég ritkán fordul elő.

1.7. Végrehajtható UML

A szoftverfejlesztés hagyományos megközelítésében gyakran két elkülönült modellt hoznak létre az elemzés és a tervezés céljára. Ez azon az elgondoláson alapul, hogy az elemzési modellben a fejlesztők a rendszer funkcionalitására (mit csinál) koncentrálnak, és figyelmen kívül hagyják azt, hogy ez miként érhető el. A tervezési modell ezzel szemben megadja, hogy a rendszer miként valósítja meg az elemzési modellben leírt viselkedést. A két modellt *platform független* (*PIM*), illetve *platform specifikus modellnek* (*PSM*) is nevezik.

A megközelítés előnye, hogy függetleníti az elemzést a tervezéstől. Ez lehetővé teszi, hogy a fejlesztők a rendszer követelményeinek meghatározására és megértésére koncentráljanak, anélkül, hogy a figyelmüket egy adott implementáció részletei megzavarnák. Ennek eredményeként olyan terv áll elő, ami pontosabban írja le, hogy mit kell csinálnia az elkészítendő rendszernek. A létrehozott modellt többször újra fel lehet használni különböző implementációkban, ahogy a technológia változik. A tervezési fázisban a fejlesztők az adott technológiát figyelembe vevő optimális megoldás elkészítésére koncentrálhatnak a követelmények pontos ismeretében.

A módszer alkalmazásakor a következő problémák merülhetnek fel.

- A gyakorlatban sokszor nehéz elkülöníteni az elemzést és a tervezést. Gyakran az elemzési szakasz eredménye egy pontatlan, elnagyolt modell, amelynek megértéséhez a rendszer tervezőinek értelmezése szükséges. Ez nem biztos, hogy megfelelő eredményhez vezet, illetve nem garantált, hogy a tervezők rendelkeznek a problémával kapcsolatos elegendő ismerettel.

- Nehéz eldönteni, hogy az elemzési modell mikor teljes. Sok időt igényel annak eldöntése, hogy mi kerüljön be, és mi maradjon ki a modellből.

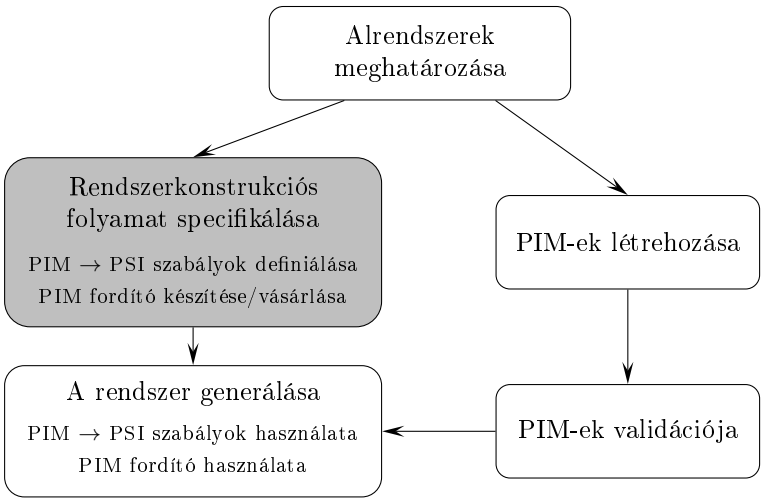
- Miután nem állnak rendelkezésünkre pontos kritériumok annak eldöntésére, hogy mit tartalmazzon az elemzési modell, ezért nehéz a modell tartalmát és minőségét értékelni.

- Ahogy az előző fejezetben már tárgyaltuk két, vagy több modell konzisztenciájának fenntartása nehézkes. Ha csak az egyik modellt tartjuk meg, akkor az elkülönítésből adódó összes előny elvész.

A fenti nehézségek egy lehetséges megoldása a *végrehajtható UML* (továbbiakban *xUML*) alkalmazása. Az xUML egyrészt az UML leszűkítése, másrészt annak kiterjesztése. Az UML-ből elhagyják a szemantikailag nem egyértelmű részeket, másrészt kiegészítik az automatikus kódgeneráláshoz szükséges eszközzel. Ez az *Action Specification Language*, röviden *ASL*.

Az ASL jelenleg nem része az UML szabványnak, de szerepel az OMG által menedzselt UML 1.5 leírásban. Ott három szintaktikailag minimálisan eltérő változat található meg. Sokak szerint közeljövőben már a szabvány része lesz.

Az xUML-ben a szoftverfejlesztés elve megegyezik a mérnöki gyakorlatban alkalmazott elvekkel. Így a fejlesztés menete a következő (1.6. ábra).



1.6. ábra. Szofverfejlesztés menete xUML-ben

1. Az alrendszerek meghatározása.
2. Pontos, a rendszer működését előrejelző, platform független modellek létrehozása.
3. A modellek kimerítő tesztelése az implementáció előtt.
4. Egy jól meghatározott (lehetőleg automatikus) előállítási folyamat bevezetése.
5. A termék előállítása újrafelhasználható elemekből.

Egy xUML modell megalkotása a következő lépésekből áll.

1. A rendszer részekre bontása.
   - 1.1. A rendszer felosztása, a megfelelő *domain*-ek létrehozása, azok kapcsolatainak azonosítása. Egy domain lényegében egy alrendszer, így itt egy vázlatos alrendszer diagram készül el.
   - 1.2. A rendszer használati eseteinek azonosítása, az egyes használati esetekhez tartozó szekvenciadiagramok elkészítése. A szekvenciadiagramokban az osztályszerepeket az egyes alrendszerek töltik be.
   - 1.3. Az alrendszerek kapcsolatainak definiálása. A kapcsolódási felületek megadása.
   - 1.4. Híd kapcsolatok megadása. Ha kettő, vagy több alrendszert vonunk össze, akkor egy híd kapcsolat adja meg a platform független megfeleltetést a megfelelő műveletek között.
2. Az új alrendszerek platform független modelljének létrehozása.
   - 2.1. A statikus modell (osztálydiagram) elkészítése. Ebben azok az adatok szerepelnek, amelyek szükségesek a dinamikus viselkedés, illetve az alrendszer funkcionalitásának megvalósításához. (Ezek nem feltétlen egyeznek meg a tárolt adatokkal.)
   - 2.2. A dinamikus modell létrehozása. Ez az állapotdiagramok elkészítését jelenti. Ezekben nem szerepel általánosítás és aggregáció, viszont a viselkedés pontos leírásához szükség esetén megadják az állapotok (belépési) akcióit.

2.3. Az akciók meghatározása. Akciókat két helyen definiálnak: az állapotdiagramokban az egyes állapotokhoz, illetve a műveletek megvalósításában. Az akciókat az ASL segítségével adják meg.

3. A platform független alrendszer modellek tesztelése, validációja. Ennek során az alrendszer használati eseteit, majd a rendszer használati eseteit hajtják végre, és ellenőrzik az eredményt.

4. A rendszer létrehozása. A platform független modellek platform specifikus implementációjának (PSI) előállítása platform specifikus megfeleltetések felhasználásával.

   4.1. Általános absztrakt tervminták specifikálása.

   4.2. A szoftver-terv értékelése és tesztelése.

   4.3. Automatikus kódgenerálás.

A rendszert csak akkor hozhatjuk létre automatikus kódgenerálással, ha rendelkezünk egy PIM fordítóval. Ezt az adott alkalmazási terület és a fejlesztő eszközök figyelembe vételével lehet elkészíteni. Ekkor magasan képzett szakemberek adják meg a PIM $\rightarrow$ PSI szabályokat, amelyekből létrehozható a fordító.

A platform független tervek validációja, a tervek végrehajtását igényli. Ezt egy újabb metanyelv bevezetésével, és a megfelelő fordító vagy értelmező létrehozásával lehet elérni.

1.7.1. ASL

Ez a nyelv teszi lehetővé az akciók pontos, platform független megadását. Megjegyzéseket elhelyezhetünk a # jel után.

A nyelv egyik részét a vezérlési szerkezetek alkotják. Lehetőségünk van feltételek szerint eltérő tevékenységek végrehajtását kezdeményezni. Az egyik utasítás a jól ismert *if-then-else* szerkezet. Ennek formája:

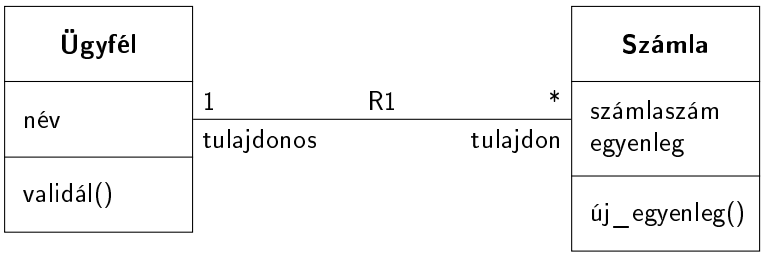
if feltétel then
    $utasítás_1$
else
    $utasítás_2$

A másik lehetőség egy érték szerinti több ágú elágazás. (A C++, illetve Java nyelvektől eltérően egy ág nem folytatódik a rákövetkezőkön, azaz minden ág tartalmaz egy implicit *break*-et.)

switch változó
    case 1 $utasítás_1$
    case 2 $utasítás_2$
    $\vdots$

A nyelv másik része az objektumok kezelésével kapcsolatos eszközöket tartalmazza. Objektumok létrehozásának módját a következő példa első sora szemlélteti. Ha a létrehozáskor attribútumokat is meg akarunk adni, akkor azt is megtehetjük, amint az a második sorban látható.

új_objektum = create Objektum
új_ügyfél = create Ügyfél with név = új_név



1.7. ábra. Az ügyfelek és a számlák közötti kapcsolat

Objektumok megsemmisítése:

delete objektum

Attribútumok értékének állítása (feltételezzük, hogy a **Számla** osztály attribútuma az egyenleg, és számla az osztály egy objektuma):

számla.egyenleg = új_egyenleg

A nyelvben lehetőségünk van egy osztály objektumainak kiválasztására, és az eredményt egy lokális változóba tehetjük. A változó lehet egyke, vagy gyűjtemény. Egyetlen objektum kiválasztását mutatja a következő példa első sora, egy csoportét a második sor.

számla = find-only Számla where számlaszám = 42
{negatív-számlák} = find-all Számla where egyenleg < 0

Két objektumot a következő módon kapcsolhatunk össze. Tegyük fel, hogy a számla objektumot akarjuk R1 reláció szerint összekapcsolni az ügyfél objektummal. Az R1 reláció multiplicitása ebben az esetben 1 és *, azaz egyetlen ügyfelet kapcsolhatunk egy számlához (1.7. ábra).

link számla R1 ügyfél

Egy kapcsolat megszüntetésének módja:

unlink ügyfél R1 számla

Egy kapcsolat szerinti navigációt, azaz a kapcsolatban részt vevő egyetlen objektum kiválasztását a következőképpen tehetjük meg, ha az előző példában egy számla tulajdonosát (ügyfelet) akarjuk meghatározni.

tulaj = számla->R1

Több objektum kiválasztásának módja:

{számlák} = ügyfél->R1

A nyelv harmadik része az üzenetek küldésével kapcsolatos. Lehetőség van paraméter nélküli, vagy paraméteres műveletek meghívására. Az ügyfél objektum validál műveletének

aktivizálásának módja:

[] = validál[] on ügyfél

Egy számla új_egyenleg műveletének paramétere az új érték, és visszatérési értéke az új egyenleg. Ezt a következőképpen lehet meghívni.

[új_érték] = új_egyenleg[érték] on számla

Paraméteres vagy paraméter nélküli üzenetet (signal) is küldhetünk objektumoknak. Egy paraméteres üzenet küldésének módja:

generate felújítás_igény(időszak) to számla

2. Minőségkezelés

A legtöbb cég arra törekszik, hogy termékeik, szolgáltatásaik minőségét magas szinten biztosítsa. A szoftver is termék, tehát erre, és az előállítóira is vonatkozik az előző megállapítás. Nem elfogadható eljárás egy gyenge minőségű termék átadása, és a hibák, hiányosságok üzembe helyezés utáni javítása.

Egy szervezeten belül az úgynevezett minőségi vezetők felelőssége, hogy a termék elérje a megkívánt szintet. A minőségkezelést és a projektvezetést célszerű elkülöníteni, hogy a költségvetésért és ütemezésért vállalt felelősség váljon el a minőségi felelősségtől, így azok ne veszélyeztessék a minőséget.

A minőségkezelés három részből áll.

1. A *minőségbiztosítás* célja magas minőségű szoftverek előállítását eredményező szervezeti eljárások és szabványok rendszerének létrehozása.

2. A *minőségtervezés* feladata a rendszerből megfelelő eljárásokat és szabványokat kiválasztani, és ezeket egy adott projektre szabni.

3. A *minőség ellenőrzése* során meg kell határozni és rendszerbe kell állítani azokat a folyamatokat, amelyek garantálják, hogy a fejlesztők alkalmazzák a kiválasztott eljárásokat, szabványokat.

A minőségbiztosítást, illetve az ellenőrzést független csapatoknak kell végezniük. Így a minőségkezelési folyamat tárgyilagosan véleményezhető.

A minőségkezelés egyik általános szabványa az ISO 9000. Szabványai közül az ISO 9001-es a legáltalánosabb, amely termékek tervezésével, fejlesztésével, karbantartásával foglalkozó szervezetekre vonatkozik. A szoftverfejlesztéshez az ISO 9000-3 kiegészítő dokumentum nyújt ajánlásokat.

Az ISO 9001 a minőségi folyamat általános modelljét adja meg. A szabvány ágazatfüggetlen, ezért a leírt szabványokat és eljárásokat nem definiálja részletesen.

A minőségkezelés három területe közül a továbbiakban csak a minőségbiztosítással foglalkozunk.

2.1. Minőségbiztosítás

A minőségbiztosítás tartalmazza a szoftverfejlesztés során, illetve a termékre vonatkozó szabványok meghatározását és kiválasztását. Kétféle szabványt különböztethetünk meg.

- A *termékszabványok* a készülő szoftverre vonatkozó szabványok, mint például a dokumentumokra vonatkozó szabványok, a dokumentációs előírások, a kódolási szabványok.

- A *folyamatszabványok* a fejlesztés menetét meghatározó szabványok, például a tervezés, validálás folyamatát leíró szabványok.

A szabványok a következők miatt fontosak.

- Összefoglalják a legjobb, illetve legrosszabb gyakorlati elemeket, amelyeket általában csak sok kísérletezés és kudarc után szerezhetünk meg. Így elkerülhetők a régi hibák, illetve megmaradnak az értékes tapasztalatok.
- Keretrendszert adnak a minőségbiztosítási folyamatnak. Ekkor a minőség ellenőrzés során csak a szabványok betartására kell ügyelni.
- Támogatják a folytonosságot. Ha új tag kapcsolódik a fejlesztésbe, illetve ha új munkába kezdenek, csökken a tanulás mennyisége.

Minden szoftver projekt tervének tartalmaznia kell minőségbiztosítási tervet. A siker egyik kulcsa a megfelelő minőségbiztosítási terv kidolgozása, és annak minden részletének betartása.

Egy megfelelő minőségbiztosítási terv kidolgozásához segítséget nyújt az Institute of Electrical and Electronics Engineering szabványa az IEEE Std 730-1998: IEEE Standard for Software Quality Assurance Plans. A szabvány kompatibilis az ISO 9000-3 és ISO 9001 követelményeivel, továbbá megfelel az ISO 10005 szabványnak. A szabvány tartalmazza a CMM második szintjének minden lényeges területét, de a minőségbiztosítás került a középpontba.

2.1.1. IEEE Std 730-1998 szabvány szerkezete

A szabvány szerint a minőségbiztosítási tervnek a következő 15 fejezetet kell tartalmaznia.

1. Szándék: a terv speciális szándéka és működési területe.
   - 1.1. A szoftver azon elemeinek megnevezése, amelyekre alkalmazni kell a tervet.
   - 1.2. A szoftver életciklusának azon részei, amelyek alatt alkalmazni kell a tervet az egyes elemekre.
2. Hivatkozott dokumentumok: a tervben hivatkozott dokumentumok teljes listája.
3. Vezetés: a szervezés, a feladatok és a felelősségek leírása.
   - 3.1. Szervezés: a szervezési struktúra, amely befolyásolja és ellenőrzi a szoftver minőségét.
   - 3.2. Feladatok: a szoftver életciklusának a minőségbiztosítási terv alá eső része; az elvégzendő feladatok, hangsúlyozva a minőségbiztosítási tevékenységeket; kapcsolat a feladatok és a tervezett fontosabb ellenőrzési pontok között.
   - 3.3. Felelősségek.
     - Közös felelősségek esetén tisztázni kell a szerepeket.
     - A menedzseri pozícióhoz tartozik az összes felelősség a teljes szoftverminőség biztosításáért.
     - A felülvizsgálati és elfogadási ciklust, valamint az aláírási jogosítványokat meg kell határozni.
     - Egy táblázatban fel kell tüntetni a személyi és szervezeti felelősségeket a feladatokkal együtt.
4. Dokumentáció.

4.1. Szándék.

- Azon dokumentumok meghatározása, amelyek a fejlesztést, a verifikálást és a validálást, a felhasználást, karbantartást irányítják.
- A dokumentumok alkalmassági ellenőrzésének megállapítása (utalásokkal a 6. fejezet megfelelő részeire).

4.2. Minimális dokumentációs követelmények.

- Követelmények (ellenőrizhető) specifikációja.
- Terv leírása. (Előzetes és részletes terv.)
- Verifikációs és validációs terv.
- Verifikációs és validációs jelentés.
- Felhasználói dokumentáció.
- Konfiguráció kezelés terve.

4.3. Egyéb (projektterv, szabványok és eljárások kézikönyve, projektmenedzselési terv, karbantartási kézikönyv, telepítési útmutató, ...).

5. Szabványok, eljárások, konvenciók és metrikák.

5.1. Szándék: az alkalmazandó szabványok, eljárások, konvenciók és metrikák azonosítása; annak megállapítása, miként biztosítható, hogy ezeknek megfeleljünk.

5.2. Tartalom: dokumentációs, szerkezeti, kódolási, tesztelés szabványok, kiválasztott termék és folyamat metrikák. (Ezek eltérőek lehetnek az életciklus különböző fázisaiban.)

6. Felülvizsgálatok és ellenőrzések.

6.1. Szándék: az elvégzendő menedzseri és technikai felülvizsgálatok és ellenőrzések definiálása, ezek megvalósításának meghatározása, továbbá annak megállapítása, hogy milyen egyéb tevékenységek szükségesek, és ezek hogyan implementálhatóak és verifikálhatóak.

6.2. Minimális követelmények. Minden felülvizsgálat és ellenőrzés meghatározásakor azonosítani kell a felelősségeket. A felülvizsgálatok, ellenőrzések eredményét dokumentálni kell, és azonosítani kell a korrigáló cselekményeket, amelyek elvégzése után fejeződik be az aktuális munkafolyamat. A következő felülvizsgálatok, ellenőrzések tartoznak ide.

- Követelmény felülvizsgálat (teljesség, tesztelhetőség, kompatibilitás).
- Előzetes terv felülvizsgálata (kapcsolatok, komponensek, adatbázis).
- Részletes terv felülvizsgálata.
- Verifikációs és validációs terv felülvizsgálata.
- Funkcionális ellenőrzés (átadás előtt).
- Fizikális ellenőrzés (átadandó kód és dokumentáció megfelel-e egymásnak).
- Folyamaton belüli ellenőrzések, amelyek a fejlesztési folyamatot mérik.
- Vezetői felülvizsgálatok, amelyek időközönként értékelik a minőségbiztosítási terv elemeit és a terv végrehajtását.
- Konfigurációs terv felülvizsgálata.
- Utólagos felülvizsgálat, amelyben a projekt lezárása után értékeli ki a fejlesztési tevékenységeket, és megfelelő intézkedéseket javasol.

6.3. Egyéb, például a felhasználói dokumentáció felülvizsgálata.

7. Teszt: azon tesztesetek azonosítása, amelyeket nem fed le a verifikációs és validációs jelentés.

8. Probléma jelentés és javító cselekmény. Egyrészt leírja a szoftverben, a fejlesztésben, a karbantartás során felmerülő problémák jelentésére, nyomon követésére megoldására alkalmazott eljárásokat, másrészt megállapítja a speciális szervezeti felelősségeket az eljárások megvalósításhoz.

9. Eszközök, technikák, módszerek. A minőségbiztosítást támogató eszközök, technikák, módszerek meghatározása és használatának leírása.

10. Kód felügyelet. Leírja azokat a módszereket, amelyek biztosítják a dokumentált és ellenőrzött verziók fenntartását, tárolását és biztosítását a teljes életciklus alatt. Ez a konfiguráció menedzselési tervhez tartozhat, ekkor egy megfelelő hivatkozást kell megadni.

11. Média felügyelet. Leírja azokat a berendezéseket és módszereket, amelyeket annak érdekében kell használni, hogy a szoftvert hol és hogyan tároljuk, másoljuk, illetve az engedély nélküli hozzáférést, szándékos károkozást megakadályozzuk. Ez is része lehet a konfiguráció menedzselési tervnek.

12. Alvállalkozói felügyelet. Az alvállalkozók által készített szoftver minőségbiztosítással kapcsolatos kérdéseit szabályozza. Elkészült szoftver esetén azt kell vizsgálni, hogy az mennyiben felel meg a minőségbiztosítási terv előírásainak, fejlesztés alatt álló program esetén garantálni kell, hogy az a minőségbiztosítási terv betartásával készüljön.

13. Dokumentáció gyűjtés, karbantartás, megőrzés. Meg kell határozni a megőrzendő minőségbiztosítási dokumentumokat, és megadni az eljárásokat ezek begyűjtéséhez, karbantartásához, tárolásához. Két típusú dokumentumról lehet szó: azok, amelyek azt mutatják, hogy a termék megfelel a szerződésben foglaltaknak; illetve referencia adatok a vállalaton belüli hosszú távú folyamatok felismeréséhez.

14. Kiképzés. Meg kell határozni azokat a képzéseket, amelyek a minőségbiztosítási terv megvalósításához szükségesek. Egy mátrix készíthető, amely tartalmazza a feladatok ellátásához szükséges, és a személyzet által elsajátított ismereteket. Ez lehetővé teszi az egyének és a szükséges ismeretek gyors azonosítását, összekapcsolását.

15. Kockázatok kezelése. Itt kell leírni a kockázatok azonosítására, értékelésére, kezelésére alkalmazandó eljárásokat, módszereket. Értékelni kell az egyes kockázatok nagyságát és kihatását.

2.2. CMM

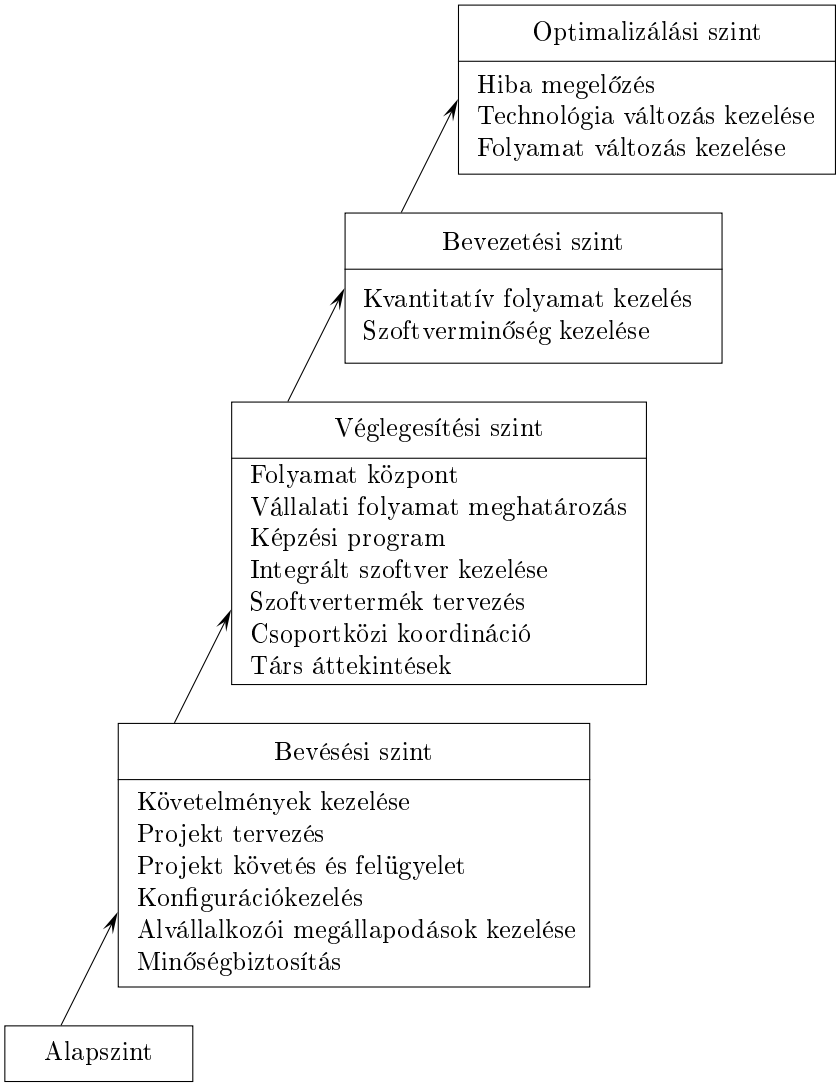
Az USA védelmi minisztériuma (DoD) a Carnegie-Mellon University-n megalapította a Software Engineering Institute-t (SEI), amely a szoftvertechnológiák terjesztését tekinti fő feladatának. Az intézet elsődleges feladata a DoD által finanszírozott projektekbe beszállító szoftver vállalatok lehetőségeinek a javítása volt. A SEI az 1980-as években kezdte tanulmányozni, miként lehet a képességeket felbecsülni. Ennek a munkának az eredménye a SEI Software Capability Maturity Model (CMM), képesség fejlettségi modell, amely a vállalatok öt lehetséges fejlettségi szintjét határozta meg. A CMM és az ISO 9000 nemzetközi

minőségbiztosítási szabvány között a következő kapcsolat áll fenn: a területek túlnyomó többsége egymásnak egyértelműen megfeleltethető, ugyanakkor a CMM részletesebb, normatívabb, és megad egy keretrendszert a folyamat továbbfejlesztésére.

Az első verzióval kapcsolatban felmerült, hogy abban túl pontatlanok a fejlettségi szintek, ezért a kezdeti tapasztalatok alapján 1993-ban átdolgozták a modellt. Ebben a második változatban is megmaradt az öt fejlettségi szint, de ezeket sokkal pontosabban, részletesebben határozták meg kulcsfolyamat területek segítségével.

A modell szintjei és kulcsfolyamatai a következők (2.1. ábra).

1. *Alapszint.* A vállalatnak nincs hatékony vezetési eljárása, projekttervet sem készítenek. Ha esetleg alkalmaznak is formális irányítási eljárásokat, azok következetes használata, illetve ellenőrzése nem biztosított. Lehetséges a sikeres szoftverfejlesztés, de a minőség és a fejlesztési folyamat (költség, határidő) kiszámíthatatlan.



2.1. ábra. A CMM szintjei és kulcsfolyamatai

2. *Bevésési szint.* A vállalat sikeresen valósíthat meg azonos típusú projekteket. Használnak formális vezetési, minőségbiztosítási és konfigurációkezelési eljárásokat, de nincs formális folyamatmodell. A projekt sikere az egyéni vezetőktől és a vállalati szokásoktól függ.

   2.1. *Követelmények kezelése*:

   - a megállapított követelmények dokumentálása;
   - a megállapított követelményeket felülvizsgálja a projektvezető és az érintett csoportok (tesztelési, tervezési, minőségbiztosítási, konfigurációkezelési, dokumentációs csoport);
   - megváltoztatott követelmények esetén a tervet, a tárgyi feltételeket, és az eljárásokat módosítani kell, hogy azok megfeleljenek a változtatásoknak.

   2.2. *Projekt tervezés*:

   - a megállapított követelményeken alapul a terv, és azok megvalósítására kötelezettséget vállal;
   - a kötelezettségek vállalásában megállapodik a projekt vezetése, és egyeztet a rendszer, hardver és tesztelés felelőseivel;
   - az előzőekben érintett csoportok felülvizsgálják a rendszer nagyságának, erőforrásigényének, költségének és ütemezésének becslését;
   - a terv menedzselt és felügyelt.

   2.3. *Projekt követés és felügyelet*:

   - a projekt követés alapja a fejlesztési terv;
   - a menedzser ismeri a projekt állapotát;
   - ha a terv teljesítése veszélybe kerül, akkor korrigálásra lehet szükség, amelyet a teljesítmény növelésével, vagy a terv megváltoztatásával lehet elérni;
   - a kötelezettség vállalások módosítása az érintett csoportok bevonásával történik.

   2.4. *Konfigurációkezelés*:

   - a felelőse minden projekt esetén meghatározott;
   - a rendszer teljes életciklusa alatt használják;
   - egyaránt alkalmazni kell a kiadott, a belső és a felhasznált (fordítóprogram) szoftverre;
   - a konfigurációs tételeket egy projekthez tartozó adattárban kell tárolni;
   - az alapverziókat és a konfigurációkezelés tevékenységeit meghatározott időközönként ellenőrizni kell.

   2.5. *Alvállalkozói megállapodások kezelése*:

   - a projektnek követnie kell a dokumentált vállalati irányelveket az alvállalkozói szerződések tekintetében;
   - az alvállalkozói szerződések irányítója felelős az alvállalkozók kiválasztásáért, a szerződések kezeléséért, a kiadott (kész)termék alvállalkozói támogatásáért (a karbantartás alatt is).

   2.6. *Minőségbiztosítás*:

   - minden projektnek tartalmaznia kell minőségbiztosítási tevékenységeket;
   - a minőségbiztosítási csoport a felső vezetésnek a projektvezetéstől, a fejlesztői csapattól és más szoftverhez kapcsolódó csoportoktól függetlenül jelent;

- a felső vezetés meghatározott időközönként felülvizsgálja a minőségbiztosítási tevékenységeket és eredményeket.

3. *Véglegesítési szint.* A vállalat definiálja a folyamatait, amelyek lehetővé teszik a minőségi folyamat továbbfejlesztését. Formális eljárásokat használnak annak ellenőrzésére, hogy minden projektben a definiált folyamatot alkalmazzák-e.

   3.1. *Folyamat központ*:

   - a vállalat követi a leírt irányvonalat a fejlesztési folyamat javítására és továbbfejlesztésére;
   - ezt az irányvonalat a felső vezetés támogatja;
   - a fejlesztési folyamat továbbfejlesztését és javítását felülvizsgálja a felső vezetés.

   3.2. *Vállalati folyamat meghatározás*:

   - a fejlesztési folyamat vállalati szabványként definiált;
   - a projektek fejlesztési folyamatai a vállalati szabvány testre szabott változatai;
   - a szabványfolyamat eszközeit karbantartják;
   - a projektekben a fejlesztési folyamattal kapcsolatos információkat össze kell gyűjteni a szabványfolyamat javítására.

   3.3. *Képzési program*:

   - minden vezetési és technikai szerephez szükséges jártasságot és tudást azonosítani kell;
   - meg kell állapítani a képzési formákat;
   - a képzés célja a vállalat tudásbázisának növelése, a projektek szükségleteinek a biztosítása, a munkatársak jártasságának és tudásának növelése;
   - a képzést vállalaton belül, illetve a szükséges külső segítséggel lehet megoldani.

   3.4. *Integrált szoftver kezelése*:

   - minden projekt dokumentálja a saját fejlesztési folyamatát (a szabvány testre szabott változatát);
   - a szabványtól való eltérést jóvá kell hagyni, majd dokumentálni kell;
   - minden projekt a meghatározott folyamatot követi;
   - minden projekt összegyűjti a projektre jellemző mérési adatokat, és azokat a vállalat fejlesztési folyamat adatbázisában tárolja.

   3.5. *Szoftvertermék tervezés*:

   - a feladatokat a fejlesztési folyamatnak megfelelően hajtják végre a fejlesztők;
   - a fejlesztés és a karbantartás során megfelelő módszereket és eszközöket használnak;
   - a szoftver, a feladatok, az egyes tárgyi tételek követhetőek a megállapított követelmények alapján.

   3.6. *Csoportközi koordináció*:

   - a rendszer követelményeit és a projekt szintű célokat az érintett csoportok (l. korábban) közreműködésével állapítják meg;

- a fejlesztő csoportok koordinálják a terveiket és a tevékenységeiket;
- a menedzserek felelősek egy együttműködést és koordinációt támogató környezet létrehozásáért és fenntartásáért.

3.7. *Társ áttekintések*:

- a vállalat azonosítja a felülvizsgálandó tárgyi tételeket;
- minden projekt azonosítja az ilyen típusú tárgyi tételeit;
- a felülvizsgálatokat gyakorlott vezetők irányítják;
- a felülvizsgálat fókusza a terméken, és nem az előállítón van;
- a felülvizsgálatok eredményeit a menedzserek nem használják fel az egyéni teljesítmények értékelésére.

4. *Bevezetési szint.* A vállalat rendelkezik definiált folyamattal, és formális eljárással gyűjtik a számszerűsített adatokat. A folyamat- és termékmetrikákat felhasználják a folyamat javítására.

4.1. *Kvantitatív folyamat kezelés*:

- minden projekt követi a vállalati irányelveket a projektfolyamat teljesítményének mérésére és számszerű követésére;
- a vállalat követi a leírt irányvonalat a vállalati szabványfolyamat fejlettségének elemzésére.

4.2. *Szoftverminőség kezelése*:

- a minőségkezelés tevékenységei támogatják a vállalat minőségjavítási elkötelezettségét;
- minden projekt definiálja és összegyűjti a minőség mérésének eredményeit;
- minden projekt meghatározza a minőségi célokat, és ellenőrzi a célok felé haladás mértékét;
- a minőségkezelési feladatok meghatározottak.

5. *Optimalizálási szint.* A vállalat kötelezi magát a fejlesztési folyamat javítására. A fejlesztési folyamat javítása része a vállalat tevékenységének, tervezik, szerepel a költségvetésben.

5.1. *Hiba megelőzés*:

- a vállalat egy leírt irányvonalat követ a hiba megelőzésben, amelyet a korábbi hibák okait elemezve állítanak elő;
- minden projekt egy leírt irányvonalat követ a hiba megelőző tevékenységében.

5.2. *Technológia változás kezelése*:

- a vállalat egy leírt irányvonalat követ a technológiai fejlettség javítására;
- a felső vezetés támogatja ezt az irányvonalat;
- a felső vezetés felelősséget vállal ezért a tevékenységért.

5.3. *Folyamat változás kezelése*:

- a vállalat egy leírt irányvonallal rendelkezik a fejlesztési folyamat javításának megvalósítására;
- a felső vezetés támogatja ezt az irányvonalat;
- a felső vezetés felelősséget vállal ezért a tevékenységért.

A CMM problémái a következők.

- A modell csak a projektvezetéssel foglalkozik, a termék fejlesztésével nem. A vállalatok által használt technológiákat sem veszi figyelembe. (A technológiák kapcsán az alkotók elismerték, hogy ez azért maradt ki, mert egyetlen szabványos módszert sem találtak a technológia használatának becslésére.)
- Nem tartalmazza a kockázat kezelését, mint kulcsfolyamatot.
- A modell alkalmazhatósági területe nincs meghatározva. A szerzők is tudták, hogy a modell nem alkalmazható minden vállalatra, de nem adták meg, melyik esetben alkalmazható és mikor nem. Kisebb vállaltok esetén például a modell túl bürokratikus.
- A modell védelmi rendszerek szoftverfejlesztésével foglalkozik, ugyanakkor nagy a különbség a védelmi és a kereskedelmi szoftverek fejlesztése között.

A CMM pozitív hatása, hogy a szoftverfejlesztő cégek számára tudatosult, hogy mit lehet, illetve kell tenni a minőség növelésének érdekében. Az USA-ban a legtöbb vállalat legalább a harmadik szint elérésére törekszik.

Nem szabad elfelejteni, hogy a modellt az USA Nemzetvédelmi Minisztériumának szánták, hogy fel tudja becsülni az egyes szoftver szállítók képességeit. Ugyanakkor nincsenek olyan nyilvános adatok, amelyek a szállítóknak egy adott fejlettségi szint elérését írnák elő, de feltételezések szerint magasabb szinten álló vállalatok előnyösebb helyzetben vannak. (Várhatóan a harmadik szint teljesítése elvárás lesz.)

3. Architektúra

Az architektúra a rendszerterv kritikus része. A rendszer elemeinek, komponenseinek együttműködésére, közös viselkedésére koncentrál, de nem foglalkozik az implementáció részleteivel, az adatok ábrázolásával, az algoritmusokkal. Az architektúra megmutatja az elemek egymásra hatását elhagyva azokat az információkat, amelyek nem tartoznak ehhez a nézethez. A szoftver architektúra eléggé rugalmas és tág fogalom, ezért nincs szabványos, mindenki által elfogadott definíciója. Egy lehetséges meghatározás a következő.

A szoftver architektúrája a rendszer olyan szerkezete, amely tartalmazza a szoftver elemeket, azok kívülről látható tulajdonságait, illetve az egymás közötti kapcsolataikat.

Egy elem kívülről látható tulajdonságai meghatározzák, hogy más elemek miként működhetnek vele együtt. Ilyen tulajdonság lehet például, hogy milyen szolgáltatásokat nyújt, hogyan kezeli a fellépő hibákat, miként használja az osztott erőforrásokat.

Az architektúra meghatározása a magas szintű tervezési folyamat része. Az architektúra megadja a legfontosabb tervezési döntéseket és azok következményeit, amelyek kihatnak a teljes előállítási folyamatra, a termékre és a karbantartásra. Az architekturális döntések helyessége kulcsfontosságú, mert ezeket igen nehéz, gyakran lehetetlen a későbbi fázisokban módosítani.

Helyes architektúra választása lehetővé teszi a meghatározott minőségi attribútumok elérését, ugyanakkor nem garantálja azt. Például nagy teljesítményű rendszer esetén figyelmet kell fordítani az elemek közötti adatáramlás sebességére; magasfokú biztonság esetén az elemek közötti kommunikációt kell védetté tenni, illetve korlátozni, hogy melyik elem milyen információt érhet el. Ezeket a szempontokat már az architektúra kialakításakor figyelembe kell venni.

Az architektúra elkészítését támogatják az architekturális minták. Ezek a minták egyben meghatározzák bizonyos minőségi jellemzők elérésének könnyedségét vagy nehézségét. A következő minőségi jellemzőket vizsgáljuk az architekturális minták kapcsán.

Rendelkezésre állás: A rendelkezésre állás a rendszer hibáival, illetve azok következményeivel kapcsolatos tulajdonság. Hiba akkor lép fel, ha a rendszer nem biztosítja a szolgáltatások és azok specifikációi közötti konzisztenciát. Kétféle hibát különböztetünk meg: failure és fault. A failure típusú hibákat a felhasználó képes érzékelni, a fault típusúakat nem. Ha egy fault megfigyelhetővé válik, mert nem korrigálta a rendszer, akkor failure lesz belőle. A rendszer rendelkezésre állása annak a valószínűsége, hogy a rendszer működőképes akkor, amikor a szolgáltatására szükség van.

Módosíthatóság: A rendszer változtathatóságát méri, a változtatás költségének függvényében. Platform módosítás esetén hordozhatóságról beszélünk.

Teljesítmény: Ez leggyakrabban időkorlátozást jelent, azaz a rendszernek bizonyos eseményekre adott időn belül válaszolnia kell.

Biztonság: Ez azzal mérhető, hogy a rendszer mennyire képes ellenállni az illetéktelen behatolásoknak, miközben a jogosult felhasználók számára a megfelelő szolgáltatásokat

nyújtja. A biztonság megsértése jelenthet jogtalan adathozzáférést vagy jogosulatlan szolgáltatás használatot.

Tesztelhetőség: Annak érdekében, hogy a rendszer jól tesztelhető legyen, képesnek kell lennünk az összes komponens belső állapotának és bemenetének irányítására, illetve a kimenetének megfigyelésére. Figyelembe véve, hogy a költségek legalább 40 százalékát tesztelésre fordítják, az architekturális szintű támogatás jelentősége igen nagy.

Használhatóság: Ez a kritérium azt vizsgálja, hogy a felhasználók mennyire kényelmesen tudják igénybe venni a rendszer szolgáltatásait. Ez több részből áll: a rendszer használatának elsajátítása, hatékony használat támogatása, hibák minimalizálása, alkalmazkodás a felhasználó igényeihez, magabiztosság növelése (jelezni a felhasználónak, hogy a művelet még folyik, illetve megfelelően végrehajtásra került).

A továbbiakban áttekintünk néhány architekturális mintát.

3.1. Csövek és szűrők

A minta egy adatfolyam feldolgozására képes szerkezetet ír le. Minden komponens rendelkezik bemenettel és kimenettel. A komponens a bemenetről olvassa az adatokat, feldolgozza, transzformálja azokat, és az eredményt a kimeneten adja meg. A transzformáció folyamatos, azaz a kimeneten még a bemenet teljes feldolgozása előtt megjelennek adatok. Egy ilyen komponenst *szűrőnek* nevezünk. A szűrők között az adatokat *csövek* továbbítják. A csövek elrendezése és száma tetszőleges.

A minta alapvető jellemzője, hogy a szűrők egymástól függetlenek, azaz nem oszthatják meg belső állapotukat, és nem tételezhetnek fel semmit a körülöttük levő komponensekről. Ugyanakkor egy szűrő specifikációjában megadható, hogy az milyen bemenet fogadására képes, illetve milyen kimenetet állít elő.

A szűrőket adatmozgás szerint két csoportba oszthatjuk:

- egy passzív szűrő bemenő adatait a megelőző szűrők nyomják tovább, kimenő adatait a rákövetkező szűrők vonják ki;
- egy aktív szűrő képes adatot kérni és/vagy továbbítani.

Ha két aktív szűrő közvetlenül kapcsolódik, akkor vagy az összekötő cső szinkronizálja azokat, vagy az egyik szűrő felelős a kommunikációért.

A minta speciális esete a csővezeték (pipeline), amelyben a szűrők elrendezése lineáris. Ennek egy további speciális változata a kötegelt szekvenciális csővezeték (batch sequential pipeline), amelyben minden szűrő felelős a transzformált bemenet továbbításáért, ami csak akkor következik be, ha a transzformáció befejeződött, teljes. Ekkor a csövek szerepe jelentéktelen.

A minta támogatja a módosíthatóságot, hiszen a szűrők függetlensége miatt, azok könnyen cserélhetőek, módosíthatóak. A rendszerhez könnyen felvehetünk újabb komponenseket. A szűrők fejlesztése, módosítása párhuzamosan történhet. A teljesítményre is jó hatással lehet a minta, ha az egyes szűrőket párhuzamosan futtatható folyamatokkal valósítjuk meg. Használhatóság szempontjából elmondhatjuk, hogy a minta nem túl jó, hiszen az interaktivitást nehéz, illetve lehetetlen benne biztosítani.

3.2. Objektumelvű rendszer

A minta komponensei objektumok, amelyek eljáráshívások segítségével kommunikálnak. Ekkor az objektumok felelősek a belső reprezentáció integritásának megőrzéséért, és ez a reprezentáció rejtett, így azt a felhasználó objektumoktól függetlenül változtathatjuk. Az objektumok önálló folyamatok lehetnek, amivel a teljesítmény növelhető.

A minta hátránya az objektumok kommunikációjához szükséges kapcsolat. Ha egy olyan objektum látható részét módosítjuk, amelyet mások használnak, a felhasználó objektumok módosítása is szükséges lehet.

3.3. Esemény alapú rendszer

Ebben az esetben a komponensek nem hívják meg közvetlenül az eljárásokat, függvényeket, hanem a komponens képes egy vagy több esemény bejelentésére, amelyekre más komponensek reagálhatnak. Ennek érdekében a komponenseknek regisztrálniuk kell, hogy milyen eseményekre kívánnak reagálni, és ha az bekövetkezik, akkor az eseményhez rendelt eljárásuk lefut.

A minta alapvető tulajdonsága, hogy a komponensek nem tudják, hogy az eseményeikre mely komponensek reagálnak, és azt sem tudják, hogy a komponensek milyen sorrendben értesülnek az eseményekről, illetve reagálnak azokra. Azt sem ismerik, hogy mi lesz a reakciók eredménye.

A minta támogatja a módosíthatóságot, hiszen könnyű komponenseket módosítani, beilleszteni, eltávolítani a többi komponens minimális változtatásával. A komponensek újra felhasználhatóak, mert csak az eseménykezelőben kell a regisztrációt elvégezni. A tesztelhetőség nehézkes, mert az irányítást nem lehet felügyelni. A rendelkezésre állást is negatívan befolyásolhatja, hiszen nem ismert, hogy egy eseményre válaszoló egységek miként viselkednek, mikor fejeződnek be.

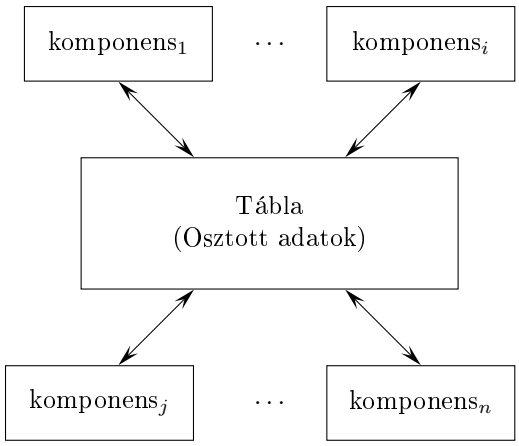
3.4. Réteg szerkezetű rendszer

Az előző félévben megismert architektúra, amelyben az egyes komponensek egymásra épülő világokat alkotnak. Egy réteg csak az alatta levő rétegek szolgáltatásait veheti igénybe szabványos felületen (protokollon) keresztül.

A minta támogatja a módosíthatóságot, hordozhatóságot, hiszen minden réteg protokollt használ a kommunikációhoz, így egy réteg a többitől függetlenül módosítható. Ha változik a felület, akkor is legfeljebb csak a kapcsolódó (zárt esetben szomszédos) rétegekre terjed ki a változtatás. A rétegek egymástól függetlenül működtethetőek (az alsóbb rétegek esetleges szimulációjával), így a tesztelés is támogatott. A rétegeknek védelmi szinteket is megfeleltethetünk, így a biztonságot is támogatja a minta. A kritikus rendszereket a belső rétegekbe helyezzük el, amelyeket védenek a külső rétegek. Zárt rétegszerkezet esetén a rétegek közötti kommunikáció a teljesítményt csökkenti, ugyanakkor ha ettől nagymértékben eltérünk, akkor az előnyökből is veszítünk.

3.5. Gyűjtemény

A minta két eltérő típusú komponenst tartalmaz. Az adattár egy központi adatszerkezet, amely reprezentálja a rendszer állapotát. Ezt független külső komponensek veszik körül,



3.1. ábra. A tábla minta

amelyek használják a központi adattárat. A minta két alesetét különböztethetjük meg az adattár és a külső komponensek közötti kapcsolat alapján.

Adatbázis A komponensek közvetlenül az adattáron végeznek műveleteket, de az adattár nem hajt végre műveletet azokon.

Tábla (Blackboard) Az adattár az állapotának függvényében képes végrehajtani műveleteket a komponenseken.

Tábla esetén a külső komponensek egymástól függetlenek, az adattáron hajtanak végre műveleteket. Ha az adattár állapota megváltozik, minden komponensnek azonnal alkalmazkodnia kell ehhez. A rendszer ilyen módosításokkal fokozatosan éri el a célállapotot (3.1. ábra).

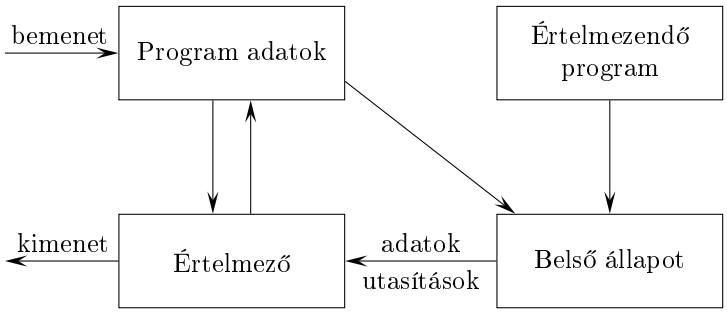
A mintára egy lehetséges példa egy CASE eszköz, amelyben a nézetek (használati, logikai, komponens, környezeti) segítségével valósítjuk meg a rendszert, és az adatokat egy központi adattárban helyezzük el. Az adattár változása esetén az összes nézet módosul az adattár alapján.

A minta támogatja biztonságot és a rendelkezésre állást, mert ezek biztosításáért az adattár felelhet. Az adatok egy helyen találhatóak, így gyorsan elérhetőek, ami a teljesítmény szempontjából pozitív. A módosíthatóság szempontjából is kedvező a helyzet a külső komponensek tekintetében, mert azokat könnyen változtathatjuk, újakat vehetünk fel. Ugyanakkor az adattár módosítása rendkívül összetett.

3.6. Virtuális gép, értelmező

A minta szerkezetét követő rendszerben a feladatot egy adott nyelven fogalmazzuk meg, és a megoldást a nyelv értelmezésével kapjuk meg. A minta négy komponenst tartalmaz (3.2. ábra):

1. értelmező (interpreter, state machine, execution engine),
2. belső állapot (internal state, stack),
3. értelmezendő program (utasítások az adott nyelven),



3.2. ábra. A virtuális gép architekturális minta elemei és azok kapcsolatai

4. program adatok (program state), például változók.

A minta legnagyobb előnye, hogy a virtuális gépen működő programok hordozhatóak. Hátránya, hogy a natív kódra fordítás jelentős futási idejű költségekkel járhat, illetve a korlátozott erőforrások miatt nem használható virtuális gép.

3.7. Modell–Nézet–Vezérlő

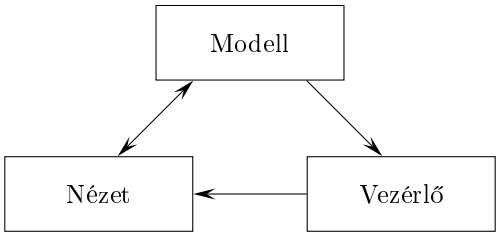
A minta interaktív alkalmazás fejlesztését kívánja támogatni olyan rugalmas szerkezettel, amelyben a felhasználói felületet érintő változtatások egyszerűen megvalósíthatóak. A minta lényege, hogy az adatokat és a rajtuk végzett műveleteket elválasszuk egymástól. Ezt három fő komponens segítségével valósítja meg (3.3. ábra)

1. *Modell.* Tartalmazza az adatokat és elvégzi azokon a műveleteket. Független a bemenettől és a kimenettől. Értesíti a nézet(ek)et, ha az adatok megváltoznak.

2. *Nézet.* A kimenet megjelenítésére szolgál, ehhez az adatokat a modell szolgáltatja.

3. *Vezérlő.* A felhasználótól származó bemenet kezeléséért felel. Ezt rendszerint esemény alapú mechanizmussal valósítja meg. Egy esemény kezelése során a modell, vagy a nézet szolgáltatásait veszi igénybe.

Hordozható rendszerek esetén a grafikus felületnek is követnie kell az eltérő platformok által biztosított különböző szabványokat. A nézet különválasztásával a minta ezt a követelményt támogatja. A használhatóság növelése érdekében lehetővé kell tenni, hogy a felhasználó személyre szabhassa a felületet. A minta ezt is támogatja. Módosíthatóság



3.3. ábra. A modell–nézet–vezérlő architekturális minta vázlata

szempontjából elmondhatjuk, hogy a meglévő komponensek újra felhasználhatóak, ugyanis egy elkészült nézettel kiegészíthetünk egy új modellt.

A minta hátránya, hogy a rendszer viszonylag bonyolulttá válhat. Sok adat és sok nézet esetén a teljesítmény csökkenhet, ha egy adat megváltozása esetén olyan nézeteket is értesít a modell, amelyeket a változás nem befolyásol.

3.8. Heterogén architektúrák

Rendszerint egyetlen architekturális minta önmagában nem ad kellő alapot az előírt minőségi jellemzőkkel rendelkező rendszer szerkezetének kialakításához. Ezért a mintákat kombinálva használják, mérlegelve az egyes elemek előnyeit és hátrányait. A heterogenitást a következő módokon érhetjük el.

- Egy architekturális minta komponenseinek belső szerkezete egy másik architekturális mintának felel meg. Ez az eljárás tetszőleges mélységben folytatható.
- Egy minta egy vagy több komponensének megengedjük, hogy egy másik mintának megfelelően is tudjon kommunikálni más komponensekkel.

4. Objektumelvű tervezés és tervminták

Objektumelvű rendszer tervezése nehéz és bonyolult feladat, különösen újrafelhasználható terv esetén.

Újrafelhasználható terv jellemzői:

- meg kell felelnie a konkrét probléma sajátosságainak;
- elég általánosnak is kell lennie ahhoz, hogy más, később felmerülő feladatok megoldása során is alkalmazható legyen;
- elkerülve, vagy minimalizálva, az esetleges újratervezést.

Jó tervek létrehozásához nagy gyakorlatra van szükség. Miért van az, hogy kezdő szakemberek rendszerint nem tudnak olyan jó, újrafelhasználható terveket készíteni, mint a nagy gyakorlattal rendelkezők?

Ennek egyik legfontosabb oka, hogy a tapasztalt tervezők rendszerint bevált tervek alapján hozzák létre egy új rendszer tervét. Egy-egy jó tervet használnak újra meg újra, ezek ismerete és használata teszi őket jó szakemberekké.

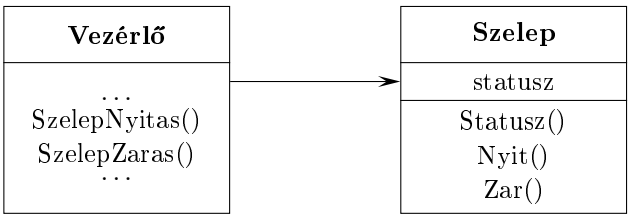
Ezeket a terveket, tervrészeket nevezzük *tervmintáknak* (design patterns). Hasonló módon használhatóak a tervezésben, mint ahogy a programozás során a kód egyes részeit kódújrafelhasználás segítségével állítjuk elő. A gyakorlatban bevállt tervmintákat gyűjtötték össze és katalogizálták Gamma és társai és jelentették meg 1995-ben[1]. A tervmintákat ezen könyv alapján mutatjuk be a következőkben.

4.1. Rossz tervek

- „Mindenható osztály": az osztály hajtja végre majdnem az összes teendőt, a többi osztálynak legfeljebb támogató műveleteket hagy. Az osztály lényegében egy bonyolult és összetett vezérlő osztály (gyakran ez is a neve), amelyet egyszerű osztályok vesznek körbe. Ezen osztályoknak csak adattárolási funkciójuk van (csak `get` és `set` műveletek).
- „Mindentudó osztály": tulajdonképpen az előző osztály egy változata, csak ez nem az összes tevékenységet hajtja végre, hanem az összes adatot tartalmazza. Ekkor a többi osztály innen nyeri ki (ezen osztály `get` és `set` műveleteivel) a műveletekhez szükséges adatokat.

Példa „mindenható osztályra" az alábbi vezérlő:

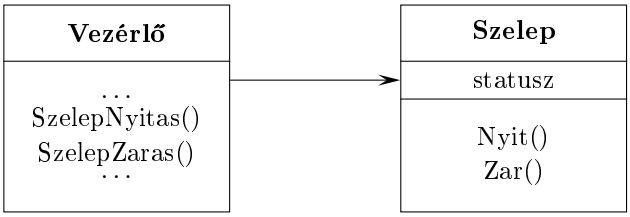
[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Inc., 1995, ISBN-0-201-63361-2



Ekkor a szelep nyitása művelet:

```
void SzelepNyitas()
{
    int    allapot;
    allapot = sz->Statusz();
    if ( allapot != nyitva )   sz->Nyit();
}
```

A következő lenne egy helyes megoldás:




```
void SzelepNyitas()
{
    sz->Nyit();
}

void Nyit()
{
    if ( statusz != nyitva )   statusz = nyitva;
    ...
}
```

4.2. Tervminta

Általában egy tervminta a következő négy alapvető elemből épül fel:

Név Ezzel hivatkozunk a tervezési feladatra és annak megoldására. Ez rendszerint egy-két szó, amely utal a funkcióra. Lehetővé teszi, hogy a tervezést magasabb absztrakciós szinten végezzük.

Feladat Itt le kell írni, hogy milyen esetekben alkalmazható a minta. A leírás tartalmazza a problémát, annak környezetét és az esetleges peremfeltételeket.

Megoldás A minta alkotóelemeinek, azok kapcsolatainak, együttműködésüknek a leírása. Ez egy absztrakt leírás, amely az általánosság érdekében nem tartalmazza az implementációt.

Következmények A minta eredményeinek és a meghozott kompromisszumoknak (futási idő, tárkapacitás) a leírása.

4.3. Tervminták osztályozása

Létrehozási: Az osztályok, objektumok (példányok) létrehozására, előállítására vonatkozó minták. Az osztályokra vonatkozó esetben az öröklődés felhasználásával érjük el, hogy különböző osztályt példányosítsunk. Az objektumok esetében a példány létrehozását egy másik objektumra delegáljuk.

Szerkezeti: Ezek a minták arra szolgálnak, hogy osztályokból vagy objektumokból nagyobb szerkezeteket hozzunk létre. A létrejött elemek rendszerint új funkcionalitással is rendelkeznek.

Viselkedési: Az objektumok közötti kapcsolatokkal, vezérlési folyamatokkal kapcsolatos minták. Használatukkal az objektumok kapcsolatára kell figyelni, a vezérlés folyama a mintában van. Az öröklődés felhasználásával érik el, hogy a viselkedést szétosszák különböző osztályok között.

4.4. Tervminták megadása

A következőkben összefoglaljuk, miként adhatunk meg egy-egy tervmintát. Az UML diagram ugyan fontos eleme ennek, de korántsem elégséges önmagában. A mintákat a megoldandó feladat alapján osztályokba soroljuk (pl.: létrehozási, szerkezeti, viselkedési), és ezt szintén fel kell tüntetnünk a meghatározás során.

1. *A minta neve és osztálya*: egy jó névnek utalnia kell a felhasználás területére, a megoldott feladatra.
2. *Cél*: rövid leírása annak, hogy mi a minta által megoldott feladat vagy feladatosztály.
3. *Más nevek*: további, mások által használt nevei a mintának, ha van ilyen.
4. *Motiváció*: egy esettanulmány, amely bemutatja a tervezési feladatot, és hogy miként oldja meg azt a minta a benne szereplő osztályok és objektumok segítségével.
5. *Felhasználhatóság*: annak leírása, hogy milyen esetekben lehet a mintát alkalmazni.
6. *Szerkezet*: itt kell megadni a megfelelő UML diagramot vagy diagramokat.
7. *Elemek*: a mintában előforduló osztályok, objektumok és szerepeik felsorolása.
8. *Együttműködés*: annak bemutatása, hogy a minta elemei miként működnek együtt a szerepük megvalósítása érdekében.
9. *Következmények*: itt kell választ adni azokra a kérdésekre, hogy miként éri el a minta a célját; milyen kompromisszumok árán; a rendszer szerkezetének milyen összetevőit lehet függetlenül változtatni.
10. *Implementáció*: itt kell megadni az implementációval kapcsolatos észrevételeket, megjegyzéseket, ajánlásokat, megszorításokat.

11. *Példa kód*: kódrészletek, amelyek bemutatják, miként lehet a minta egyes elemeit egy adott nyelven megvalósítani.

12. *Ismert használat, esettanulmány*: példák a minta előfordulásaira működő rendszerekben.

13. *Rokon minták*: a hasonló minták nevei, a fontosabb különbségek felsorolása, illetve annak a megadása, hogy mely más mintákkal célszerű együtt használni.

A tervminta megadásának elemei közül néhány (10-13) értelemszerűen elmaradhat.

Az, hogy mely terveket tekintünk tervmintáknak, relatív fogalom. Az egyetlen követelmény, hogy az többször felhasználható legyen. Ezen belül egy adott fejlesztői közösség dönthet. Ugyanakkor léteznek jól bevált tervminták.

5. Figyelő

Név, osztály: figyelő (observer); viselkedési (behavioral).

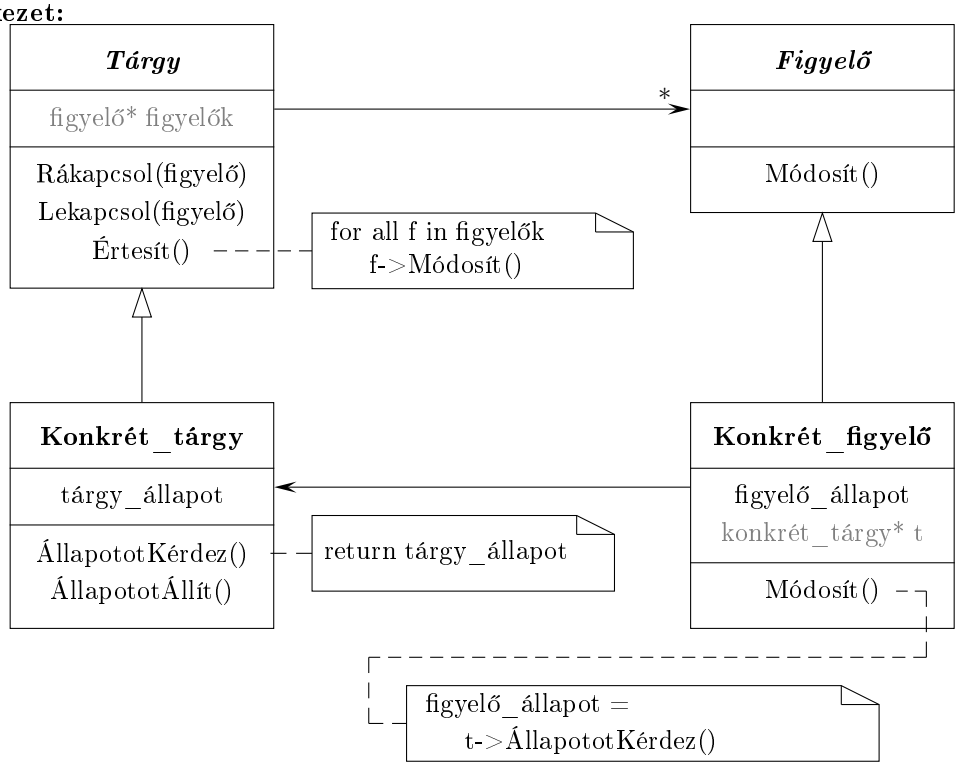
Cél: Olyan egy-több függőség megadása objektumok között, amelyben ha egy objektum állapotot vált, akkor az összes tőle függő objektumot értesíteni és módosítani kell.

Más nevek: dependents, publish-subscribe.

Felhasználhatóság: A figyelő tervminta használható az alábbi esetekben:

- Ha egy absztrakcióban két olyan tényező szerepel, amelyek közül az egyik függ a másiktól. Ha ezeket külön objektumoknak tekintjük, akkor lehetséges egymástól független változtatásuk és újrafelhasználásuk.
- Amikor egy objektum állapotának megváltoztatása más objektumok változtatását teszi szükségessé, és nem ismert ezen objektumok száma.
- Amikor egy objektumnak üzenetet kell küldenie más objektumoknak, amelyekről nem tehetünk fel semmit; azaz nem akarjuk szorosan összekapcsolni ezeket az objektumokat.

Szerkezet:



6. Iterátor

Név, osztály: iterátor (iterator); viselkedési (behavioral).

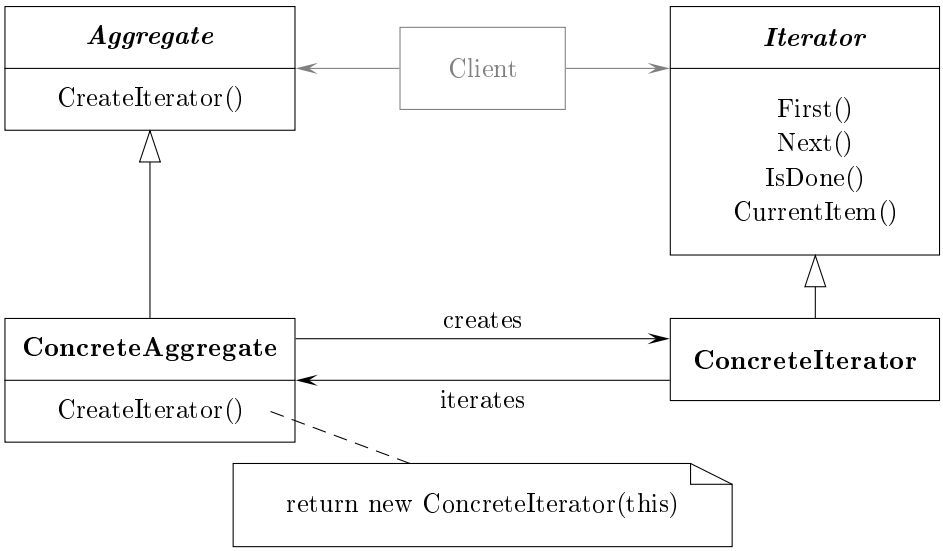
Cél: Egy aggregátum objektum elemeinek az elérését biztosítani, anélkül, hogy a reprezentációt ismernénk.

Más nevek: cursor.

Felhasználhatóság: Az iterátor minta használható az alábbi esetekben:

- Egy aggregátum elemeinek elérésére úgy, hogy nem fedjük fel a belső reprezentációt.
- Aggregátumok többféle és többszörös bejárásának támogatására.
- Különböző aggregációs szerkezetek bejárásához egy egységes felületet biztosít.

Szerkezet:



7. Állapot

Név, osztály: állapot (state); viselkedési (behavioral).

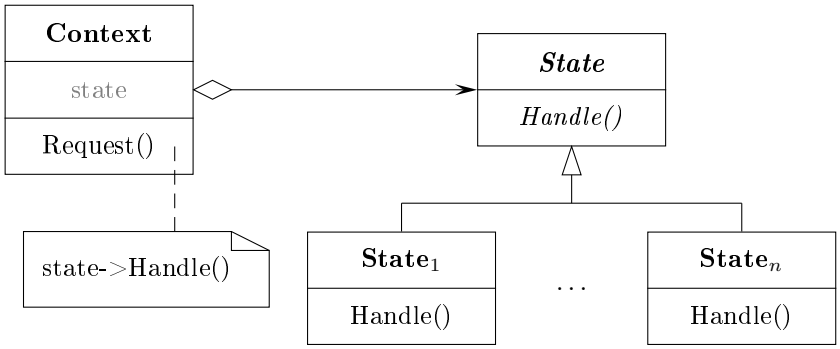
Cél: Lehetővé tenni, hogy egy objektum megváltoztassa a viselkedését, ha az állapota változik. A hatása olyan, mintha az objektum megváltoztatná az osztályát.

Más nevek: objects for states.

Felhasználhatóság: Az állapot minta használható az alábbi esetekben:

- Egy objektum viselkedését az állapota határozza meg, és a viselkedést futási időben kell megváltoztatni az állapot függvényében,
- A műveletekben nagy méretű, több részes feltételes utasítások szerepelnek, ahol a feltétel az objektum állapotától függ. Az állapotot rendszerint egy felsorolási típussal adjuk meg. Gyakran több művelet is ugyanezt a feltételes szerkezetet tartalmazza.

Szerkezet:



8. Egyke

Név, osztály: egyke (singleton); objektum létrehozási (object creational).

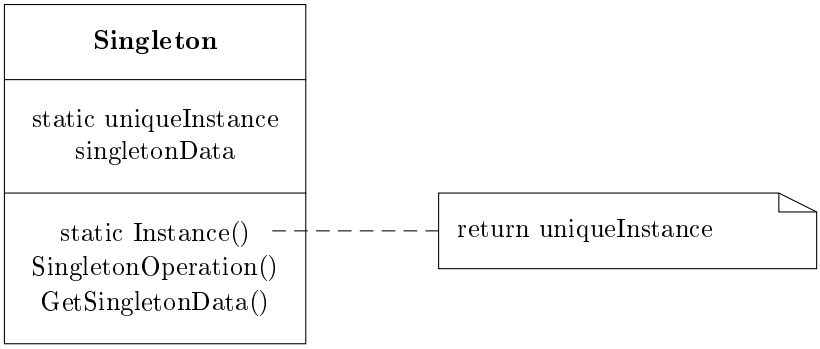
Cél: Biztosítani, hogy egy osztályhoz csak egy példány tartozhat, és megteremteni a példány globális elérhetőségét.

Más nevek: —.

Felhasználhatóság: Az egyke minta használható az alábbi esetekben:

- Egy osztálynak csak egyetlen példánya lehet, és azt a klienseknek egy előre ismert módon kell elérniük, manipulálniuk.
- Amikor az egyetlen példány kiterjeszthető származtatással, és a klienseknek egy ilyen kiterjesztett példányt kell használniuk anélkül, hogy a kódot megváltoztatnánk.

Szerkezet:



9. Közvetítő

Név, osztály: közvetítő (mediator), viselkedési (behavioral).

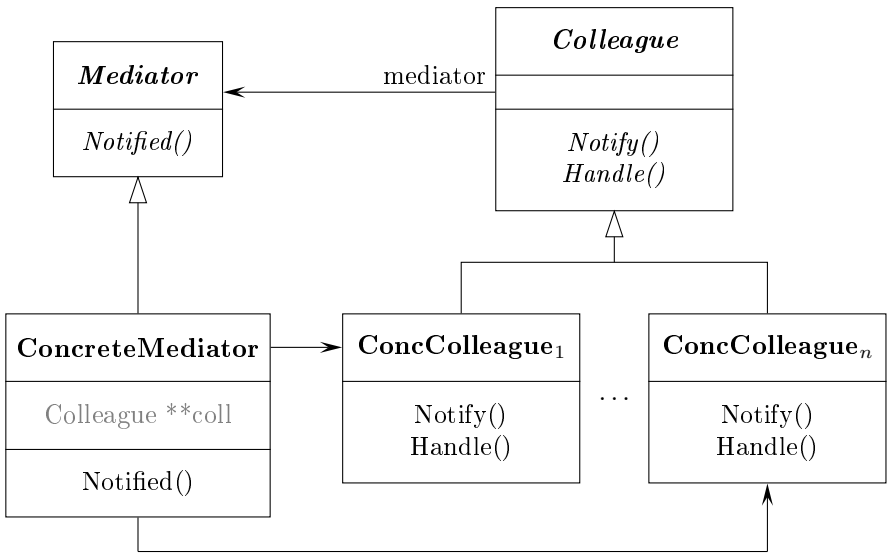
Cél: Olyan objektum megadása, amely tartalmazza, hogy objektumok egy csoportja miként működik együtt.

Más nevek: —.

Felhasználhatóság: A közvetítő minta használható az alábbi esetekben:

- Objektumok halmaza jól definiált, de összetett módon kommunikálnak; a kölcsönös függések szerkezete áttekinthetetlen, nehezen érthető.
- Egy objektum újrafelhasználása nehéz, mert sok objektumra hivatkozik, illetve sok objektummal kommunikál.
- Több osztályra szétosztott viselkedést kell megvalósítanunk túl sok származtatás nélkül.

Szerkezet:



10. Feljegyzés

Név, osztály: feljegyzés (memento), viselkedési (behavioral).

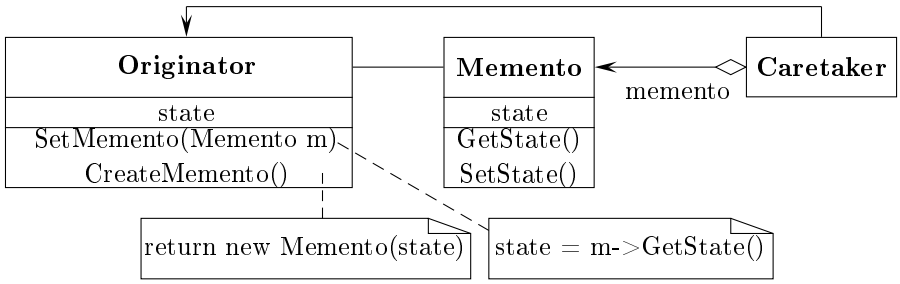
Cél: Egy objektum belső állapotának felfedése és feljegyzése anélkül, hogy az információ elrejtést megsértenénk. A feljegyzett állapot alapján az objektum állapota a későbbiekben visszaállítható.

Más nevek: token.

Felhasználhatóság: A feljegyzés minta használható az alábbi esetben:

- Egy objektum pillanatnyi állapotát (vagy egy részét) el kell mentenünk, és visszaállítanunk később, és nem akarjuk felfedni az objektum reprezentációját.

Szerkezet:



11. Parancs

Név, osztály: parancs (command), viselkedési (behavioral).

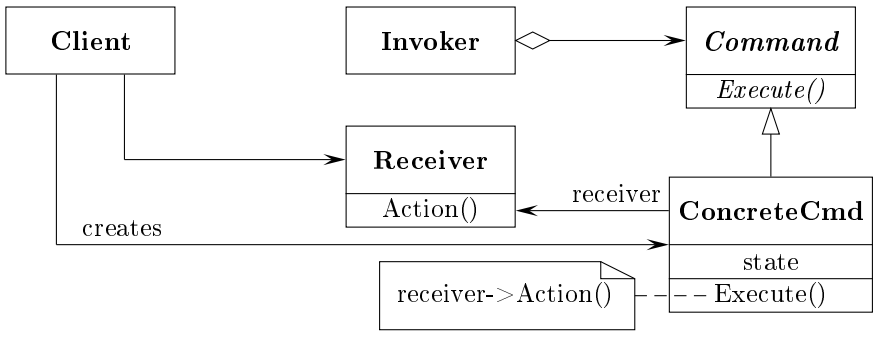
Cél: Egy igény objektumba ágyazása, így lehetővé téve, hogy a kliensek különböző igényeket adjanak ki, és ezeket tároljuk, visszavonjuk.

Más nevek: action, transaction.

Felhasználhatóság: A parancs minta használható az alábbi esetekben:

- Ha objektumokat végrehajtandó akciókkal kell paraméterezni (callback függvények).
- Ha igényeket kell meghatároznunk, sorba tennünk, végrehajtanunk különböző időpontokban. A parancs objektum élettartama független az eredeti igénytől.
- A visszavonási művelet támogatására. Ebben az esetben a végrehajtás során tárolnunk kell a megfelelő állapotot, és rendelkezni kell egy `Unexecute` művelettel is.
- Változások naplózására. A napló alapján a tevékenységek újra végrehajthatók.
- Összetett tevékenységek (tranzakciók) megvalósítására.

Szerkezet:



12. Kezelési lánc, felelősséglánc

Név, osztály: kezelési lánc, felelősséglánc (chain of responsibility), viselkedési (behavioral)

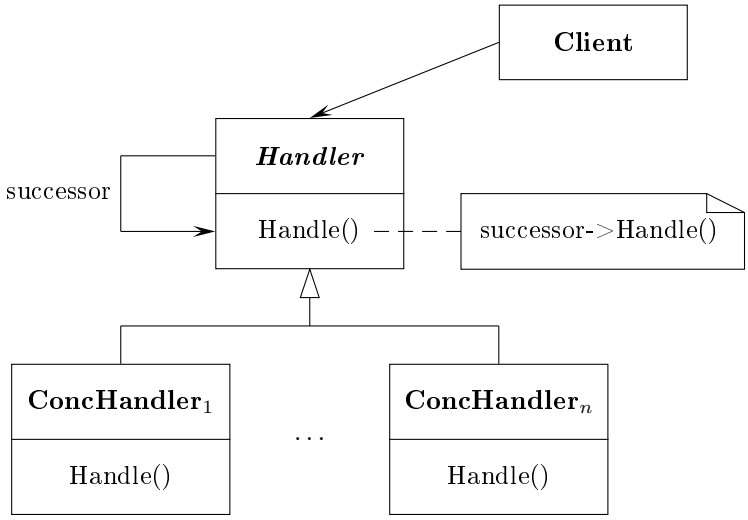
Cél: Elkerülni, hogy egy küldő objektumot és az igényt fogadó objektumot párosítsuk. Így megteremtjük annak a lehetőségét, hogy egynél több objektum is kezelhesse az igényt. A fogadó objektumokat láncba fűzzük, és az igényt a lánc mentén továbbítjuk, amíg egy objektum le nem kezeli.

Más nevek: –

Felhasználhatóság: A kezelési lánc minta használható az alábbi esetekben:

- Ha egynél több objektum kezelhet egy üzenetet, és a kezelőt nem ismerjük előre, azt automatikusan kell kiválasztanunk.
- Egy igényt számos objektum egyikének szeretnénk kiadni anélkül, hogy a fogadót explicit megadnánk.
- Egy üzenet kezelésére képes objektumok halmazát dinamikusan kell megadnunk.

Szerkezet:



13. Stratégia

Név, osztály: stratégia (strategy), viselkedési (behavioral)

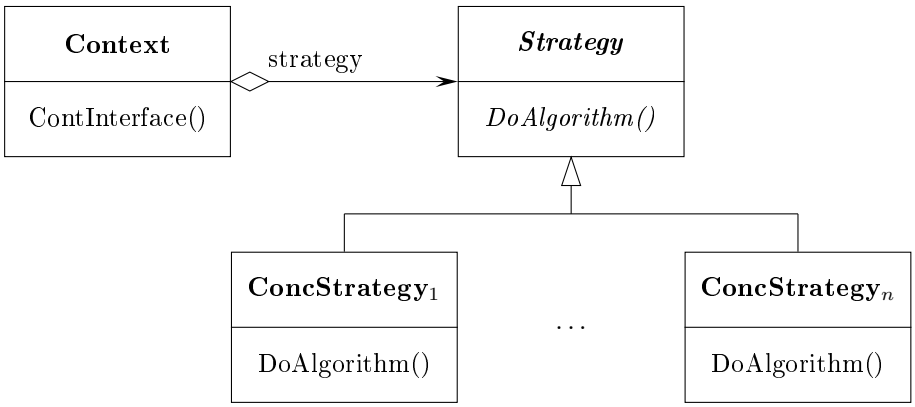
Cél: Algoritmusok halmazának létrehozása, azok beágyazása, és kicserélhetőségük biztosítása.

Más nevek: policy

Felhasználhatóság: A stratégia minta használható az alábbi esetekben:

- Több osztály csak a viselkedésében tér el. A stratégiákkal egy osztályt a viselkedések egyikével konfigurálhatunk.
- Algoritmusok különféle variációira van szükségünk.
- Az algoritmusok olyan adatokat használnak, amelyeket a klienseknek nem kellene ismerniük. A minta használatával elkerülhetjük bonyolult, algoritmus-specifikus adatszerkezetek felfedését.
- Egy osztály több viselkedést határoz meg, és ezek elágazásokkal valósíthatók meg a műveletekben. A sok elágazás helyett, az egyes ágakat a megfelelő stratégia osztályba helyezzük el.

Szerkezet:



14. Sablon művelet

Név, osztály: sablon művelet (template method), viselkedési (behavioral)

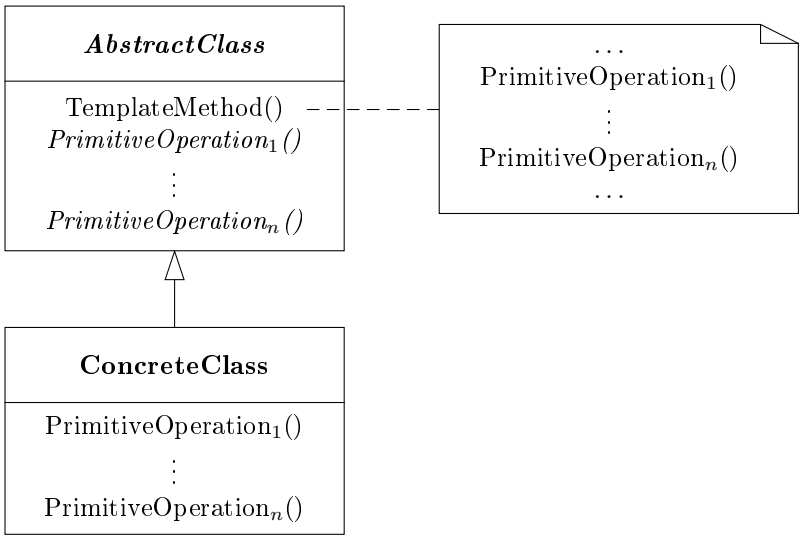
Cél: Egy műveletben szereplő algoritmus vázának definiálása úgy, hogy néhány lépést alosztályokra hagyunk. A sablon művelet segítségével az alosztályok a szerkezet megtartásával újra definiálhatják az algoritmus bizonyos lépéseit.

Más nevek: –

Felhasználhatóság: A sablon művelet minta használható az alábbi esetekben:

- Egy algoritmus állandó részének egyszeri implementálására, a változó részeket a származtatott osztályokban valósítjuk meg.
- Amikor osztályok közös viselkedését kell kiemelnünk és lokalizálnunk egy közös osztályban, hogy elkerüljük a kód ismétlődését.
- Alosztályok kiterjesztésének kézben tartására. A sablon művelet „hook” műveleteket hív a megfelelő helyeken, így csak ezeken a pontokon lehet kiegészíteni.

Szerkezet:



15. Összetétel

Név, osztály: összetétel (composite), szerkezeti (structural)

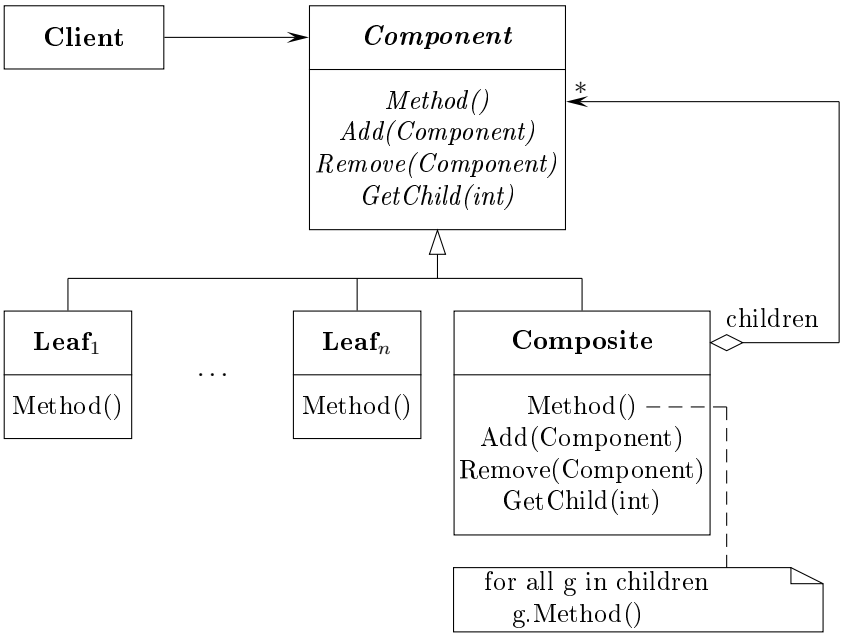
Cél: Objektumok összefogása fa szerkezetbe, hogy rész-egész hierarchiákat ábrázoljunk. Az összetételek lehetővé teszik, hogy a kliensek az egyedi objektumokat és az összetett szerkezeteket is egységesen kezeljék.

Más nevek: –

Felhasználhatóság: Az összetétel minta használható az alábbi esetekben:

- Objektumok rész-egész hierarchiáját kell ábrázolni.
- A használat szempontjából (kliensek) nem akarunk különbséget tenni az egyszerű elemek és az összetett elemek között, hanem egységesen akarjuk kezelni őket.

Szerkezet:



16. Látogató

Név, osztály: látogató (visitor), viselkedési (behavioral)

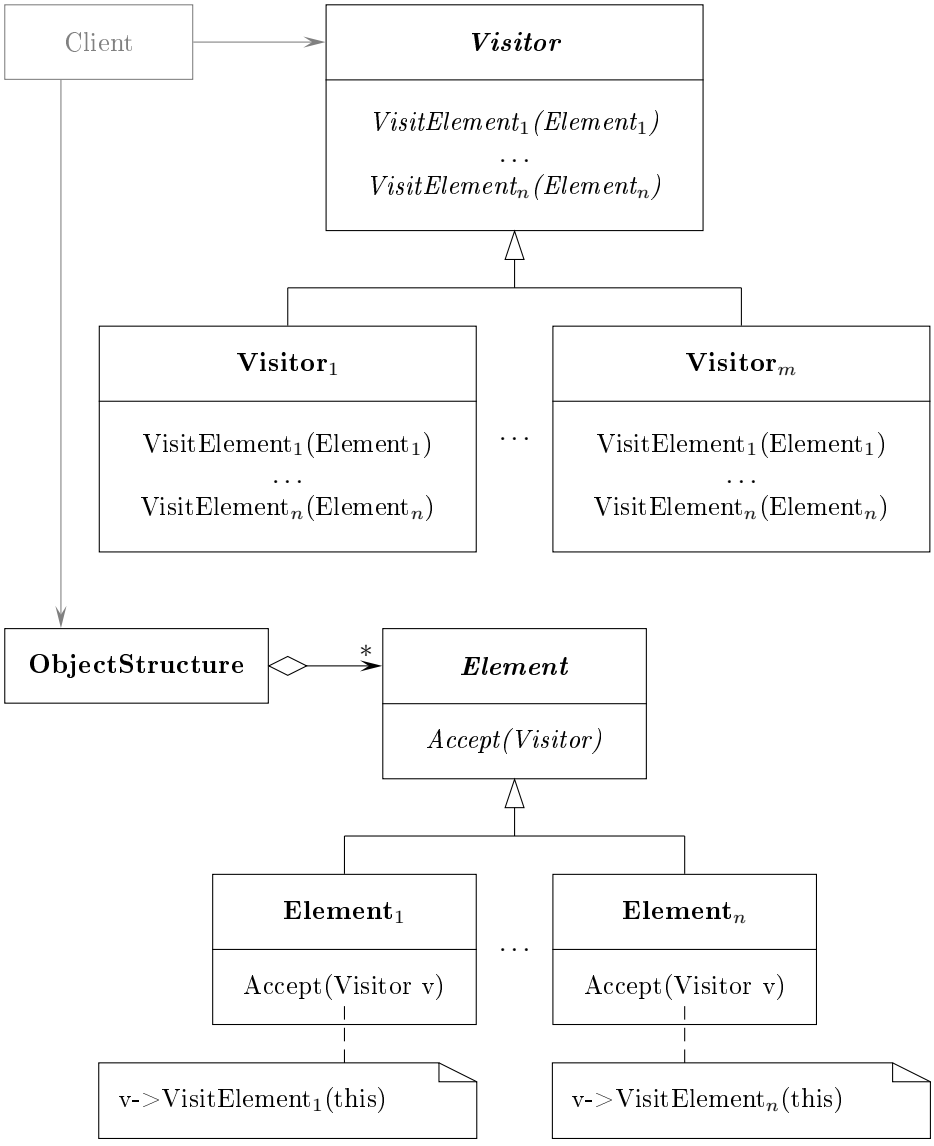
Cél: Egy szerkezet objektumain végrehajtandó művelet ábrázolása. Így lehetővé válik új művelet megadása anélkül, hogy megváltoztatnánk azon elemek osztályait, amelyeken a műveletet végrehajtjuk.

Más nevek: –

Felhasználhatóság: A látogató minta használható az alábbi esetekben:

- Ha egy szerkezet sok, eltérő felülettel rendelkező osztályból áll, és ennek objektumain kell olyan műveletet végrehajtanunk, amelyik függ a konkrét osztálytól.
- Sok és különböző műveletet kell végrehajtanunk egy szerkezet objektumain, és nem akarjuk az osztályokat túlzsúfolni ezekkel a műveletekkel.
- Az osztályszerkezet nemigen változik, de sokszor kell új műveletet bevezetnünk. (Ha az osztályszerkezet gyakran változik, akkor nem célszerű a minta alkalmazása!)

Szerkezet:


Client
Visitor
VisitElement1(Element1)
...
VisitElementn(Elementn)
Visitor1
VisitElement1(Element1)
...
VisitElementn(Elementn)
...
Visitorm
VisitElement1(Element1)
...
VisitElementn(Elementn)
ObjectStructure
*
Element
Accept(Visitor)
Element1
Accept(Visitor v)
...
Elementn
Accept(Visitor v)
v->VisitElement1(this)
v->VisitElementn(this)

17. Absztrakt gyártó

Név, osztály: absztrakt gyártó (abstract factory), létrehozási (object creational)

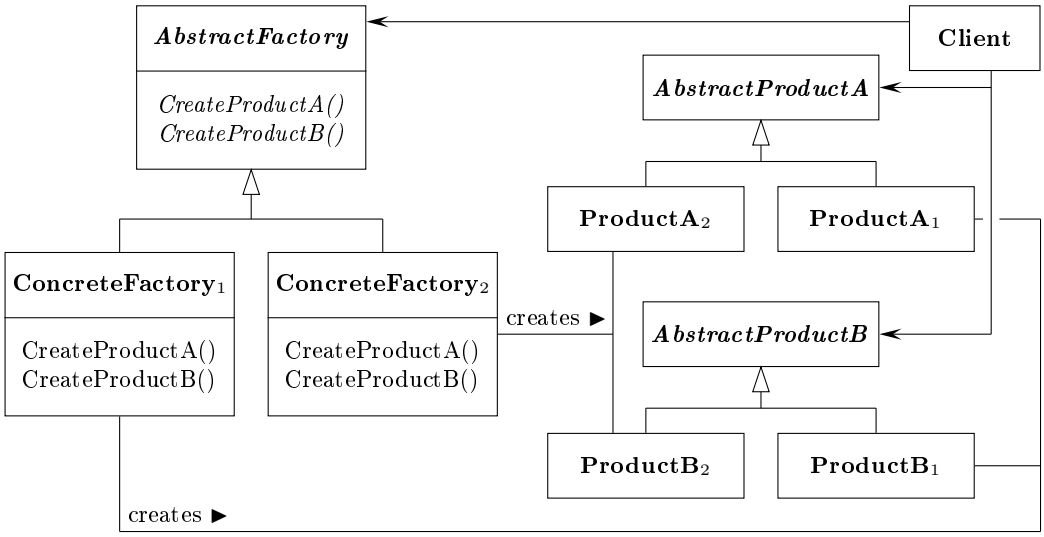
Cél: Felületet biztosítani arra, hogy kapcsolatban vagy függőségben álló objektumokat hozzunk létre a konkrét osztály meghatározása nélkül.

Más nevek: kit

Felhasználhatóság: Az absztrakt gyártó minta használható az alábbi esetekben:

- Egy rendszer nem függhet attól, hogy az egyes termékeit (objektumai) miként hozzuk létre, állítjuk össze és ábrázoljuk.
- Egy rendszert több családba tartozó termékek valamelyikével kell konfigurálnunk.
- Kapcsolatban álló termékek egy családját a terv szerint együtt kell használnunk, és ezt a megszorítást biztosítani akarjuk.
- Termékek osztálykönyvtárát szeretnénk kialakítani, és csak az interfészt akarjuk felfedni, az implementációt nem.

Szerkezet:



18. Híd

Név, osztály: híd (bridge), szerkezeti (structural)

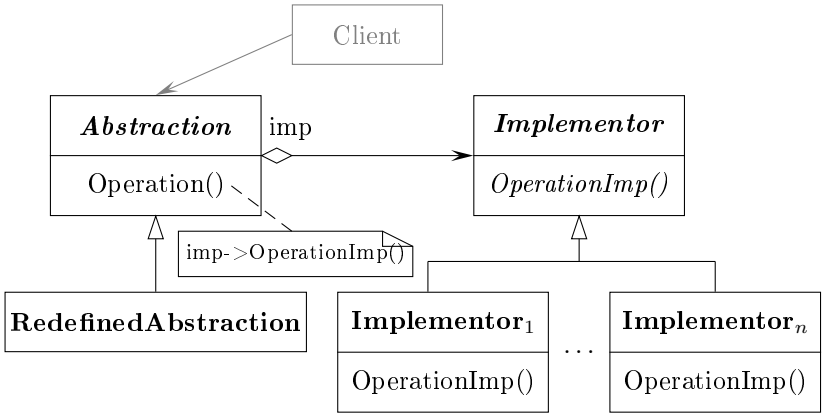
Cél: Szétválasztani egy absztrakciót az implementációjától, így a kettő egymástól függetlenül változhat.

Más nevek: handle/body

Felhasználhatóság: A híd minta használható az alábbi esetekben:

- El akarjuk kerülni egy absztrakciónak és megvalósításának permanens összekötését (a megvalósítást futási időben kell kiválasztani, vagy változtatni).
- Az absztrakciót és a megvalósítást is ki akarjuk terjeszteni származtatással, különbözőképpen kombinálva ezeket.
- Az absztrakció megvalósításának módosítása ne legyen hatással a kliensekre (ne kelljen újra fordítani a kódot).
- Az absztrakció megvalósítását teljes egészében el akarjuk rejteni a kliensek elől (C++: az osztály reprezentációja látható az osztály deklarációjában, ezt lehet elkerülni).
- Egy magvalósítást több objektum között szeretnénk szétosztani, és ezt el akarjuk rejteni a kliens elől.

Szerkezet:



19. Építő

Név, osztály: építő (builder), létrehozási (object creational)

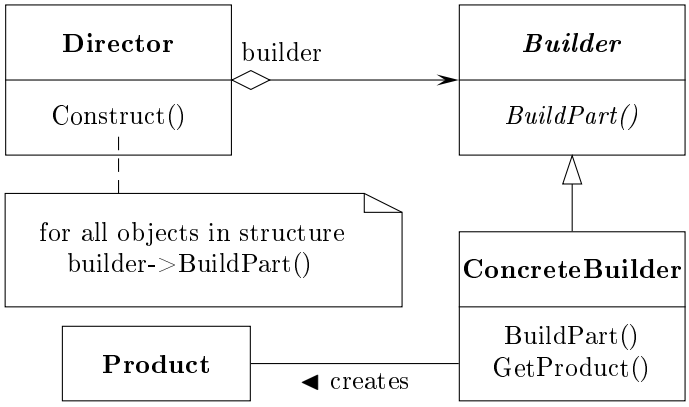
Cél: Egy összetett objektum konstrukciójának és reprezentációjának szétválasztása, így ugyanaz a konstrukciós folyamat eltérő reprezentációkat hozhat létre.

Más nevek: –

Felhasználhatóság: Az építő minta használható az alábbi esetekben:

- Egy összetett objektum létrehozásának algoritmusát függetleníteni kell a részektől és azok összeállításától.
- A konstrukciós eljárásnak meg kell engednie a létrehozandó objektum eltérő reprezentációit.

Szerkezet:



20. Gyártó művelet

Név, osztály: gyártó művelet (factory method), létrehozási (class creational)

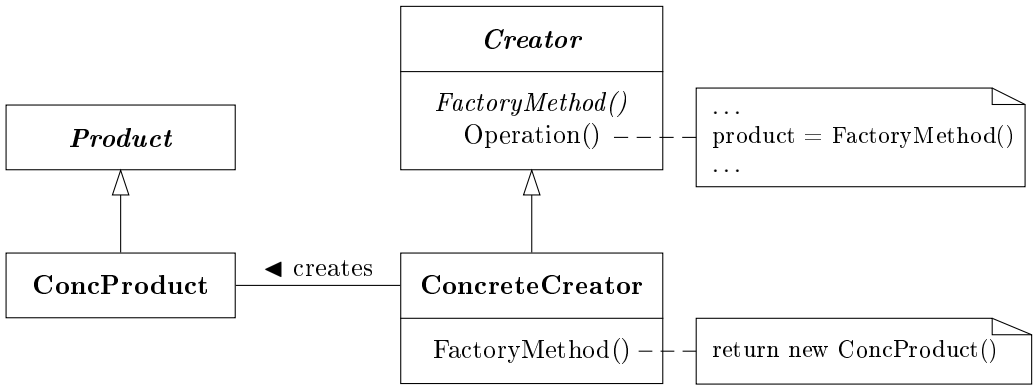
Cél: Egy objektum létrehozási felületének meghatározása úgy, hogy az alosztályok döntsék el, hogy az objektum melyik osztály példánya legyen. A gyártó művelet lehetővé teszi, hogy egy osztály a példányosítást elhalassza az alosztályhoz.

Más nevek: virtual constructor

Felhasználhatóság: A gyártó művelet minta használható az alábbi esetekben:

- Egy osztályban nem tudjuk előre megadni, hogy milyen osztályok példányait kell létrehoznunk.
- Egy osztály rá akarja bízni a leszármazottaira, hogy maghatározzák a létrehozandó objektumokat.
- Osztályok tovább adják a felelősséget számos segéd alosztály egyikének, és lokalizálni akarjuk annak ismeretét, hogy melyik segéd alosztály a kiválasztott.

Szerkezet:



21. Prototípus

Név, osztály: prototípus (prototype), létrehozási (object creational)

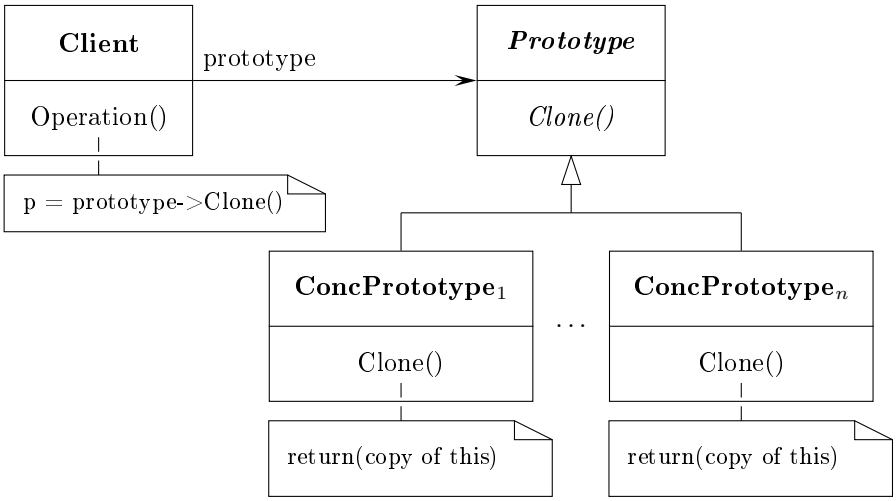
Cél: Prototípus alapján példányosítható objektumok jellegének meghatározása, és új objektumok létrehozása a prototípus másolásával.

Más nevek: –

Felhasználhatóság: A prototípus mintát használjuk ha a rendszernek függetlennek kell lennie attól, hogy a termékeket miként hozzuk létre, állítjuk össze, reprezentáljuk és

- amikor a példányosítandó osztályokat futási időben határozzuk meg, vagy
- nem akarjuk a gyártó osztályok olyan hierarchiáját létrehozni, amely a termékekével egyezik meg, vagy
- amikor egy osztály példányai csak néhány lehetséges állapotkombináció egyikét vehetik fel.

Szerkezet:



22. Átalakító

Név, osztály: átalakító (adapter), szerkezeti (structural)

Cél: Egy osztály felületének konvertálása a kliens által elvártra. Az átalakító lehetővé teszi, hogy olyan osztályok is együttműködjenek, amelyek az eltérő felület miatt nem lennének erre képesek.

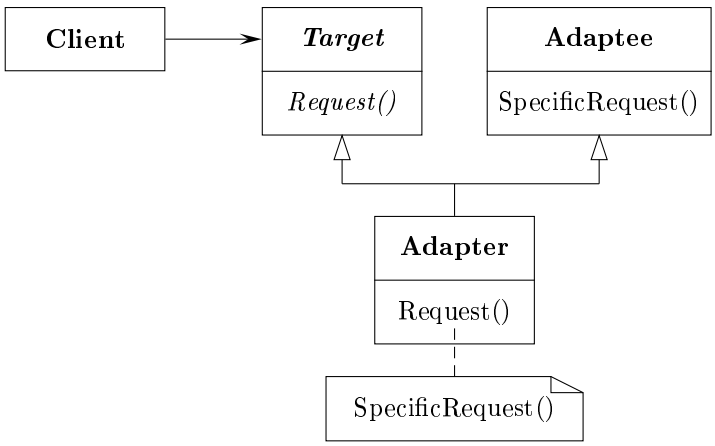
Más nevek: wrapper

Felhasználhatóság: Az átalakító minta használható az alábbi esetekben:

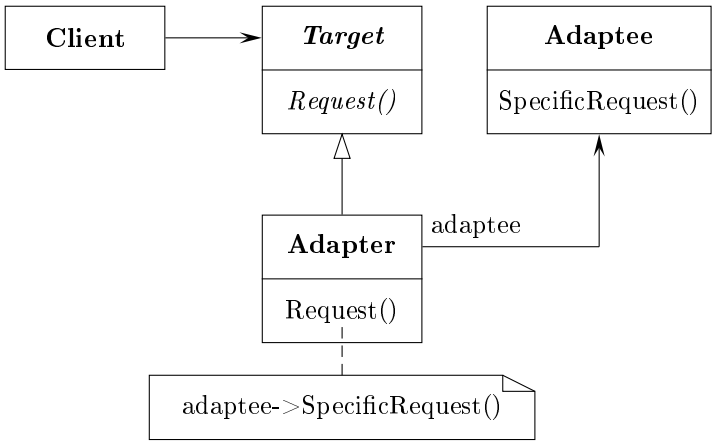
- Egy meglévő osztályt szeretnénk felhasználni, de annak a felülete nem illeszkedik az igényekhez.
- Egy újrafelhasználható osztályt szeretnénk létrehozni, amelyik nem kapcsolódó vagy előre nem látott osztályokkal működik együtt, azaz olyanokkal, amelyek felülete nem feltétlen kompatibilis.
- (*csak objektum átalakítás esetén*) Számos osztályt kell használnunk, de nem célszerű ezek felületét származtatással átalakítanunk. Ekkor egy átalakító objektum megváltoztatja a szülő osztály felületét.

Szerkezet:

Osztály átalakító:



Objektum átalakító:



23. Díszítő

Név, osztály: díszítő (decorator), szerkezeti (structural)

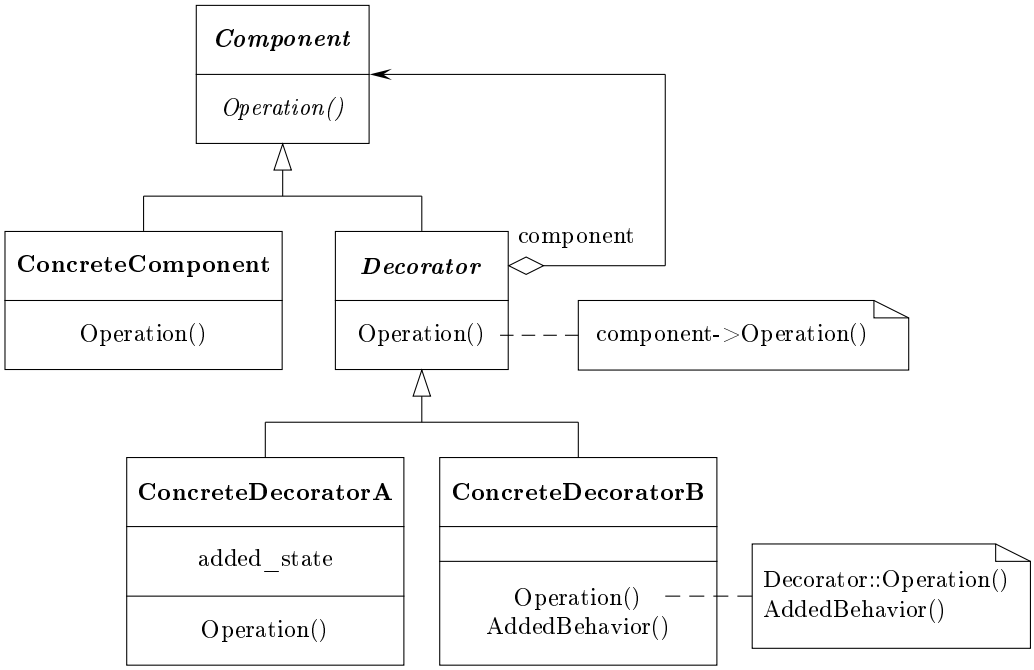
Cél: Egy objektum funkcionalitásának dinamikus kiegészítése. A díszítők a származtatás helyett használhatók erre a célra.

Más nevek: wrapper

Felhasználhatóság: A díszítő minta használható az alábbi esetekben:

- Egyedi objektumok funkcionalitásának dinamikus és átlátszó bővítésére. (Átlátszó: nincs hatással az objektumra.)
- Visszavonható funkcionalitás esetén.
- Ha az öröklődéssel történő kiterjesztés nem praktikus. (Például túl sok osztály jönne létre.)

Szerkezet:



24. Arculat

Név, osztály: arculat (facade), szerkezeti (structural)

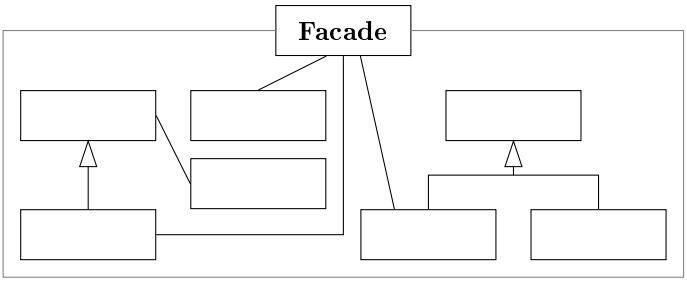
Cél: Egységes felület biztosítása egy alrendszer felületeihez. Az arculat egy magasabb szintű felületet definiál, amelynek segítségével az alrendszer használata egyszerűbbé válik.

Más nevek: –

Felhasználhatóság: Az arculat minta használható az alábbi esetekben:

- Egyszerű felületet szeretnénk biztosítani egy összetett alrendszerhez. Az arculat a legtöbb kliens számára megfelelő egyszerű, alapértelmezett nézetét biztosítja az alrendszernek. Csak speciális igényekkel rendelkező kliensek használnak mélyebb felületet.
- Túl sok a függőség (kapcsolat) egy absztrakció osztályainak megvalósítása (alrendszer) és a kliensek között. Vezessünk be egy arculatot, amellyel elválasztjuk az alrendszert a kliensektől, így biztosítva az alrendszer függetlenségét és hordozhatóságát.
- Az alrendszerek rétegezésének kialakításához. Arculat használható minden szint belépési pontjának definiálására. Az alrendszerek közötti függőségek egyszerűsíthetőek, ha csak az arculatokon keresztül kommunikálnak.

Szerkezet:



25. Könnyűsúlyú

Név, osztály: könnyűsúlyú (flyweight), szerkezeti (structural)

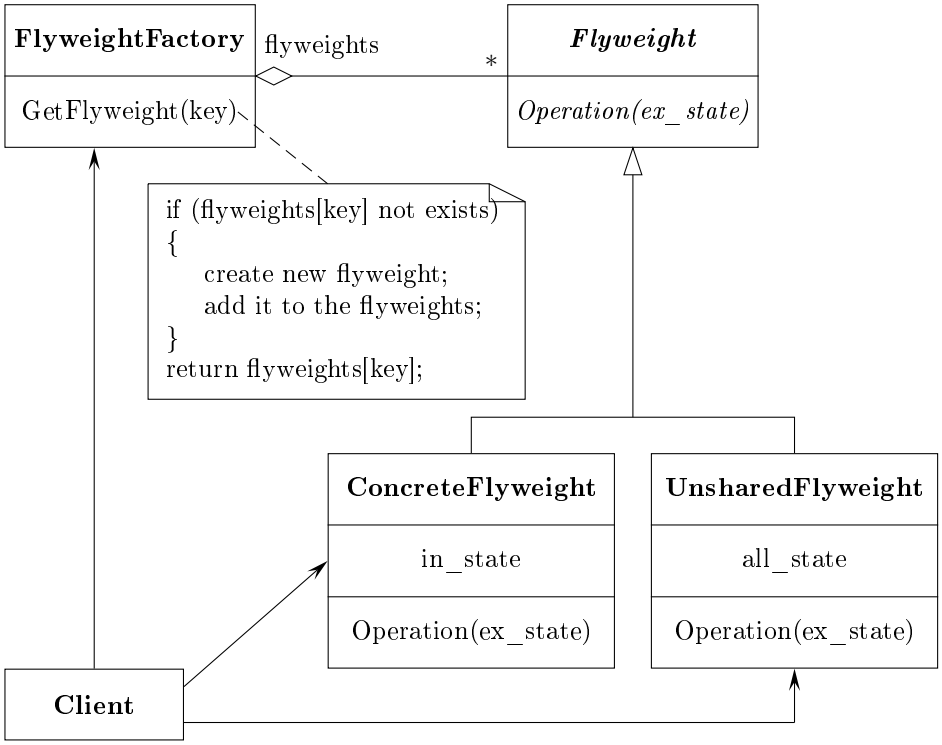
Cél: Megosztás használata, nagy számú „finomszemcsés” objektum hatékony kezeléséhez. (Finomszemcsés: sok információt kell nyilvántartani. Megosztás: nem objektumonként tartjuk nyilván az információt, hanem azt kiemeljük.)

Más nevek: –

Felhasználhatóság: A könnyűsúlyú minta használható, ha a következők együttesen fennállnak:

- Az alkalmazásban használt objektumok száma nagy.
- A memóriaigény nagy a használt objektumok száma miatt.
- A legtöbb objektum állapota külsővé tehető.
- Objektumok számos csoportját helyettesíthetjük viszonylag kevés osztott objektummal, ha a külső állapotot kiemeljük.
- Az alkalmazás nem függ az objektumok identitásától. (Miután könnyűsúlyú objektumok osztottak, az identitás teszt azonos eredményt adhat két különböző objektum esetén is.)

Szerkezet:


FlyweightFactory
GetFlyweight(key)
flyweights
*
Flyweight
Operation(ex_state)
if (flyweights[key] not exists)
{
create new flyweight;
add it to the flyweights;
}
return flyweights[key];
ConcreteFlyweight
in_state
Operation(ex_state)
UnsharedFlyweight
all_state
Operation(ex_state)
Client

26. Helyettes

Név, osztály: helyettes (proxy), szerkezeti (structural)

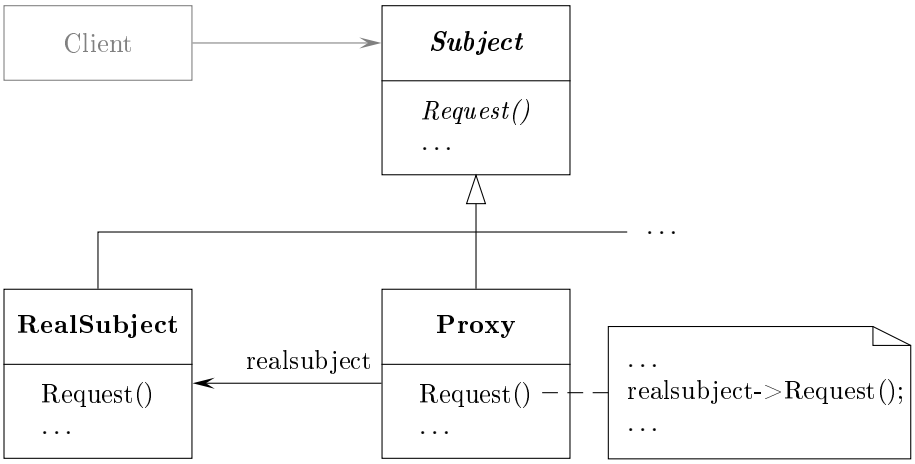
Cél: Egy helyettes létrehozása egy objektumhoz, annak érdekében, hogy szabályozzuk a hozzáférést.

Más nevek: surrogate

Felhasználhatóság: A helyettes minta használható, ha egy objektumra egy egyszerű mutatónál rogalmasabban vagy finomabban szeretnénk hivatkozni. Néhány lehetséges példa:

- A *távoli helyettes* egy lokális reprezentánsa egy eltérő hozzáférési területen található objektumnak. (Lehetséges elnevezés: követ, ambassador).
- A *virtuális helyettes* költséges objektumokat hoz létre igény esetén.
- A *védelmi helyettes* szabályozza a hozzáférést az eredeti objektumhoz. Ez hasznos, ha objektumokhoz eltérő hozzáférési jogok tartoznak.
- Az *intelligens hivatkozás* egy sima mutatót helyettesít, és kiegészítő műveleteket hajt végre, amikor az eredeti objektumhoz fordulunk.
  - Hivatkozás számlálás, és automatikus felszabadítás.
  - Állandó (perzisztens) objektumok betöltése az első hivatkozásnál.
  - Az eredeti objektum zárolása hivatkozás esetén kizárólagos használat céljára, és ennek ellenőrzése.

Szerkezet:



27. Értelmező

Név, osztály: értelmező (interpreter), viselkedési (behavioral)

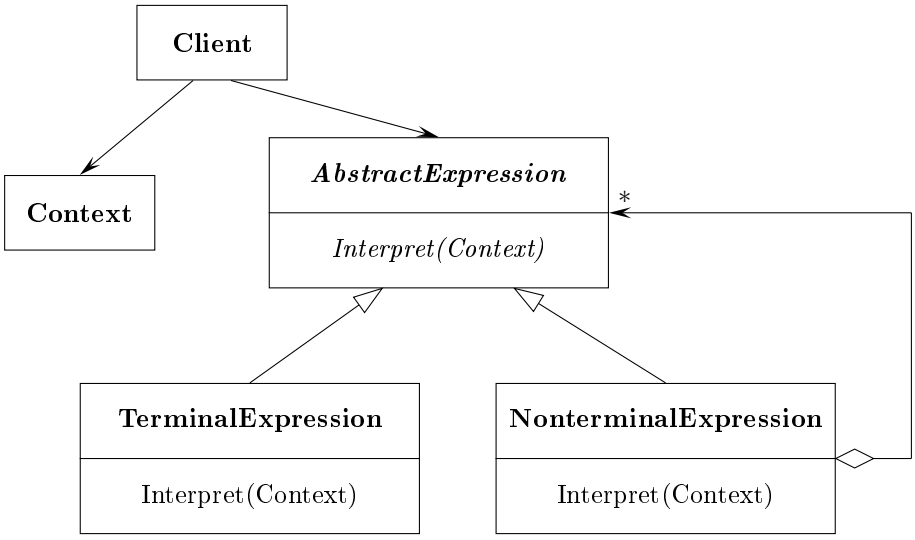
Cél: Megadni egy adott nyelvtannak megfelelő reprezentációt és értelmezőt, amelyben az utóbbi felhasználja a reprezentációt a nyelv mondatainak meghatározásához.

Más nevek: –

Felhasználhatóság: Az értelmező minta használható, amikor egy nyelvet kell értelmezni, és a nyelv mondatai absztrakt szintaxisfában ábrázolhatóak. Az értelemező minta a leginkább alkalmazható, ha:

- a nyelvtan egyszerű (bonyolult esetben túl nagy és kezelhetetlen osztály szerkezethez jutunk);
- a hatékonyság nem kritikus tényező (különben célszerűbb automatákat implementálni).

Szerkezet:



28. További tervminták

Kétféle módon vezethetünk be új tervmintákat:

1. a meglevő osztályokat bővítjük újabb mintákkal, vagy
2. új osztályokat vezetünk be.

Ebben a fejezetben az első lehetőséget vizsgáljuk meg, azaz bemutatunk néhány olyan mintát, amelyek a GoF osztályozásba illeszthetők, de nem GoF minták.

28.1. Lusta példányosítás

Név, osztály: lusta példányosítás (lazy initialization), létrehozási (object creational)

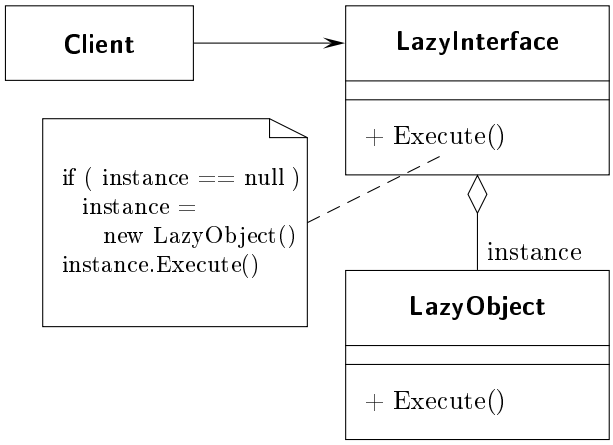
Cél: Egy (összetett, bonyolult) objektum létrehozásának elhalasztása akkorra, amikor ténylegesen használni fogjuk az objektumot.

Más nevek: –

Felhasználhatóság: A lusta példányosítás mintát használjuk, ha egy objektum létrehozását az első használat idejéig késleltetni szeretnénk, ami azért lehet célszerű, mert

- nem garantált, hogy az összetett objektumot egyáltalán igénybe vesszük, és ezért felesleges lenne előre létrehozni, illetve
- az objektum létrehozását elhalasztjuk arra az időpontra, amikor a rendszer kevésbé terhelt.

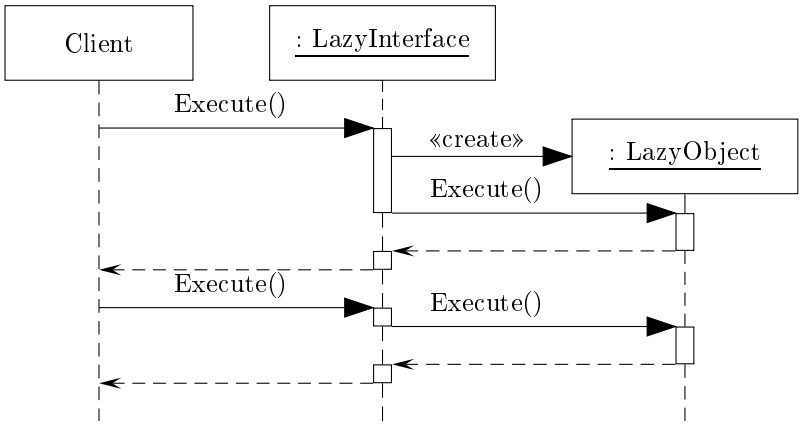
Szerkezet:



Elemek:

- LazyInterface: A lusta példányosítású objektum felülete, a kliens csak ezt éri el. Hivatkozik az objektumra, műveleteinek megvalósításában ellenőrzi, hogy létezik-e már az objektum. Ha szükséges létrehozza, és az objektum megfelelő műveletét hajtja végre.
- LazyObject: Az objektum, aminek példányosítását el szeretnénk halasztani.

Együttműködés:



Ez a minta tulajdonképpen a helyettes (proxy) minta egy egyszerűsített változata, hiszen amikor az objektumra szükség van, akkor azt létrehozzuk és továbbítjuk felé az igényt, szemben a helyettessel, ahol bizonyos műveleteket a helyettesben valósítunk meg.

28.2. Lusta gyártó

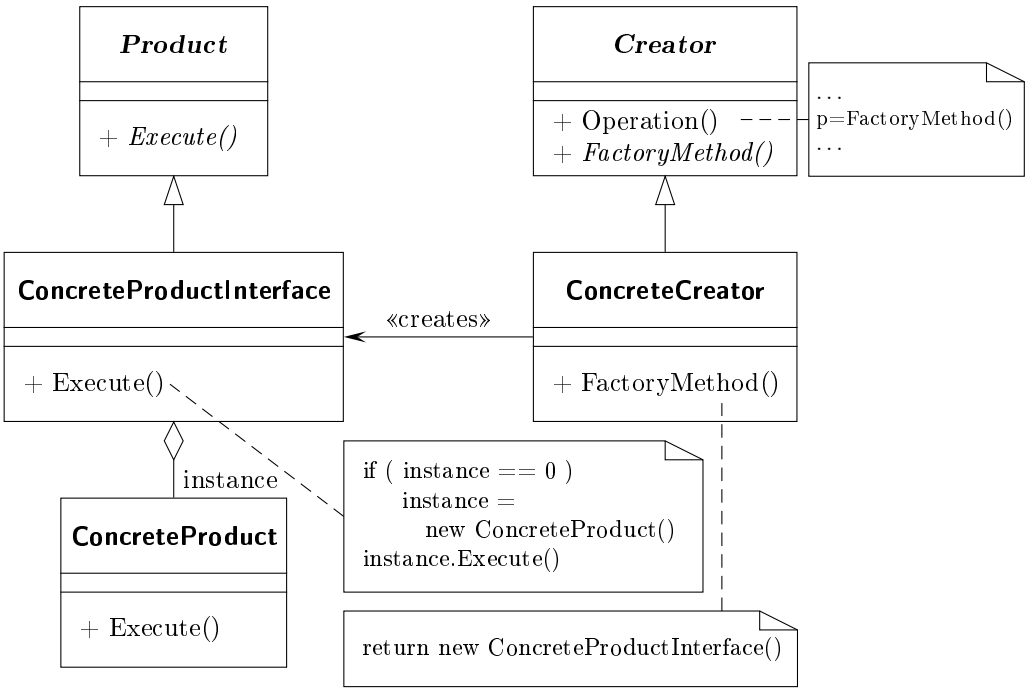
Név, osztály: lusta gyártó (lazy factory), létrehozási (object creational)

Cél: A létrehozandó objektum típusát egy származtatott osztályra delegáljuk, és biztosítani szeretnénk, hogy csak akkor jöjjön létre az objektum, amikor arra ténylegesen szükség van.

Más nevek: –

Felhasználhatóság: A létrehozandó objektumok osztályát nem akarjuk, vagy nem tudjuk előre megadni, és a létrehozást a lehető legjobban el akarjuk halasztani.

Szerkezet:



Elemek:

- Product: a termékek közös felületét megadó absztrakt osztály.
- Creator: a termékeket használó osztály, ami a konkrét példány létrehozását a származtatott osztályra hagyja. A terméket az Operation műveletben hozza létre.
- ConcreteCreator: a terméket (esetünkben egy lusta példányt) létrehozó osztály.
- ConcreteProductInterface: a konkrét termék felületével megegyező osztály, ami felel a termék létrehozásáért, és a továbbiakban az igényeket annak továbbítja.
- ConcreteProduct: a konkrét termékek osztálya.

Együttműködés: A gyártó művelet mintához hasonlóan járunk el, azonban a műveletben előállított termék csupán a tényleges termék felületét tartalmazza (ConcreteProductInterface), ami felel a tényleges tevékenységet végző objektum (ConcreteProduct) létrehozásáért. A Creator osztály az Operation műveletben csak létrehoz egy példányt, de azt nem feltétlenül használja még ekkor. (Ha ebben a műveletben feltétel nélkül használná a példány egy műveletét, akkor nincs értelme eltérni a gyártó művelet mintától, hiszen azonnal létrejönne a konkrét termék is.)

Ez a minta tulajdonképpen a gyártó művelet és a lusta példányosítás kombinációja.

28.3. Objektumkészlet

Név, osztály: objektumkészlet (object pool), létrehozási (object creational)

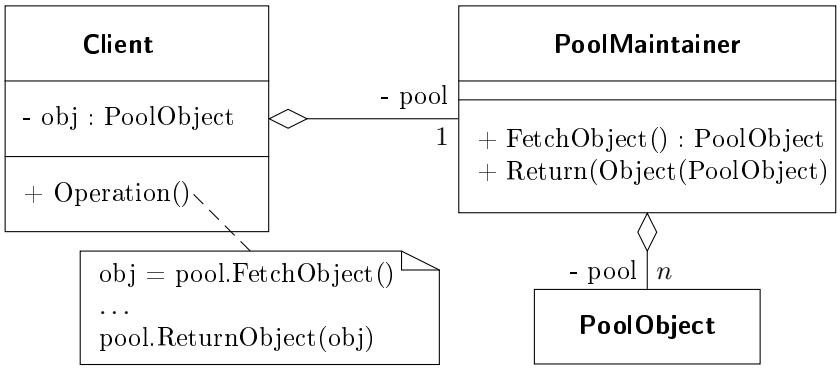
Cél: Objektumok gyakori létrehozásának és törlésének elkerülése úgy, hogy objektumok egy halmazát mindig készleten tartjuk, és igénybe vesszük.

Más nevek: –

Felhasználhatóság: Objektumok létrehozása, illetve törlése költséges művelet lenne, ugyanakkor

- sokszor hozunk létre és szüntetünk meg objektumokat, és
- egyidejűleg csak korlátozott számú példányt használunk.

Szerkezet:



Elemek:

- PoolObject: a gyakran létrehozandó és törlendő objektumok osztálya.
- PoolMaintainer: az objektumok halmazát kezelő osztály, *készletkezelő*. Nyilvántart adott számú PoolObject elemet, amelyeket szükség esetén az igénylők rendelkezésére bocsájt.
- Client: tetszőleges objektum; egyike azoknak, amelyek gyakran hoznának létre és szüntetnének meg PoolObject típusú elemeket. Hivatkozik egy készletkezelő példányra, amelyen keresztül objektumokhoz férhet hozzá.

Együttműködés:

Az igénylő (Client) szükség esetén az objektumok létrehozása helyett egy elemet igényel a készletkezelőtől a FetchObject művelet segítségével, és amikor már nincs szüksége az objektumra – a megszüntetés helyett – visszahelyezik azt a ReturnObject művelettel. A megkapott objektumhoz kizárólagos hozzáféréssel rendelkezik, azaz azt másnak egyidejűleg nem adhatja ki a készletkezelő.

29. Rossz minták (antiminták)

Antimintának tekintünk minden olyan gyakorlatban gyakran előforduló elvet, vagy eljárást, amely szakszerűtlen, használhatatlan, vagy káros, és ezért elkerülendő.

Nemcsak az objektumelvű tervezésben, hanem a szoftvertechnológia minden területén találunk ilyen rossz mintákat, beleértve a projektmenedzsmentet, az architektúrát, az implementációt és a karbantartást.

Az objektumelvű tervezés esetén 13 rossz mintát szokás azonosítani.

Mindenható objektum (God Object)

Olyan osztályt definiálunk, vagy alkalmazunk, amely túl nagy felelősséggel bír, azaz a tevékenységek, vagy az adatok jelentős részét kezeli. Az objektum így hasonlítani fog a procedurális szemlélet főprogramjához. Ezzel teljesen ellentmondunk az objektumelvű megközelítésnek, amely kimondottan a felelősség továbbadására épít. (Lásd 4.1. részt.)

Megoldás: Osszuk meg a felelősséget új osztályok bevezetésével, amelyek példányait az eredeti osztály kezeli.

Kör – ellipszis probléma (Circle – ellipse problem)

Származtatás esetén előfordulhat, hogy olyan speciális osztályt hozunk létre, amely az őshöz képest erőteljes invariánsokat tartalmaz, és az ősből örökölt metódusok ezeket az invariánsokat megszeghetik, mert ott még nem kellett figyelni rájuk. Ezáltal a származtatott objektum inkonzisztens állapotba kerülhet. Például, amikor egy ellipszisből származtatunk egy kört, és az ellipszisnek van olyan utasítása, amely csak az egyik irányba növeli a sugarat, a másikba nem.

Megoldás: Sok megoldás kínálkozik a problémára, például a származtatott osztályban felüldefiniáljuk az összes, az invariánst megsérthető műveletet, vagy lefutás előtt ellenőrizzük, hogy a művelet végrehajtható-e, illetve változtathatunk a származtatás sorrendjén.

Vérszegény tárgyköri modell (Anemic Domain Model)

Az osztályainkat úgy tervezzük, hogy a viselkedésért felelős osztályokat teljesen elkülönítjük az adatoktól felelős osztályoktól. Ez hasonló, mintha szeparálnánk egy osztály műveleteit és attribútumait két külön osztályba. Ezáltal nemcsak bonyolultabb, nehezebben megérthető szerkezetet kapunk, de a működése is kevésbé felügyelhető.

Megoldás: Egy osztályhoz tartozó viselkedést, illetve adatokat tartsuk egy helyen, a csak viselkedéssel, illetve csak adattal rendelkezőeket lehetőség szerint vonjuk össze.

Felesleges rétegződés (Yet Another Useless Layer)

A programot annyi rétegbe (komponensbe, alrendszerbe) tagoljuk, hogy az már felesleges a programszerkezet számára, így az összetartozó szerkezetek is szétbomlanak, ezért nehézkessé válik a rendszer áttekintése és működése.

Megoldás: A rétegeket a programszerkezetnek megfelelően alakítsuk ki, és utólag ha szükséges, vonjunk össze azokat a rétegeket, amelyek hasonló funkciókért felelnek.

Jojó probléma (Yo–yo problem)

Egy olyan osztályszerkezetet kell áttekintenünk, amely rendkívül bonyolult, és hosszú öröklődési láncot fut be, ezért a programozónak oda-vissza kell ugrálnia az osztályok definíciói között, hogy megértse a műveletek működését és a vezérlési folyamatot.

Megoldás: Igyekezzünk az öröklődési láncot olyan alacsonyan tartani, hogy az ne haladjon meg egy olyan korlátot, ami áttekinthetetlenné teszi a szerkezetet. Származtatás helyett alkalmazhatunk aggregációt is, ezért vizsgáljuk felül, hol válthatjuk ki vele az öröklődési technikát.

Objektumtobzódás (Object Orgy)

Nem megfelelő a láthatóság szabályozása, ezzel túlzott hozzáférést biztosítunk az osztály implementációjához külső osztályok objektumainak. Ezért nehéz az objektum viselkedésének követése, illetve nem várt viselkedést tapasztalhatunk, amennyiben a külső objektum olyan műveletet futtat, amelyiket csak belső használatra szántunk.

Megoldás: Az osztályon belüli láthatóságok újraosztása, vagy minden tag elrejtése, és egy új felület kialakítása az osztályhoz.

Kopogó szellem (Poltergeist)

Olyan osztály definiálása, amelyből rövid élettartamú, állapotoktól mentes objektumokat példányosítunk, és feladata pusztán az, hogy üzenetet továbbítson egy permanensebb osztálynak. Ezáltal egy felesleges osztály keletkezik a szerkezetben.

Megoldás: Töröljük az osztályt, és a funkcióját delegáljuk a permanensebb osztályra.

Szekvenciális csatolás (Sequencial Coupling)

Egy osztály műveleteit csak adott sorrendben lehet végrehajtani, és a rossz sorrendű hívás hibás működéshez vezethet.

Megoldás: Adjunk olyan felületet (műveletet) az osztályhoz, amely a hívási szekvenciát magában foglalja, így a külső objektum nem érzékeli a sorrendiséget.

Segéd származtatásának elkerülése (BaseBean)

Származtatás egy kisegítő osztályból. A kisegítő osztályok rendszerint stabilak, és hasonlóak a különböző implementációkban, azonban, ha bármilyen változtatást végzünk az osztályban, az hatással van a leszármazottakra, továbbá a leszármazott olyan funkciókat is örökölhet, amelyek nem kívánatosak. Például egy vektorból származtatunk egy sort.

Megoldás: Származtatás helyett használjunk aggregációt, azaz csatoljuk bele az eredeti ős osztályt az új osztályba, és alakítsunk ki egy olyan felületet, amely pontosan megfelel az elvárásainknak. (Újrafelhasználás.)

Függvény visszacsatolás (Call Super)

A származtatás során felül kell definiálnunk az ős egy műveletét, majd a felüldefiniált műveletben meghívjuk az ős osztály műveletét. Ezáltal az általános viselkedést kapjuk vissza a leszármazottban, így ez a funkció nem specializálható.

Megoldás: Amennyiben tényleg szükségünk van az ős meghívására, akkor vezessünk be az ős osztályban egy absztrakt műveletet, amelyet az ős művelete meghív, és ezt definiáljuk felül a leszármazottban, ezáltal a származtatott osztály kiegészítéseit is belehelyezhetjük.

Üres részosztály hiba (Empty Subclass Failure)

Úgy származtatunk egy osztályból, hogy a leszármazottat semmilyen új tulajdonsággal nem bővítjük, ennek ellenére példányosítva másként viselkedik, mint az ős egy példánya.

Megoldás: Valószínűleg a konstrukciós folyamatban van a hiba, és a példány rossz kezdeti értékekkel jön létre, ezért vizsgáljuk felül a konstruktort.

Kifogyó objektumkészlet (Object Cesspool)

Az objektumkészlet tervminta használata során felmerülő probléma, amikor objektumokat egy közös tárból veszünk elő, azonban a visszahelyezésük nem történik megfelelően – például nem kerülnek vissza az objektumok, vagy nem megfelelő állapotban kerülnek vissza –, ezért előbb utóbb elfogy az objektumkészlet, vagy hibás objektumok kerülnek újra használatba.

Megoldás: Az objektumkészlet kezeléséért felelős objektum (készletkezelő) viselkedése hibás, ezért azt kell javítanunk.

Egykeség (Singletonitis)

Az Egyke tervminta túlzott használata, amikor sok objektumból csak egy példány hozható létre. Ezáltal felmerül annak a veszélye, hogy több példányra szánt adatmennyiséget egy példány fog kezelni, így nehezebb lesz az adatok nyomon követése és kezelése.

Megoldás: Csak abban az esetben alkalmazzuk az Egyke megszorításait, amennyiben az valóban szükséges, és legfeljebb egy példány megengedett az adott objektumból. Ha az objektum valamely része elfogadható több példányban is, akkor ezeket a tulajdonságokat elemeljük ki egy új osztályba.

30. Konkurens programok előállítása

Egy konkurens program elemei a *processzek* (*folyamatok*, vagy egyszerűbb esetben *thread*-ek, azaz *szálak*[1]) és az *osztott objektumok*. A folyamatok párhuzamosan végrehajtott szekvenciális programok, amelyek az osztott objektumok segítségével kommunikálnak, illetve tartanak kapcsolatot egymással. Így egy párhuzamos rendszer elkészítésekor létre kell hoznunk a folyamatokat alkotó szekvenciális programokat, valamint vezérelnünk kell a folyamatok közötti kommunikációt, kapcsolatokat. Ez utóbbit nevezzük *szinkronizációnak*.

A szinkronizáció során két eszközt használhatunk.

- Biztosítanunk kell, hogy utasítások sorozata ne legyen megszakítható. Ez egy folyamat konzisztenciájának fenntartását szolgálja. Egy ilyen utasítássorozatot *atomi utasításnak* nevezünk.
- Lehetőséget kell teremtenünk arra, hogy egy folyamatot késleltessünk addig, amíg a rendszer egy adott tulajdonságot ki nem elégít. Ezt nevezzük *feltétel szinkronizációnak*.

Az absztrakt programokban *őrfeltételes utasítások* segítségével valósítjuk meg a szinkronizációt. Az őrfeltételes utasításokat eltérő módon valósíthatjuk meg különböző programozási nyelvekben. (Lényegében azonos típusú nyelvi elemek, vagy ha más nincs szemaforok segítségével.)

Egy őrfeltételes utasítás két részből áll: egy feltételből (*őr*), és egy utasítássorozatból (*törzs*). A törzs végrehajtása nem szakítható meg, így az egy atomi utasításnak felel meg. Az őr garantálja, hogy a végrehajtás csak adott feltétel teljesülése esetén kezdődhessen meg. Ha a feltétel azonosan `igaz`, akkor egy egyszerű atomi utasítást adunk meg. Az őrfeltételes utasítás formája:

await <*feltétel*> **then** <*utasítássorozat*> **ta**;

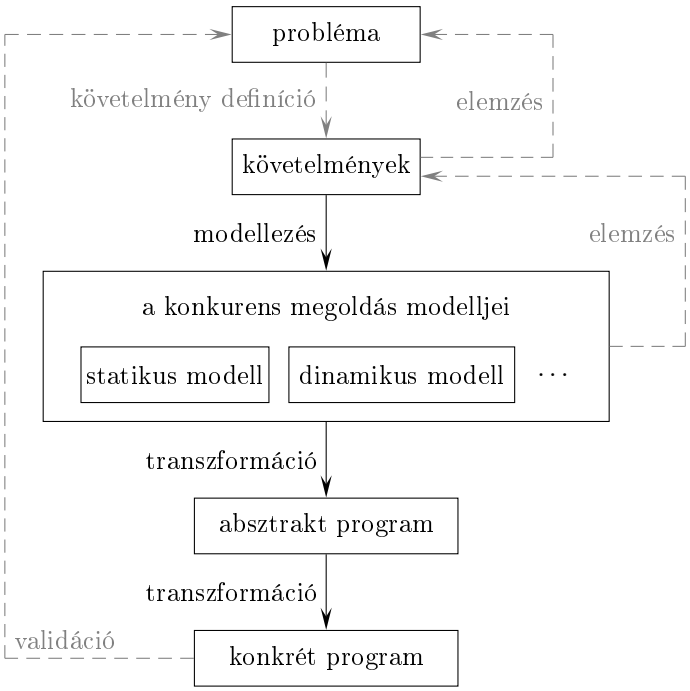
ahol <*feltétel*> egy logikai kifejezés, és <*utasítássorozat*> nem tartalmazhat iterációt vagy szinkronizációs (várakozó) utasításokat.

Kétféle módon hozhatunk létre konkurens rendszereket.

- A bemutatott eszközök közvetlen alkalmazásával valamilyen matematikai modell alapján, esetleg formális eszközök felhasználásával. Ekkor a rendszer tulajdonságait formálisan is elemezhetjük, helyességét bizonyíthatjuk.
- Egy objektumelvű modellt állítunk elő, amit az UML segítségével adunk meg. Ezt a modellt alakítjuk át absztrakt, illetve konkrét programmá. Ez kevésbé formális, könnyebben követhető, ugyanakkor a formális elemzést nem tartalmazza (30.1. ábra).

Mi a második módszert szemléltetjük a következőkben. Ebben először létre kell hoznunk a statikus modellt, majd az állapotdiagramot a dinamikus modellből. Az állapotdiagram

[1] Egy folyamat önálló memória területtel rendelkezik, a szálak közös memóriát használnak. Egy folyamat tartalmazhat több szálat. A szálak közötti kommunikáció a közös memória felhasználásával valósul meg, a folyamatok közötti kommunikációra csatornákat használhatunk.



30.1. ábra. Konkurens programok UML alapú előállítása

alapján határozhatjuk meg az őrfeltételes utasításokat, és állíthatjuk elő az absztrakt programot.

30.1. Konkurens programok előállításának lépései

Feltesszük, hogy a létrehozandó rendszer követelményleírása adott. Ekkor a következő eljárással állíthatjuk elő a megoldást.

1. Készítsük el a rendszer statikus modelljét (osztálydiagram)! Azonosítsuk a folyamatokat, és az általuk használt közös erőforrásokat! Határozzuk meg az osztályok attribútumait, az osztályok közötti kapcsolatokat!

2. Állítsuk elő a dinamikus modellből az állapotdiagramot! A rendszer állapota a folyamatok és az erőforrások állapotainak aggregációja lesz.

   - Határozzuk meg a folyamatok és az erőforrások állapotait! Egy folyamat állapotait a tevékenységei adják meg, az erőforrások állapotai szolgálnak a szinkronizációs feltételek definiálására. A továbbiakban azt mondjuk, hogy egy folyamat egy állapota *aktív*, ha abban erőforrást használ.
   - Ha a folyamatok eltérő prioritásokkal rendelkeznek, akkor vezessünk be speciális állapotokat – például igényel – a magasabb prioritású folyamatok esetén.
   - Határozzuk meg az erőforrás állapotok invariánsait! Szükség esetén vezessünk be új attribútumokat!

- Határozzuk meg az állapotátmenetek akcióit! Adjuk meg ezen akciók előfeltételeit! A folyamatok állapotátmenetei lesznek az absztrakt program atomi utasításai, az előfeltételek pedig a megfelelő őrfeltételes utasítások őrei. Az akciókat megadhatjuk, mint az entry és exit fázisai az aktív állapotoknak, illetve igénylésként az igénylési állapot(ok)ba történő belépéskor.
- Az állapotdiagramból elhagyhatjuk azokat az eseményeket, amelyek nem indukálnak állapotváltozást.

3. Készítsük el az absztrakt programot!
   - Határozzuk meg a dinamikus modellben bevezetett változók kezdeti értékeit, és írjuk fel a program vázát, mint egy kezdeti értékadás, és a folyamatok párhuzamos végrehajtása!
   - Készítsük el a folyamatok vázait az állapotdiagram felhasználásával!
   - Határozzuk meg az atomi utasításokat az állapotdiagram akciói alapján, és állítsuk elő a megfelelő őrfeltételes utasításokat! Helyezzük el ezeket az őrfeltételes utasításokat a folyamatok vázaiba az állapotátmeneteknek megfelelően!
4. Hozzuk létre a programot!
   - Valósítsuk meg az őrfeltételes utasításokat a választott nyelven!
   - Implementáljuk a szekvenciális részeket!
   - Szükség esetén transzformáljuk a programot, ha lehet, hogy egyszerűbb, illetve hatékonyabb legyen!

30.2. Első esettanulmány

A bemutatott eljárás menetét egy egyszerű feladaton keresztül mutatjuk be. Csak az absztrakt program előállításával foglalkozunk ebben az esetben.

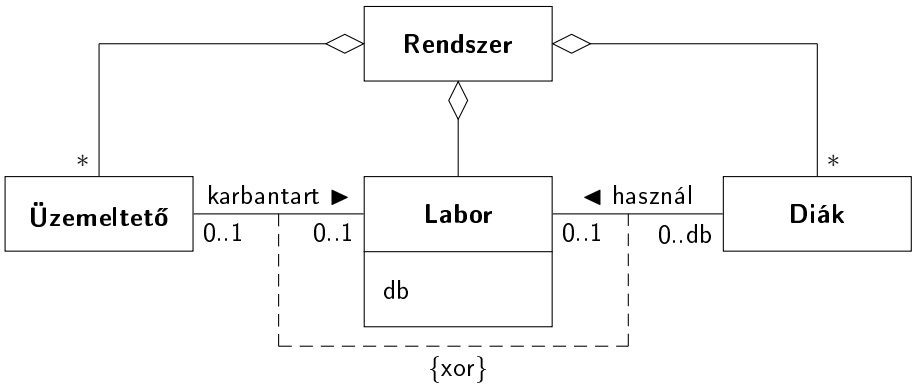
A feladatban egy számítógépes laboratórium használatát kell programmal szimulálnunk. Egyetlen laborral foglalkozunk, amelyben adott számú számítógép található. A labort hallgatók akarják használni. Egy hallgató végzi a tanulmányait, és amikor számítógépre van szüksége, akkor a laborhoz megy, és kint várakozik. Ha van szabad gép a laborban, akkor egy várakozó diák beléphet, és elkezdhet dolgozni a gépen. Miután befejezte a munkáját, a diák távozik, és folytatja tanulmányait, majd újra a laborhoz megy, és így tovább. A laboratórium számítógépeit az üzemeltetők tartják karban. Egyszerre csak egy üzemeltető végezhet karbantartást a labor összes gépen, de több üzemeltető jelenthet be karbantartási igényt. Karbantartás alatt diákok nem használhatják a labor gépeit. Ha egy üzemeltető karbantartási igényt jelent be, akkor diák nem léphet be a laborba, meg kell várnia a karbantartás végét. A karbantartás megkezdődhet, ha az összes laborban tartózkodó diák befejezi a munkáját, és elhagyja a labort.

30.2.1. Statikus modell

A leírás alapján a következő osztályokat azonosíthatjuk.

Rendszer: a modellezendő rendszernek megfelelő osztály.

Labor: a számítógépes laboratórium osztálya. Rendelkezik egy db attribútummal, ami megadja a laborban található gépek számát.



30.2. ábra. A rendszer osztálydiagramja

Üzemeltető: az üzemeltetők osztálya.

Diák: a diákok osztálya.

Az osztályok között a következő relációk állnak fenn.

- A **Rendszer** osztály a másik három osztály aggregációja. A multiplicitás értéke 1 a labor esetén, és tetszőleges a másik két osztály esetén.
- A **Diák** osztály asszociációs kapcsolatban áll a **Labor** osztállyal, az asszociáció neve használ. A relációban részt vevő diákok száma 0 és db között lehet.
- Az **Üzemeltető** osztály asszociációs kapcsolatban áll a **Labor** osztállyal, a reláció neve karbantart. Legfeljebb egy üzemeltető vehet részt ebben a kapcsolatban.

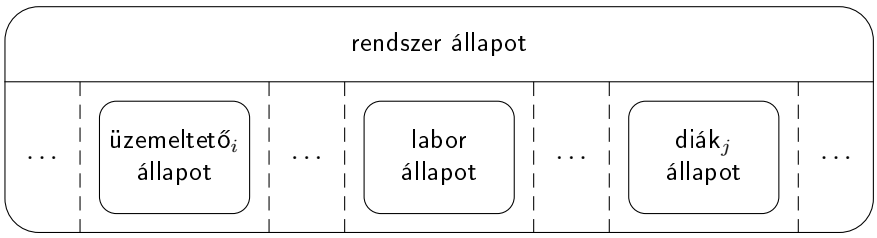
A használ és a karbantart relációk egymást kölcsönösen kizárják, amit megszorítással fejezhetünk ki. Így a 30.2. ábrán látható osztálydiagramhoz jutunk.

30.2.2. Dinamikus modell

A rendszer állapotait az üzemeltetők, a diákok és a labor állapotai együtt határozzák meg (30.3. ábra). A diákok és az üzemeltetők a rendszer folyamatai, így az állapotaik a folyamatok tevékenységeinek felelnek meg. A labor az erőforrás, amelynek állapotai adják meg a szinkronizációs feltételeket.

Egy diáknak két állapotát különböztethetjük meg.

- A diák a tanulmányait végzi, vagy a labor előtt várakozik. Az állapot neve legyen: vár.



30.3. ábra. A rendszer állapotai

- A diák a labor egyik gépén dolgozik, azt használja. Az állapot neve: használ.

Az állapotok közötti átmeneteket a kezd és az elmegy események indukálják.

Az üzemeltetőknek három állapotuk lehet.

- Az üzemeltető valahol máshol dolgozik. Az állapot: dolgozik.
- Az üzemeltető karbantartási igényt jelzett. Az állapot: igényel. (Ez egy passzív állapot, a sorra kerülésére várakozik.)
- Az üzemeltető a karbantartja a labor gépeit. Az állapot: karbantart.

Az igénylés akció vezet a dolgozik állapotból az igényel állapotba. A karbantartás kezdete (karbantartás) köti össze az igényel és karbantart állapotokat. A karbantartás befejezésekor (végez) kerülünk a karbantart állapotból a dolgozik állapotba.

A labor állapotát a bent tartózkodó diákok száma és az üzemeltetők tevékenységei határozzák meg. Jelölje t a bent tartózkodó diákok számát, r a karbantartást igénylő üzemeltetők számát, és w a karbantartást végző üzemeltetők számát.

A diákok szerint három állapotot különböztethetünk meg.

- Nincs diák a laborban. Az állapot neve: üres, invariánsa: $\mathsf{t} = 0$.
- Nincs szabad gép, a labor tele van. Az állapot neve: tele, invariánsa: $\mathsf{t} = \mathsf{db}$.
- A labor egy köztes állapotban van. Az állapot neve: normál, invariánsa: $0 < \mathsf{t} < \mathsf{db}$.

Az üzemeltetők kapcsán is három állapotot azonosíthatunk.

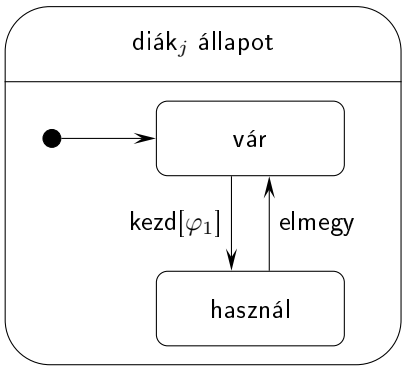
- Senki sem igényel karbantartást, és senki sem tartja karban a labort. Az állapot neve: OK, invariánsa: $\mathsf{r} = 0 \wedge \mathsf{w} = 0$.
- Karbantartást igényeltek, és senki sem tartja karban a labort. Az állapot neve: igényelt, invariánsa: $\mathsf{r} > 0 \wedge \mathsf{w} = 0$.
- A labort karbantartják. Az állapot neve: karbantartva, invariánsa: $\mathsf{w} = 1$.

Az állapotátmenetek feltételei a következők.

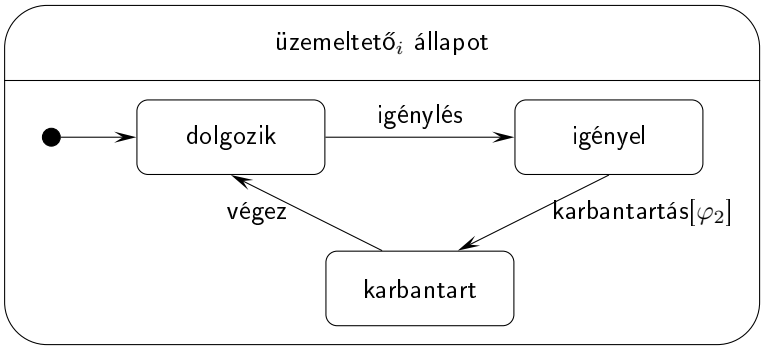
- Egy diák beléphet a laborba, és elkezdhet egy gépet használni (kezd esemény), ha a labor nincs tele, és az állapota OK, azaz $\varphi_1 : \neg\, in\ \mathsf{tele} \wedge in\ \mathsf{OK}$ feltétel fennáll. Másképpen: $\varphi_1 : \mathsf{t} < \mathsf{db} \wedge \mathsf{r} = 0 \wedge \mathsf{w} = 0$.
- Egy üzemeltető megkezdheti a karbantartást (karbantartás esemény), ha a labor üres és senki más sem tartja karban, azaz $\varphi_2 : in\ \mathsf{üres} \wedge \neg\, in\ \mathsf{karbantartva}$ feltétel teljesül. Másképpen: $\varphi_2 : \mathsf{t} = 0 \wedge \mathsf{w} = 0$.
- A labor megtelik, ha csak egy szabad gép volt, és egy diák azt elkezdi használni. A feltétel: $\varphi_3 : \mathsf{t} = \mathsf{db} - 1$.
- A labor üres lesz, amikor az utolsó diák elhagyja, azaz $\varphi_4 : \mathsf{t} = 1$.
- A labor állapota igényelt lesz egy karbantartás befejeződésekor, ha van még karbantartási igény, azaz $\varphi_5 : \mathsf{r} > 0$.
- A labor állapota OK lesz, ha egy karbantartás befejeződik és nincs több karbantartási igény, azaz $\varphi_6 : \mathsf{r} = 0$.

A fenti feltételek közül φ_1 és φ_2 a folyamatok szinkronizációs feltételeit adják meg.

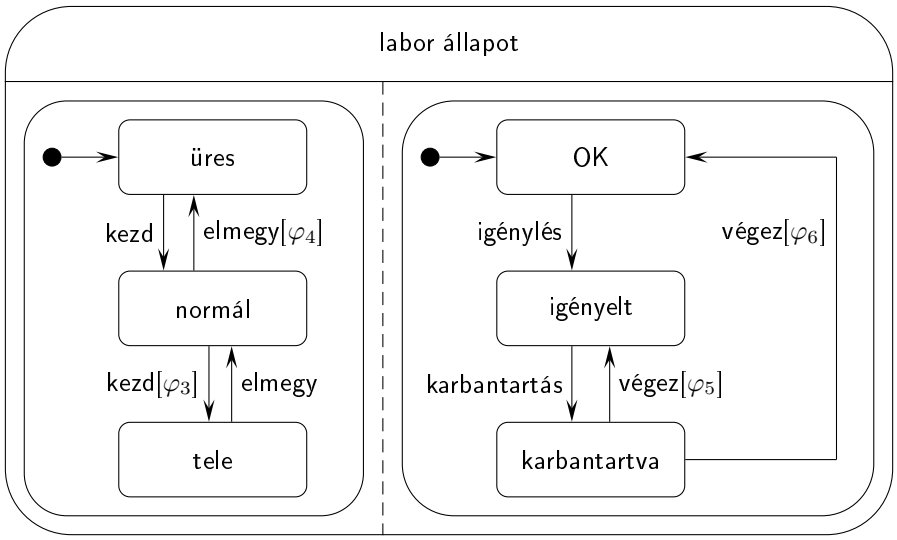
Az állapotdiagramok a 30.4–30.6. ábrákon láthatóak.



30.4. ábra. Egy diák állapotdiagramja



30.5. ábra. Egy üzemeltető állapotdiagramja



30.6. ábra. A labor állapotdiagramja

30.2.3. Absztrakt program

A dinamikus modell részét képező állapotdiagram megalkotásakor bevezettünk három változót az állapotok leírásához:

- t : a laborban tartózkodó diákok száma,
- r : a karbantartási igényt benyújtott üzemeltetők száma,
- w : a labort karbantartók száma.

Ezen változók kezdeti értékei a következők:

$\mathsf{w} = 0 \wedge \mathsf{r} = 0 \wedge \mathsf{t} = 0.$

Tegyük fel, hogy az üzemeltetők tényleges száma n, a diákok száma pedig m. Ekkor a program váza a 30.7. ábrán, a folyamatok vázai pedig a 30.8. ábrán láthatóak.

Az atomi utasításokat, az állapotdiagram megfelelő akcióit és az átmeneti feltételeket (azaz őrfeltételeket) mutatja a 30.9. ábra. Az absztrakt programban az atomi utasításokat őrfeltételes utasításokkal valósítjuk meg. Az őrfeltételek garantálják a helyes ütemezést. Az állapotdiagram megfelelő akcióihoz tartozó előfeltételek adják meg az **await** utasítás őrét.

A folyamatok vázára alkalmazva a megfelelő transzformációkat a 30.10. ábrán látható eredményre jutunk. Az absztrakt programot megkapjuk, ha ezeket a folyamatokat helyettesítjük a 30.7. ábra programjába.

A konkrét programot megkaphatjuk az **await** utasítások, illetve az egyes tevékenységek implementálásával.

```
w ← 0; r ← 0; t ← 0
parbegin
      üzemeltető_1; || ... || üzemeltető_n; ||
      diák_1; || ... || diák_m;
parend
```

30.7. ábra. A program váza

```
üzemeltető_i:
      while true do
            dolgozik;
            igényel;
            karbantart;
      od
```

```
diákt_j:
      while true do
            vár;
            használ;
      od
```

30.8. ábra. A folyamatok vázai

atomi utasítás	akció	őrfeltétel
$\langle \mathsf{w} \leftarrow \mathsf{w} - 1 \rangle$	karbantart.exit	true
$\langle \mathsf{r} \leftarrow \mathsf{r} + 1 \rangle$	igénylés	true
$\langle \mathsf{r} \leftarrow \mathsf{r} - 1;\ \mathsf{w} \leftarrow \mathsf{w} + 1 \rangle$	karbantart.entry	$\mathsf{w} = 0 \wedge \mathsf{t} = 0$
$\langle \mathsf{t} \leftarrow \mathsf{t} + 1 \rangle$	használ.entry	$\mathsf{t} < \mathsf{db} \wedge \mathsf{w} = 0 \wedge \mathsf{r} = 0$
$\langle \mathsf{t} \leftarrow \mathsf{t} - 1 \rangle$	használ.exit	true

30.9. ábra. A program atomi utasításai

```
üzemeltető_i:
    while true do
        dolgozik;
        await true then r ← r + 1 ta;
        await w = 0 ∧ t = 0 then r ← r − 1; w ← w + 1 ta
        karbantart;
        await true then w ← w − 1 ta;
    od
```

```
diák_j:
    while true do
        vár;
        await t ≠ db ∧ w = 0 ∧ r = 0 then t ← t + 1 ta;
        használ;
        await true then t ← t − 1 ta;
    od
```

30.10. ábra. Az absztrakt folyamatok

30.3. Második esettanulmány

Ebben az egyszerű példában bemutatjuk miként lehet konkrét programot előállítani. A választott implementációs nyelv a Java, a feladat megoldása során ezt végig figyelembe vesszük. A modellt úgy készítjük el, hogy minél kevesebb nyitott kérdést hagyjon az implementációra.

A párhuzamos folyamatokat szálakkal (*Thread*) valósítjuk meg, a szinkronizációt a *wait*, *notify* műveletekkel. Ezekről elegendő azt tudni, hogy egy szál *synchronized* művelete biztosítja a kölcsönös kizárást a művelet végrehajtása alatt, és ha egy ilyen műveletben a *wait* szerepel, akkor a szál blokkolódik, amíg egy *notify* üzenetet nem kap. (Ezt kiadhatjuk például az objektum egy másik műveletében, amit egy másik objektum hív meg.)

Készítsünk egy metróvonal forgalmát szimuláló programot a következő leírás alapján! A metróvonal állomásokat tartalmaz rögzített sorrendben. A vonalon járatok közlekednek, az egyszerűség érdekében csak egyirányú forgalmat vizsgálunk. Az állomások között utasok közlekednek a járatok segítségével. Egy utas egy kiinduló állomásra érkezik, és egy célállomáson száll le. Egy járat bemehet egy állomásra, ha a bevezető lámpa zöld, ellenkező esetben az állomás előtt várakozik. Az állomáson az odatartó utasok leszállnak a járatról, a várakozó utasok felszállnak, ha van hely. A leszállás, és a felszállás időt vesz igénybe, amelynek mértéke arányos az utasok számával. A felszállás befejeztével a járat elhagyja az állomást, ha a kivezető lámpa zöld, különben várakozik az állomáson. A bevezető lámpa pirosra vált, ha egy járat bemegy az állomásra, és zöldre, ha elhagyja azt. A kivezető lámpa pirosra vált, ha egy járat elhagyja az állomást, ezután adott idő elteltével zöldre vált. Miután az utolsó járat bemegy egy állomásra, az állomás bezár, azaz újabb utas nem érkezhet oda.

30.3.1. Statikus modell

A szimuláció szempontjából nem kell az utasokat külön-külön vizsgálnunk, csak a számuk érdekes az állomásokon, illetve a járatokon. A célállomást akkor határozzuk meg, amikor egy járat egy állomásra ér. Ekkor a járaton tartózkodó utasok adott százaléka leszáll.

Megfelelő arányokkal ugyanarra az eredményre jutunk, mintha az utasok célállomásait egyesével kezelnénk az utas megjelenésekor.

Az állomásokhoz tartozó bevezető és kivezető lámpák közös tulajdonságait érdemes kiemelni egy közös ősosztályba, amelyből származtatjuk a két speciális osztályt. Így első megközelítésben a következő osztályokhoz jutunk:

- **Metró**: a metróvonal;
- **Állomás**: a metróvonal állomásai;
- **Járat**: a vonalon közlekedő járatok;
- ***Lámpa***: a lámpák közös tulajdonságait leíró absztrakt osztály;
- **BeLámpa**: a bevezető lámpák;
- **KiLámpa**: a kivezető lámpák.

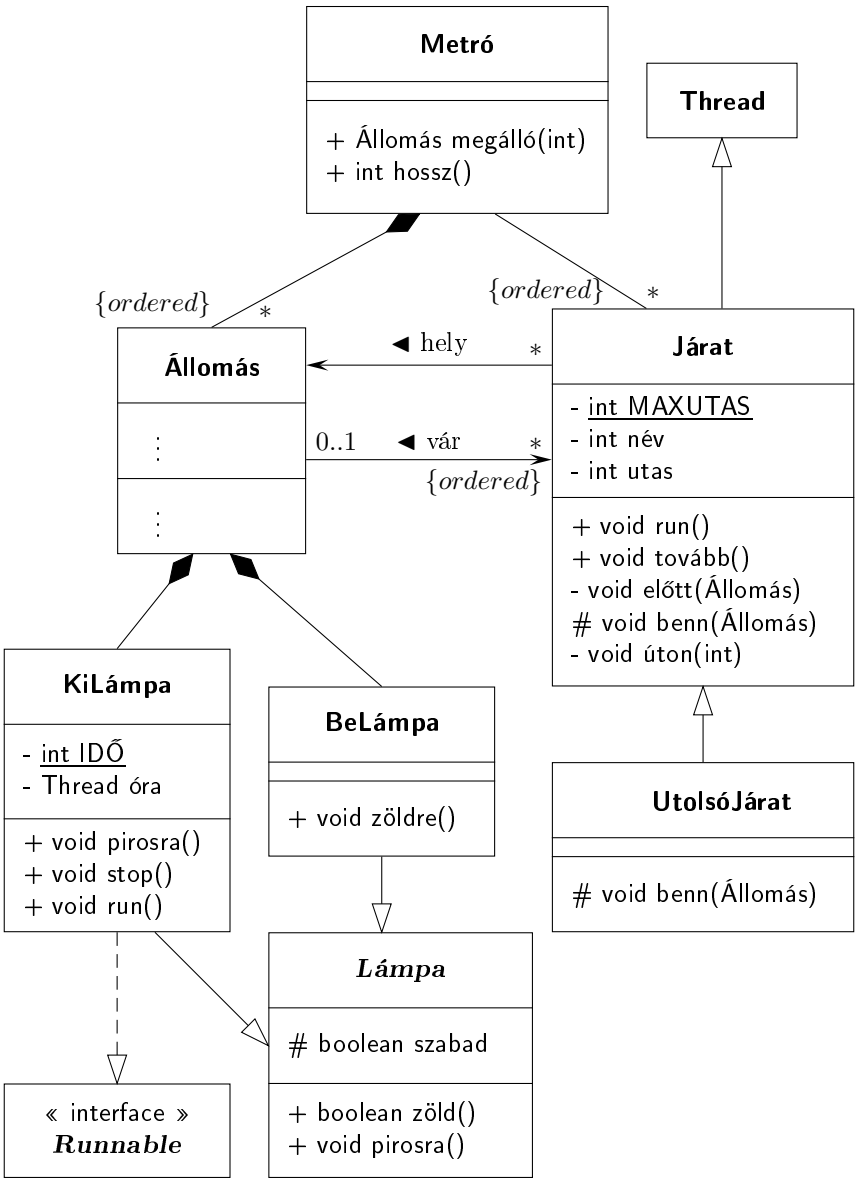
Minden állomást jellemez a neve, az ott várakozó utasok száma, a bizonyos időközönként érkező új utasok száma, a beérkező járatról leszálló utasok aránya, a következő állomás távolsága, illetve, hogy nyitva van-e. Egy állomás műveletei: a megfelelő adattagok értékeit megadó függvények, az állomás bezárása (*bezár*), az utasok mozgásával kapcsolatos eljárások (*felszáll*, *érkeznek*), a járatokkal kapcsolatos műveletek (*szabad*, *vár*, *bejön*, *mehet*, *kimenne*, *kienged*, *elhagy*). Az áttekinthetőség érdekében az **Állomás** osztály esetében az adattagokat és műveleteket nem tüntetjük fel az osztálydiagramban.

A járatok jellemzői a maximálisan szállítható utasok száma (ez minden járat esetén ugyanaz az érték), az azonosításra szolgáló név, a szállított utasok száma, és a még el nem hagyott állomás. Minden járat külön folyamat lesz, ezért a **Thread** osztályból kell származtatnunk. A járatok műveletei: a **Thread** osztály *run* művelete, állomás előtti várakozás (*előtt*), állomáson tartózkodás (*benn*), és két állomás közti utazás (*úton*). Ezen kívül a várakozások feloldásához hell egy *tovább* művelet. Az utolsó járat csak abban tér el a többi járattól, hogy belépéskor be kell zárni az állomást. Ezt megtehetjük a járatok osztályának specializálásával, ha a *benn* műveletet az elején kiegészítjük az állomás bezárásával. Így az **UtolsóJárat** osztályhoz jutunk.

A lámpák közös jellemzője az aktuális állapot, ami leírható egy logikai értékkel (zöld $\equiv$ igaz), annak lekérdezése, hogy zöld-e a lámpa, illetve a lámpa pirosra állítása. Bevezető lámpák esetén egy zöldre állító eljárás kell még. Kivezető lámpák esetén a pirosra állítást ki kell egészíteni egy óra elindításával, ami a megadott idő leteltével zöldre állítja a lámpát. (Ezért nem kell külön zöldre állító művelet.) Az óra egy külön szál (**Thread**) lesz, ezért ennek az osztálynak meg kell valósítania a ***Runnable*** felületet (*run* művelet). Az idő az összes lámpa esetén megegyezik, ezért ezt osztályszintű változóban tarthatjuk nyilván.

A metróvonal biztosítja a szimulációhoz szükséges műveleteket, például a vonalon található állomások számát (javahossz). A járatokban a hely asszociációt megvalósító *hely* adattag egy index lesz, a szokásos mutató helyett. (Így egyszerűbb a következő állomás meghatározása.) Ezért a **Metró** osztály rendelkezik egy *megálló* művelettel, amely a megfelelő indexű állomást adja meg.

Az osztályok közötti kapcsolatok és azok tulajdonságai értelemszerűen adódnak a leírásból, így a 30.11. ábrán látható osztálydiagramhoz jutunk. Az osztályok közötti kapcsolatok megvalósítása értelemszerűen újabb adattagok bevezetését igényli, ezeket az UML-nek megfelelően nem tüntettük fel a diagramban. A **KiLámpa** osztály esetén a kompozíciós kapcsolat mindkét irányban navigálható, mert a lámpa értesíti az állomást, ha zöldre vált. Az osztályban a kapcsolatot a *hol* attribútum valósítja meg. Az **Állomás** osztályban a vár kapcsolatot a *sor* adattaggal implementáljuk.

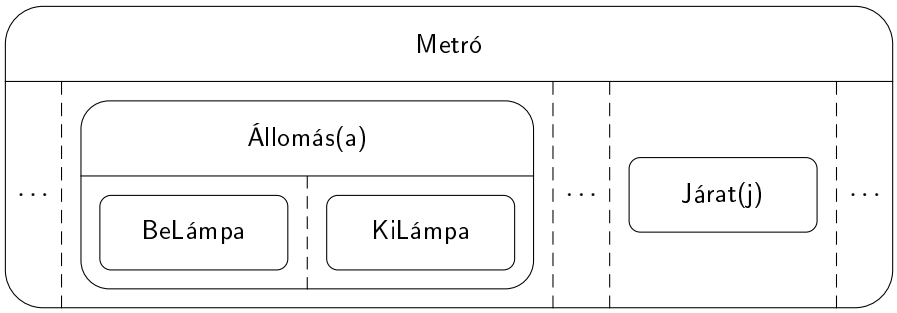


30.11. ábra. A metróvonal osztálydiagramja

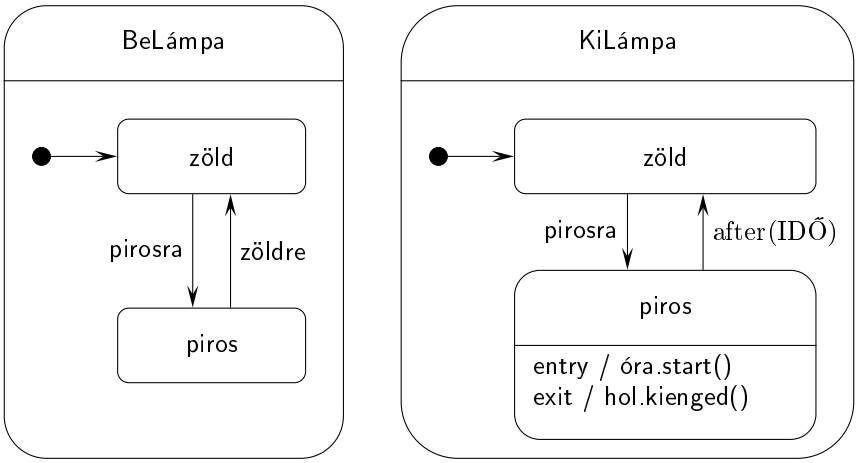
30.3.2. Dinamikus modell

A rendszer állapotait az állomások és a járatok állapotai együttesen határozzák meg. Egy állomás állapotának két összetevője van: a bevezető és a kivezető lámpa állapota (30.12. ábra). A nyitottság és az utasok száma is lényeges, de ezek egy attribútum értékével kifejezhetőek és kezelhetőek.

A lámpáknak két állapotuk van: zöld, vagy piros. A kezdeti állapot a zöld. Ebből a pirosra esemény hatására mennek át a piros állapotba. A bevezető lámpák a zöldre üzenet esetén kerülnek a zöld állapotba, a kivezető lámpák adott idő eltelte után. A kivezető



30.12. ábra. A metróvonal állapotdiagramja



30.13. ábra. A lámpák állapotdiagramja

lámpák esetén ezt kifejezhetjük az állapotinvariánssal:

$I(\mathsf{piros}) = \quad t < \mathsf{IDŐ},$

ahol t a pirosra esemény óta letelt időt jelenti. Ennek megfelelően az állapotátmenet eseménye: after(IDŐ). A piros állapotba lépve el kell indítani az időt megadó órát, az állapot megszűnésekor értesíteni kell az állomást. Ezt megadhatjuk a diagramban, mint az állapothoz tartozó entry és exit akciót (30.13. ábra).

Egy járat sorban halad végig az állomásokon, amíg az utolsó állomást el nem hagyja. Ez három állapot ismétlődését jelenti:

- megérkezett az állomáshoz, és az állomás előtt arra vár, hogy bemehessen;
- az állomáson benn tartózkodik;
- az állomást elhagyja, és úton van a következő állomás felé.

Az állapotokhoz a következő invariánsok tartoznak. Ezek megszűnésekor hagyja el az állapotot, és kerül a következő állapotba. (A rövidség érdekében a *hely* attribútumot nem indexként, hanem mutatóként kezeljük a formulákban. Pontosan *Metró.megálló(hely)*-et kellene írnunk.)

előtt: az állomás bevezető lámpája piros vagy van előtte várakozó járat, azaz:

$$\neg\big(\mathsf{hely.belámpa}\ in\ \mathsf{zöld}\ \wedge\ (\mathsf{hely.sor} = \emptyset \vee\ \mathsf{hely.sor}(0) = this)\big).$$

benn: az utasok még nem fejezték be a mozgást, vagy a kivezető lámpa piros, azaz, ha le, illetve fel adja meg a leszálló, illetve felszálló utasok számát, és *idő* az ehhez szükséges időt, akkor:

$$t < \mathit{idő}(\text{le, fel})\ \vee\ \mathsf{hely.kilámpa}\ in\ \mathsf{piros}.$$

úton: még nem telt le az utazási idő (táv), azaz:

$$t < \mathsf{táv}.$$

Ennek megfelelően a következő esetekben kerül egy állapotból egy másikba egy járat:

előtt→*benn*: when(hely.belámpa *in* zöld $\wedge$ (hely.sor $= \emptyset \vee$ hely.sor$(0) = this$));

benn→*úton*: after(*idő*(le, fel))[hely.kilámpa *in* zöld];

úton→*előtt*, *úton*→*befejezés*: after(táv).

Ezeket az áttekinthetőség érdekében nem tüntetjük fel az állapotdiagramban.

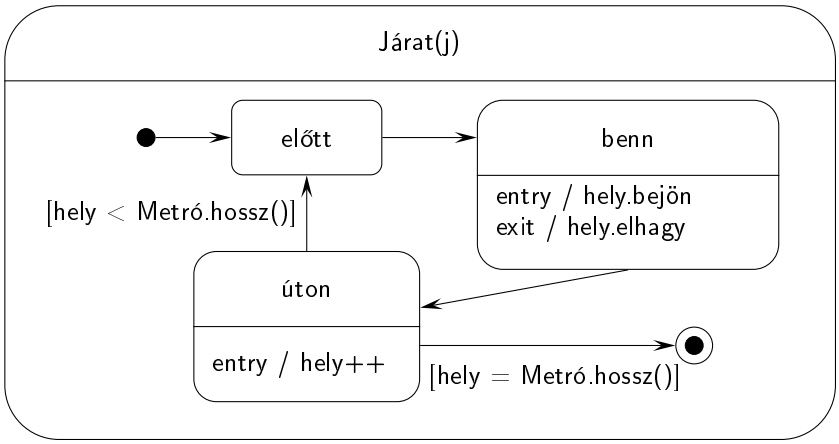
Ha egy járat bemegy egy állomásra, akkor erről értesíteni kell a bevezető lámpát. Ezt az állomáson keresztül tehetjük meg, erre szolgál a *bejön* művelet. Másképpen fogalmazva a bevezető lámpa pirosra eseménye akkor következik be, amikor egy járat belép az állomásra:

$$\mathsf{BeLámpa.pirosra} \equiv \mathsf{hely.bejön} \equiv \mathsf{benn.entry}.$$

Ha egy járat elhagyja az állomást (benn.exit), akkor a kivezető lámpát pirosra kell állítani, a bevezető lámpát pedig zöldre:

$$\mathsf{BeLámpa.zöldre}, \mathsf{KiLámpa.pirosra} \subset \mathsf{hely.elhagy} \equiv \mathsf{benn.exit}.$$

Egy út megkezdésekor a járat a következő állomással kerül kapcsolatba, azaz az úton állapot belépési akciója a *hely* növelése. Az eddigiek alapján a 30.14. ábrán látható állapotdiagramhoz jutunk.



30.14. ábra. Egy járat állapotdiagramja

```
parbegin
      járat_1; || ... || járat_n;
parend
```

30.15. ábra. A program váza

```
járat_j:
      while hely < Metró.hossz() do
            előtt(hely);
            benn(hely);
            úton(hely);
      od
```

```
előtt(hely):
      await hely.szabad() then skip ta;
```

```
benn(hely):
      // bemegy az állomásra
      // utasok le- és felszállása
      await hely.mehet() then hely.elhagy() ta;
```

30.16. ábra. Egy járat folyamat váza

30.3.3. Absztrakt program

Az absztrakt program elkészítésekor a folyamatok közötti szinkronizációt vizsgáljuk, ezért az ebből a szempontból érdektelen részleteket elhanyagoljuk.

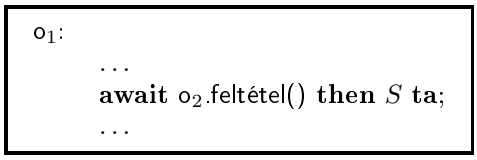
A rendszer párhuzamos folyamatait a járatok alkotják. Ezen kívül a **KiLámpa** osztályban is található folyamat (az idő mérésére szolgál), de ez független a többitől, ezért az absztrakt programban nem vesszük figyelembe (30.15. ábra).

Egy járat folyamat a dinamikus modellben szereplő három állapot ciklikus ismétlődéséből áll, amíg a járat végig nem halad a metróvonalon. Az előtt állapotnak egy várakozás felel meg, a benn állapotban a megfelelő tevékenységeket kell elvégezni, és a szükséges szinkronizációs feltétel teljesülése után lehet elhagyni az állomást, az úton állapothoz egyszerű műveletek és egy késleltetés tartozik. Ez utóbbit nem részletezzük a folyamat vázában (30.16. ábra).

30.3.4. Konkrét program

Az implementáció során először a szinkronizációs utasítások megvalósításával kell foglalkoznunk. A feladatban egyszerű esettel állunk szemben, ugyanis bármely folyamat szinkronizációs feltétele egy másik objektumtól függ. Ezt kihasználjuk az implementáció során. (Ha a feltételt több objektum állapota határozza meg, akkor másként kell eljárni.) Legyen o_1 a szinkronizációs feltételt tartalmazó objektum, o_2 a feltételt megadó objektum (30.17. ábra).

Az o_1 objektum tevékenységeinek szinkronizálandó részét egy *synchronized* eljárásba (legyen ez *proc*) helyezzük. Ebben ellenőrizzük az őrfeltételt, és ha az nem teljesül, akkor



30.17. ábra. A szinkronizáció absztrakt formája

```
private synchronized void proc(...)
{
    if ( !o2.feltétel() )
    {
        o2.vár(this);
        try { wait(); } catch (InterruptedException e) {}
    }
    // S
}

public synchronized void tovább() { notify(); }
```

30.18. ábra. A szinkronizáció megvalósítása o_1 objektumban

```
public synchronized void vár(O1 o1) { obj = o1; }

public synchronized void feltételOK()
{
    if ( obj != null ) obj.tovább();
    obj = null;
}
```

30.19. ábra. A szinkronizáció megvalósítása o_2 objektumban

értesítjük o_2 objektumot annak a *vár* műveletével, és várakozunk (*wait*). Ekkor blokkolt állapotba kerül a szál, amíg egy *notify* üzenetet nem kap. A várakozást követik az atomi utasítás elemei. Készítünk egy eljárást, amellyel a várakozó állapotból felébresztjük az objektumot (*notify*), és tovább léphetünk. Legyen ez a *tovább* művelet. Az O_1 osztály megfelelő elemeit mutatja a 30.18. ábra.

Az o_2 objektum esetén tárolni kell az értesítendő objektumot a *vár* műveletben, és az őrfeltétel teljesülésekor a *feltételOK* műveletben (ami lehet egy másik objektumtól kapott üzenet hatása) értesíteni kell a várakozó objektumot. Az O_2 osztály megfelelő műveletei láthatóak a 30.19. ábrán.

A ***Lámpa*** és a **BeLámpa** osztály megvalósítása (30.20. ábra) adódik a modellből. Az állapotváltozató akciók lesznek az adattagot módosító műveletek (*pirosra*, és a **BeLámpa** osztály esetén *zöldre*), amit az állapotot lekérdező (*zöld*) művelettel kell kiegészíteni. Ezek mindegyike szerepel az UML modellben, csak a megvalósítást kell megadnunk, ami értelemszerű. Az absztrakt osztály konstruktora biztosítja, hogy minden lámpa kezdeti állapota a zöld legyen.

Tulajdonképpen a **KiLámpa** osztály implementációja is triviális, csak a beágyazott szál kezelésére kell ügyelnünk, a Java ajánlásokat kell figyelembe venni. A „keveredés” elkerü-

```
public abstract class Lámpa
{
    protected boolean szabad;
    public Lámpa() { szabad = true; }
    public boolean zöld() { return szabad; }
    public void pirosra() { szabad = false; }
}

public class BeLámpa extends Lámpa
{
    public BeLámpa() {}
    public void zöldre() { szabad = true; }
}

public class KiLámpa extends Lámpa implements Runnable
{
    private static final int IDŐ = 8;
    private Thread óra;
    private Állomás hol;
    public KiLámpa(Állomás hol)
    {
        óra = null; this.hol = hol;
    }
    public synchronized void pirosra()
    {
        super.pirosra();
        if ( óra == null )
        {
            óra = new Thread(this);
            óra.start();
        }
    }
    public synchronized void stop()
    {
        if ( óra != null ) óra.interrupt();
        óra = null; notifyAll();
    }
    public void run()
    {
        if ( óra != Thread.currentThread() ) return;
        try { óra.sleep(IDŐ * 1000); }
        catch (InterruptedException e) {}
        stop(); szabad = true; hol.kienged();
    }
}
```

30.20. ábra. A ***Lámpa***, a **BeLámpa** és a **KiLámpa** osztályok megvalósítása

lése érdekében, minden időmérés kezdetén egy új szálat (*óra*) indítunk el. A szál egyetlen feladata az időt mérni. Az idő leteltével a szálat az ajánlások szerint leállítjuk. A konstruktorban a szálnak extremális értéket adunk, illetve a kompozíciós kapcsolat navigálásához szükséges értéket (*hol*) állítjuk be a paraméternek megfelelően. Az időmérést a *pirosra* műveletben kell megkezdenünk. A *stop* művelet a szál ajánlott leállítási módja, a *run* művelet pedig maga az időmérés. A megvalósítás minden egyéb eleme szerepel a modellben (30.20. ábra).

Az **Állomás** osztályban az állomás előtt várakozó járatokat tartalmazó *sor* adattagot a könyvtári **Vector** osztály segítségével valósítjuk meg. Ezen kívül az állomásnak kell értesítenie az ott tartózkodó szerelvényt, ha a kivezető lámpa zöldre vált. Erre szolgál a *ki* adattag. A többi attribútum adódik a leírásból.

A *ki* adattag csak akkor mutat járatra, ha az állomáson tartózkodik járat, és az a kivezető lámpára vár. (Ellenkező esetben az érték *null*.) Ennek a beállítására szolgál a *kimenne* művelet, amit a járat hív meg, ha letelt az utasok mozgásából adódó várakozási idő, és a kivezető lámpa akkor még piros.

```
import java.util.Vector;

public class Állomás
{
    private String név;
    private BeLámpa belámpa;
    private KiLámpa kilámpa;
    private int utas;
    private int le;
    private int új;
    private int táv;
    private Vector sor;
    private boolean nyitva;
    private Járat ki;

    public Állomás(String név, int le, int új, int táv)
    {
        this.név = név; this.le = le;
        this.új = új; this.táv = táv;
        belámpa = new BeLámpa();
        kilámpa = new KiLámpa(this);
        utas = új;
        nyitva = true;
        sor = new Vector();
        ki = null;
    }

    public String Név() { return név; }
    public void bezár() { nyitva = false; }
    public boolean nyitott() { return nyitva; }
```

30.21. ábra. Az **Állomás** osztály megvalósítása

```
    public int felszállna() { return utas; }
    public int távolság() { return táv; }
    public int leszáll() { return le; }
    public void felszáll(int u) { utas -= u; }

    public synchronized void érkeznek()
    {
        if ( nyitva ) utas += új;
    }

    public synchronized boolean szabad()
    {
        return sor.size() == 0 && belámpa.zöld();
    }

    public boolean mehet() { return kilámpa.zöld(); }

    public void bejön() { belámpa.pirosra(); }

    public synchronized void elhagy()
    {
        kilámpa.pirosra();
        if ( sor.size() > 0 )
        {
            ((Járat)sor.get(0)).tovább();
            sor.remove(0);
        }
        else belámpa.zöldre();
    }

    public void vár(Járat j) { sor.add(j); }

    public void kimenne(Járat j) { ki = j; }

    public synchronized void kienged()
    {
        if ( ki != null ) ki.tovább();
        ki = null;
    }
}
```

30.22. ábra. Az **Állomás** osztály megvalósítása (folytatás)

A *ki*, *mehet*, *kimenne* és *kienged* elemek valósítják meg az állomás elhagyására vonatkozó szinkronizációt. Az állomás belépési szinkronizációját a *sor*, *szabad*, *vár* elemek és az *elhagy* művelet megfelelő része implementálja.

Gyakorlati okokból az *elhagy* művelet implementációja is eltér a modelltől. Csak akkor állítja a bevezető lámpát zöldre, ha nincs várakozó járat az állomás előtt. Ellenkező esetben

a sor első elemét beengedi.

Ha egy járat nem tud bemenni az állomásra, akkor az állomás *vár* műveletével csatlakozik az állomás előtt várakozó járatokhoz.

A konstruktor művelet paraméterei szolgálnak az állomás nevének, forgalmi jellemzőinek (leszálló utasok aránya, érkező utasok száma), illetve a következő állomás távolságának beállítására. Az osztály megvalósítása a 30.21–30.22. ábrákon látható.

A **Járat** osztály megvalósítása is adódik a modellből a következő kiegészítésekkel. Itt helyeztük el a rendszer működésének figyelésére szolgáló üzenetek kiírását. Csak akkor várakoztatunk egy járatot a *wait* művelettel, ha ez szükséges, azaz, ha nem tud azonnal belépni egy állomásra, illetve nem tudja azt elhagyni az utasok mozgása után. Mindkét

```
public class Járat extends Thread
{
    private static final int MAXUTAS = 400;
    private int név;
    private int hely;
    private int utas;

    public Járat(int név)
    {
        this.név = név; hely = 0; utas = 0;
        System.out.println(név + " járat elindult.");
        start();
    }

    public void run()
    {
        Állomás a;
        while ( hely < Metró.hossz() )
        {
            a = Metró.megálló(hely);
            előtt(a);
            benn(a);
            úton(a.távolság());
        }
        System.out.println(név + " járat leállt.");
    }

    private synchronized void előtt(Állomás a)
    {
        if ( !a.szabad() )
        {
            a.vár(this);
            try { wait(); } catch (InterruptedException e) {}
        }
    }
```

30.23. ábra. A **Járat** osztály megvalósítása

```
    protected synchronized void benn(Állomás a)
    {
        a.bejön();
        System.out.println(név + " járat " + a.Név() +
                           " állomásra bement.");
        int le = (a.leszáll() * utas) / 100;
        utas -= le;
        int fel = Math.min(MAXUTAS - utas, a.felszállna());
        System.out.println(név + " járat " + a.Név() + " állomás: "
            "leszáll " + le + ", felszáll " + fel + " utas.");
        utas += fel;
        a.felszáll(fel);
        try { sleep(Math.max(2, (fel + le) * 25)); }
        catch (InterruptedException e) {}
        if ( !a.mehet() )
        {
            a.kimenne(this);
            try { wait(); } catch (InterruptedException e) {}
        }
        System.out.println(név + " járat " + a.Név() +
                           " állomást elhagyta.");
        a.elhagy();
    }

    private void úton(int táv)
    {
        if ( ++hely == Metró.hossz() ) return;
        try
        {
            sleep(táv * 1000);
            System.out.println(név + " járat " +
            Metró.megálló(hely).Név() + " állomáshoz ért.");
        }
        catch (InterruptedException e) {}
    }

    public synchronized void tovább() { notify(); }
}
```

30.24. ábra. A **Járat** osztály megvalósítása (folytatás)

esetben az állomás a *tovább* művelettel tudja aktivizálni a járatot. A megvalósítást mutatják a 30.23–30.24. ábrák.

Az **UtolsóJárat** osztály implementációja értelemszerű, a **Járat** *benn* műveletét kell kiegészítenünk az állomás bezárásával (30.25. ábra).

A **Metró** osztályban az **Állomás** objektumokat definiálunk egy konstans tömbben, ezzel implementáljuk a két osztály közötti kompozíciós kapcsolatot. A **Metró** osztály megvalósításából (30.26. ábra) a *main* műveletet nem tartalmazza a modell. Ebben adott számú

```
public class UtolsóJárat extends Járat
{
    public UtolsóJárat(int név) { super(név); }

    protected synchronized void benn(Állomás a)
    {
        a.bezár();
        super.benn(a);
    }
}
```

30.25. ábra. Az **UtolsóJárat** megvalósítása

járatot indítunk el késleltetéssel, illetve az állomásokra érkező utasokról gondoskodunk. Az egyszerűség érdekében a szimuláció paramétereit a programban konstans adatokkal, illetve véletlen számokkal helyettesítjük.

Az volt a célkitűzésünk, hogy a modell támogassa az implementációt, minél kevesebb döntés maradjon a modellen kívül. Látható, hogy kellően pontos és részletes modell esetén az implementáció nagy része valóban egyszerűen előállítható. Ugyanakkor vannak bizonyos részek, amelyek nem jeleníthetőek meg az eddig megismert UML elemekkel. Ezek egyik csoportját az implementációs nyelv sajátosságaiból, másik részét ez ebből, vagy egyéb megfontolásokból származó változások alkotják.

UML alapú tervek készítésekor lényegében két lehetőségünk van. Áttekintő vázlatos tervet készítünk, ami „csak” dokumentációs célokat szolgál, vagy pontos és részletes tervet állítunk elő, ami lényegében megfelel az elkészítendő programnak.

Az első megoldás hátránya, hogy a program készítésekor bekövetkezett változások nem feltétlen jelennek meg a tervben. (Ezt elkerülni lényegében két – esetleg még több – elem konzisztens módosításával lehet a rendszer előállítása során.) Ha ez a helyzet, akkor egy idő múlva az UML modellnek elég kevés köze lesz a tényleges rendszerhez, ami nyilvánvalóan káros.

A második módszer akkor jelent előnyt az elsővel szemben, ha a tervből tudjuk előállítani a futtatható kódot, vagy legalábbis annak nagyon nagy részét.

Az UML modellező eszközök támogatnak valamilyen szintű kódgenerálást. A generált kódot vagy ki kell egészíteni, vagy a „teljes” kódot állítják elő egyéb ismeretek felhasználásával. Az egyéb ismeretek egyrészt bizonyos kódgenerálási ismereteket (például elkészített sablonok) jelentenek minden esetben. Ezeket „szakértők” helyezik el a fejlesztőrendszerben az adott alkalmazásnak megfelelően. Ez még a generált kód funkcionális kiegészítését igényli. Az ismeretek másrészt tartalmazhatnak olyan elemeket, amelyek a hiányzó funkcionalitást leírását „beemelik” a modellbe. Ez utóbbi módszert követik az úgynevezett végrehajtható UML-ben (eXecutable UML) (1.7. rész), röviden xUML-ben.

```
import java.util.Random;

public class Metró
{
    private static Állomás[] megállók =
    {
        new Állomás("Első", 0, 20, 7),
        new Állomás("Második", 15, 20, 9),
        new Állomás("Harmadik", 25, 25, 8),
        new Állomás("Negyedik", 40, 20, 11),
        new Állomás("Ötödik", 50, 10, 8),
        new Állomás("Hatodik", 100, 0, 0),
    };

    private static final int MAXJÁRAT = 10;

    public static Állomás megálló(int i)
    {
        return megállók[i];
    }

    public static int hossz() { return megállók.length; }

    public static void main(String[] args)
    {
        int járat = 1;
        Random rnd = new Random();
        new Járat(járat);
        while ( megállók[hossz() - 1].nyitott() )
        {
            try
            {
                Thread.sleep((rnd.nextInt(3) + 3) * 1000);
            }
            catch (InterruptedException e) {}
            for ( int i = 0; i < megállók.length; i++ )
                megállók[i].érkeznek();
            if ( járat < MAXJÁRAT && rnd.nextInt(3) == 0 )
            {
                if ( ++járat == MAXJÁRAT )
                    new UtolsóJárat(járat);
                else
                    new Járat(járat);
            }
        }
    }
}
```

30.26. ábra. A **Metró** osztály megvalósítása

31. Konkurens rendszerek mintái

Az előzőekben megismerkedtünk a GoF mintákkal, és láttunk példát ezek bővítésére új mintákkal. Lehetőségünk van további minták megadására nemcsak a megadott osztályokon belül, hanem új osztályokat is bevezethetünk. Ilyenek lehetnek például:

- konkurens rendszerekhez kapcsolódó minták,
- GUI minták,
- adatbázis kezeléssel kapcsolatos minták.

A továbbiakban konkurens rendszerek mintáival foglalkozunk. Ezek nem felelnek meg az eddig megismert tervmintáknak, ugyanis nem ugyanazon az absztrakciós szinten helyezkednek el, mert egy implementációs környezethez, esetünkben a Java nyelvhez, kapcsolódnak.

31.1. Eseményalapú aszinkron hívás (Event-Based Asyncronous Call)

Cél

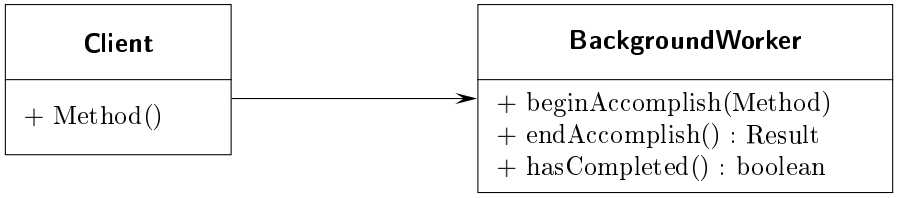
Hosszú ideig futó műveletek aszinkron futtatása úgy, hogy a hívást egy új szálra delegáljuk, ezzel elkerüljük a hívó szál blokkolódását.

Felhasználhatóság

Egy művelet futtatása túl sok ideig tart, és nem akarjuk, hogy addig a végrehajtást végző szál blokkolt legyen. A művelet eredményére nincs szükségünk azonnal, azt csak a később kell használnunk.

Szerkezet

A művelet futtatását egy másik szálat használó objektum felügyeli (BackgroundWorker), amelytől le lehet kérdezni, hogy befejeződött-e a művelet (hasCompleted).



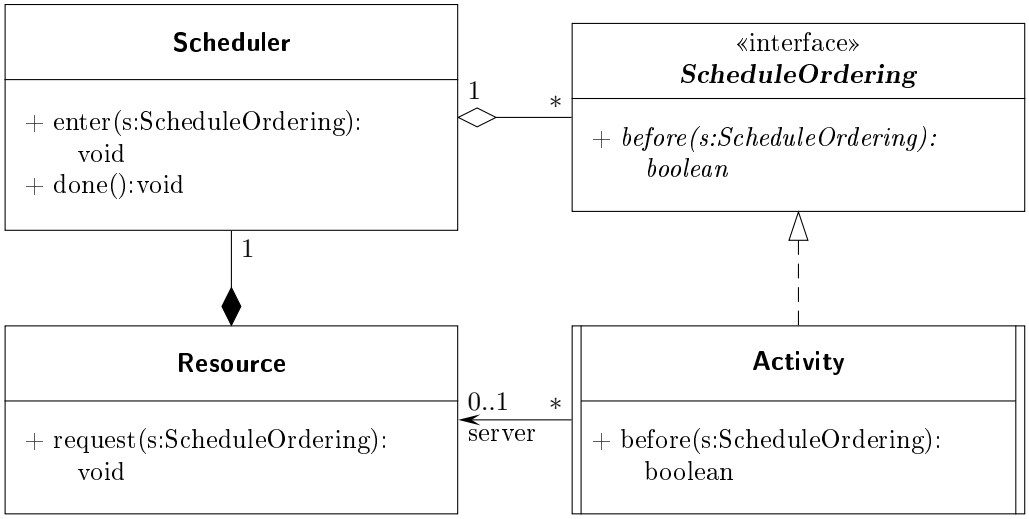
31.2. Ütemező (Scheduler)

Cél

Egy objektummal vezérelni szálak hozzáférési sorrendjét egy szekvenciális kódhoz (erőforráshoz). Az objektum sorba állítja a várakozó szálakat. Az ütemező minta egy mechanizmust biztosít az ütemezési eljárás megvalósítására. Ez a mechanizmus független az ütemezés konkrét módjától. A szálak egymást kölcsönösen kizárják a használatból, egyszerre legfeljebb egy szál férhet hozzá az erőforráshoz.

A mintában szereplő megoldás alapötlete egy ütemező (scheduler) objektum létrehozása, amelyben szerepel egy művelet (*enter*), ami nem adja vissza a vezérlést, amíg a hívó szálra nem kerül a sor a használatban. Rendelkezik egy másik művelettel is, amely a használat végét jelzi (*done*). Az osztott erőforrás egy ütemező objektumot tartalmaz, amelynek a fenti műveleteit hívja egy szál igényének kielégítése során (*request*). A szálak egy közös interfészt (***ScheduleOrdering***) valósítanak meg, ami az ütemezés sorrendjét szabályozza (*before*). A szálak ismerik az erőforrást, azt a *request* művelet hívásával használják.

Szerkezet



Elemek

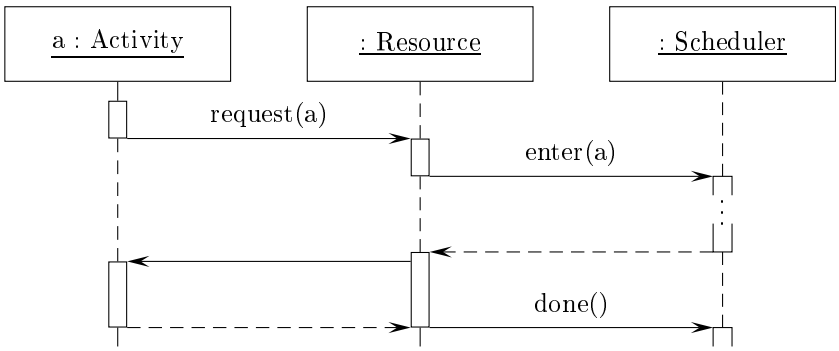
ScheduleOrdering: Az ütemezéshez használt megelőzési relációt megadó interfész.

Scheduler: Az ütemezést megvalósító objektum.

Resource: A közös erőforrás.

Activity: Az erőforrást használni akaró tevékenységek, szálak.

Együttműködés



Példa kód

A Java nyelvű megvalósításban a tevékenységekben (szálakban) egyszerű várakozással (sleep) szimuláljuk az erőforrás használatát, illetve az egyéb teendők elvégzését. Egy konkrét esetben ezeket kell helyettesíteni az erőforrás megfelelő műveleteinek hívásával az erőforrás használat során. A tevékenységek osztályszintű *stopSimulation* műveletével állíthatjuk le az összes tevékenységet.

```
public interface ScheduleOrdering
{
    public boolean before(ScheduleOrdering s);
}

public final class SingleResource
{
    private Scheduler scheduler = new Scheduler();

    public void request(ScheduleOrdering s)
    {
        try
        {
            scheduler.enter(s);
            ((Activity)s).using();
            scheduler.done();
        }
        catch (InterruptedException e) {}
    }
}

public class Activity extends Thread implements ScheduleOrdering
{
    private static final int UNIT = 100;
    private static boolean running = true;
    public static void stopSimulation() { running = false; }

    private int precedence;
    private String name;
    private int usetime;
    private int worktime;
```

```
    private SingleResource server;

    public Activity(int precedence, String name, int usetime, int worktime,
                    SingleResource server)
    {
        this.precedence = precedence; this.name = name; this.server = server;
        this.usetime = usetime * UNIT; this.worktime = worktime * UNIT;
    }

    public boolean before(ScheduleOrdering s)
    {
        if ( s instanceof Activity ) return precedence > ((Activity)s).precedence;
        return false;
    }

    public void run()
    {
        while ( running )
        {
            working();
            server.request(this);
        }
    }
    public void using()
    {
        System.out.println(name + ": using");
        try { sleep(usetime); } catch (InterruptedException e) {}
    }

    private void working()
    {
        System.out.println(name + ": working");
        try { sleep(worktime); } catch (InterruptedException e) {}
    }

    public void waiting()
    {
        System.out.println(name + ": waiting");
    }
}

import java.util.Vector;

public class Scheduler
{
    private Thread running = null;
    private Vector<ScheduleOrdering> waiting;
    private Vector<Thread> threads;

    public Scheduler()
    {
        waiting = new Vector<ScheduleOrdering>();
        threads = new Vector<Thread>();
    }
```

```
    public void enter(ScheduleOrdering s) throws InterruptedException
    {
        Thread current = Thread.currentThread();
        synchronized (this)
        {
            if ( running == null ) { running = current; return; }
            ((Activity)s).waiting();
            int place = findPlace(s);
            threads.add(place, current);
            waiting.add(place, s);
        }

        synchronized (current)
        {
            if (current != running) current.wait();
        }
    }

    synchronized public void done()
    {
        if ( running != Thread.currentThread() )
            throw new IllegalStateException("Wrong Thread");
        if ( threads.isEmpty() ) { running = null; }
        else
        {
            running = threads.remove(0); waiting.remove(0);
            synchronized (running) { running.notify(); }
        }
    }

    synchronized protected int findPlace(ScheduleOrdering s)
    {
        int i;
        for ( i = 0; i< waiting.size(); i++ )
            if ( s.before(waiting.get(i)) ) break;
        return i;
    }
}
```

A minta előnyei a következők:

- Nincs szükség külön feltételekre, állapotokra a tevékenységekben, azokat az ütemező objektum kezeli.
- Az erőforrás használatához csak a *request* művelet hívása szükséges, minden egyéb teendő rejtett. Ez nem csak egyszerűsíti a használatot, de a konzisztenciát is biztosítja, ugyanis az erőforrás a használat végén felszabadul, külön művelet hívása nélkül.

A minta hátrányai:

- Szinte minden vezérlés az ütemezőbe kerül, így a tevékenységek kezelése (felfüggesztése) is. Ehhez ismerni kell azok kezelési módját, azaz nem csak a ***ScheduleOrdering*** interfészt használjuk. Esetünkben ez az implementációban a *threads* tömbben tárolt

szálakat *Thread*-eket, és kezelésüket jelenti. Ez tulajdonképpen egy csalás, hiszen sehol sem szerepelt, mégis használjuk a **Thread** osztályt.

- Több lehetséges művelet esetén a ***ScheduleOrdering*** interfész nem elégséges (ahogy azt az implementációban láttuk), hiányzik a tevékenység értesítésének művelete (*using*), amellyel biztosítjuk, hogy a tevékenység megfelelő módon használhassa az erőforrást. Ez csak akkor valósítható meg, ha az erőforrás ismeri a tevékenység konkrét osztályát, amely tartalmaz ilyen műveletet.

- Az ütemezőben kihasználjuk, hogy az egyes tevékenységek futását egyszerűen fel tudjuk függeszteni. Ez Java szálak esetén igaz, de nem feltétlen teljesül egyéb esetekben (például C++ szálak, Ada taszkok).

- Eddig ugyan még nem vizsgáltuk az időhöz kötött várakozást, de a minta ezt nyilvánvalóan nem támogatja. Ebben az esetben ugyanis adott idő letelte után a tevékenység lemond az erőforrás használatáról, és másként folytatja működését. A mintában a tevékenység nem kap arról jelzést, hogy miért került hozzá vissza a vezérlés, így nem tudja miként kell a továbbiakban ténykednie. (A tevékenység szempontjából ugyanaz következik be amikor a megadott idő lejár, illetve amikor az erőforráshoz hozzáfér.)

31.3. Aktív objektum (Active Object)

Cél

Szétválasztani műveletek hívását és végrehajtását úgy, hogy a műveletek futtatását egy külön objektumra delegáljuk, amely ezután azokat párhuzamosan tudja futtatni. Ezzel az objektum szinkronizált hozzáférése is egyszerűsödik. Bizonyos esetekben a mintát szokás *Concurrent Object*-nek is nevezni.

Felhasználhatóság

A műveletek egy objektumhoz kötődnek, azonban egymástól függetlenek, így akár párhuzamosan is futtathatóak, vagy egy művelet sokszori futtatása szükséges, és a futtatások nincsenek hatással egymásra.

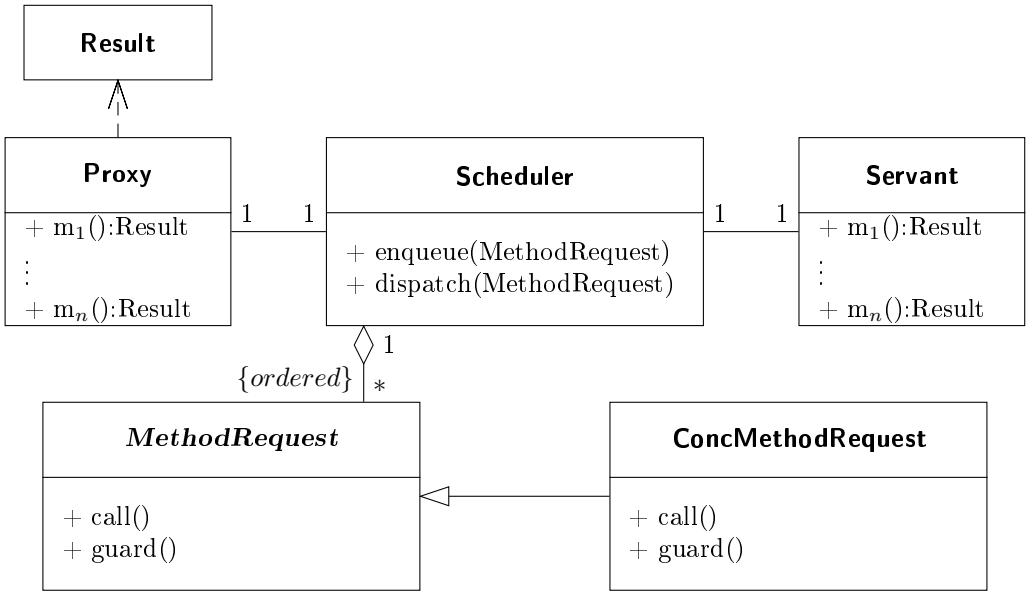
Sok alkalmazás használja az aktív objektumokat a szolgáltatás javítására, például lehetővé téve, hogy a kliensnek több kérését egyszerre szolgálja ki. Ahelyett, hogy egy objektum szekvenciálisan szolgálja ki a kéréseket, párhuzamos objektumok felelnek a kliensek kéréseire. Ennek megvalósítására három feltétel szükséges:

- a sok erőforrást igénylő műveletek feldolgozása nem blokkolhatja a konkurens objektumokat;

- a közösen használt objektumok műveleteit nem lehet közvetlenül meghívni, hanem azokat parancsokként egy köztes rétegen keresztül, ütemezve továbbítjuk;

- az alkalmazásban a párhuzamosság nem függhet a szoftver/hardver környezettől.

Szerkezet



Elemek

Proxy: Hozzáférési felület a műveletekhez. Ezen keresztül éri el a kliens az aktív objektum szolgáltatásait.

MethodRequest: A műveleteket végrehajtó objektumok közös felülete.

ConcMethodRequest: Egy konkrét kérésnek megfelelő parancsobjektum. Példányait a Proxy objektum hozza létre a kliens kérésének hatására.

Scheduler: Gondoskodik a műveletek megfelelő sorrendben történő futtatásáról. Tartalmaz egy sort, amelybe elhelyezi a végrehajtandó műveleteket, és innen választja ki a végrehajtandó tevékenységet.

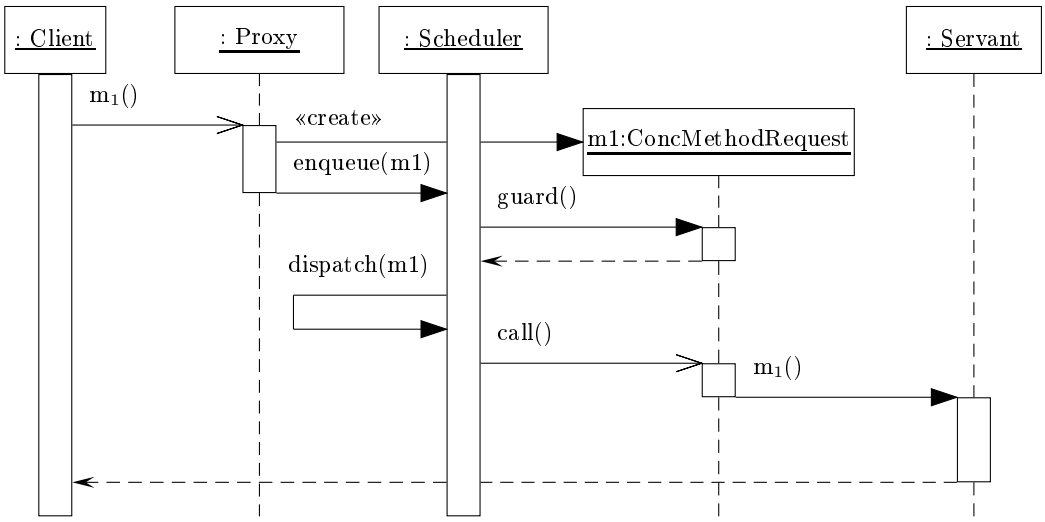
Servant: Megvalósítja azokat a műveleteket, amelyeket a Proxy szolgáltat a felhasználónak. Ezeket használják a parancsok.

Result: A kliens ezen keresztül érheti el a művelet eredményét.

Egy felületen (Proxy) keresztül biztosítunk hozzáférést a műveletekhez, amelyeket művelet végrehajtó objektumok (MethodRequest) futtatnak. A műveletek megfelelő sorrendben való futtatásáról egy ütemező (Scheduler) objektum gondoskodik. Ez tartalmaz egy sort, amelybe elhelyezi az igényeket

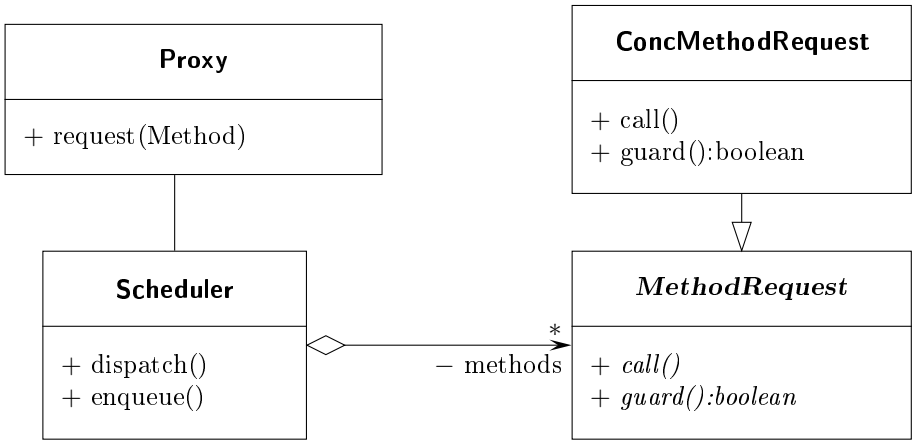
Együttműködés

A felület (Proxy) létrehozza a parancsokat, továbbítja a kéréseket az ütemező felé, amely sorba teszi azokat, és meghívja a műveleteket.



A mintát használják például Symbianban (Activ object framework), illetve .NET környezetben az aszinkron függvényhívás a minta egy megvalósítása.

Egy alternatív lehetőség, ha a Proxy a klienseknek nem a megvalósító objektum műveleteit adja meg, hanem a felület egy művelet megadását teszi lehetővé. Ezt a szerkezetet szemlélteti a következő osztálydiagram.



31.4. Blokkolás (Balking)

Cél

Párhuzamos rendszerek esetén a szinkronizáció miatt a szálak blokkolódnak, amíg egy feltétel nem teljesül, azaz megfelelő állapotba nem kerülünk. Ha ezt el kell kerülnünk, akkor használhatjuk a blokkolás mintát[1]. Ekkor csak abban az esetben engedélyezzük egy másik szálnak művelet végrehajtását, ha az objektum egy adott állapotban van, ellenkező esetben felfüggesztés nélkül visszaadjuk a vezérlést.

[1] Ha a blokkolódás elfogadható, és várhatunk, akkor használható például az őrzött felfüggesztés minta.

Felhasználhatóság

Amennyiben egy művelet elvégzése adott állapothoz, vagy állapotokhoz kötött, és el akarjuk kerülni, hogy a rossz állapot miatt kivétel keletkezzen – például egy fájlból úgy akarunk olvasni, hogy fájlt még nem nyitottuk meg –, vagy szabályozni akarjuk, hogy egy művelet mikor legyen végrehajtható az objektumon.

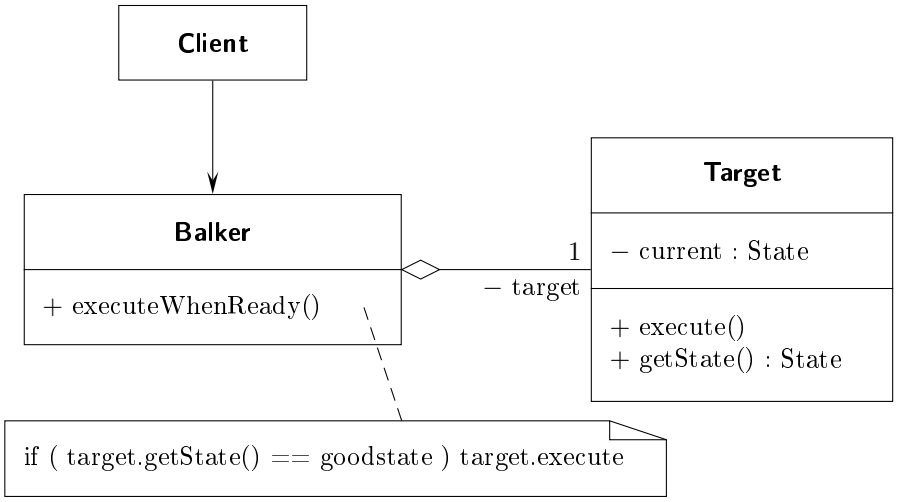
Tegyük fel, hogy egy objektum műveletét két különböző szálon hívjuk meg, és a második hívás megelőzi az elsőt. Három módon kezelhető a probléma:

1. Az első hívást végrehajtjuk, a másodikat figyelmen kívül hagyjuk.
2. Az első hívás után végrehajtjuk a másodikat is.
3. Az első hívás végrehajtását megszakítjuk, és végrehajtjuk a másodikat.

A blokkolás minta lényegében az első megoldást valósítja meg, azaz ha egy szál olyan állapotba helyez egy objektumot, amikor más nem férhet hozzá, akkor a beérkező igényeket nem hajthatjuk végre, azok elutasításra kerülnek felfüggesztés nélkül.

Szerkezet

A kliens és a célobjektum közé elhelyezünk egy blokkoló objektumot (Balker), amely nem engedi a műveletet végrehajtani, ha a célobjektum nincs a megfelelő állapotban. Ennek érdekében a blokkoló információt szerez a célobjektumtól, például lekérdezi a annak állapotát (getState).



Egyszerűbb esetekben a blokkoló objektum elhagyható, és a célobjektum kiszolgáló műveletében kezeljük a blokkolást, ahogy azt a következő Java kódrészlet mutatja.

```
public class TaskObject
{
    private boolean jobInProgress = false;
    public void job()
    {
        synchronized(this)
        {
```

```
            if ( jobInProgress ) return;
            jobInProgress = true;
        }
        //Execute job
        ...
    }

    void jobCompleted() { jobInProgress = false; }
}
```

A minta előnyei:

- Az objektum műveletének hívása valamilyen eredménnyel lefut.
- A művelet végrehajtását semmi sem késlelteti (ha nem áll fenn a megfelelő feltétel, azonnal visszatér).
- Ha az objektum állapota nem megfelelő, akkor a beérkező kérések figyelmen kívül maradnak.

31.5. Őrzött felfüggesztés (Guarded Suspension)

Cél

Zárolással, valamint feltétellel rendelkező műveletek megfelelő végrehajtása.

Felhasználhatóság

Amennyiben egy művelet végrehajtása nemcsak a zárolás feloldását igényli, hanem bizonyos feltételeket is teljesítenie kell, ekkor hasonlóan járhatunk el, mint a Blokkolás tervminta esetén. Azaz:

- Művelet végrehajtása feltételhez kötött, és ha a feltétel nem teljesül, a művelet nem fejeződne be.
- Egy másik művelet elvégzése után a feltétel teljesülne, de a másik művelet nem hajtható végre az előző művelet zárolása miatt. Az akadályt okozó állapot így nem szűnhet meg, aminek holtpont az eredménye.
- Őrzött felfüggesztés esetén a blokkoló műveletet nem futtatjuk, hanem várakoztatjuk, amíg a feltételek nem teljesülnek.

Szemléltetésként tekintsünk egy olyan sort, amelyet párhuzamosan több szál használ. Ebben az esetben például egy elem behelyezése (push), illetve kivétele (pull) egymást kölcsönösen kizáró műveletek, azaz valamilyen módon mindkettőnek zárolnia kellene a sort, hogy nehogy inkonzisztens állapot jöjjön létre. Ugyanakkor azt is szeretnénk, hogy a pull művelet üres sorra is működjön, és ne váltson ki kivételt, hanem várjon, amíg egy elem a sorba kerül.

Ebben az esetben a problémát az okozná, hogy az üres sorra kiadott pull művelet a zárolás miatt folyamatosan blokkolja a sort, így a push művelet sem hajtható végre, azaz sosem kerül elem a sorba, tehát a pull sosem fejeződik be, holtponthoz jutunk.

A probléma lehetséges megoldása, hogy a pull műveletet ténylegesen nem hajtjuk végre, hanem várunk, amíg elem kerül a sorba.

Szerkezet

Amint a zárolás feloldódik, és megkezdődik a művelet futtatása, ellenőrizzük az előfeltételt, és blokkoljuk a műveletet, amíg az nem teljesül.

```
public class Example
{
    synchronized void guardedMethod()
    {
        while ( !preCondition() )
        {
            try
            {
                wait();   //Continue to wait
            }
            catch (InterruptedException e) {}
        }
        //task implementation
    }

    synchronized void alterObjectStateMethod()
    {
        //Change the object state
        notify();   //Inform waiting threads
    }
}
```

Példa

A következő Java kódrészlet szemlélteti a példaként bemutatott párhuzamos sor lehetséges megvalósítását.

```
import java.util.ArrayList;

public class Queue
{
    private ArrayList data = new ArrayList();

    synchronized public void put(Object obj)
    {
        data.add(obj);
        notify();
    }

    synchronized public Object get()
    {
        while ( data.size() == 0 )
        {
            try { wait(); } catch (InterruptedException e) {}
        }
        return data.remove(0);
    }
    ...
}
```

A minta előnye, hogy egy osztály műveletei szinkronizáltak lesznek, és elkerülhető az akadályt okozó állapot miatti holtpont. A minta egy lehetséges problémája, hogy nem tartalmaz döntést arról, hogy több várakozó folyamat esetén melyik kapja meg a vezérlést. Ez feloldható az Ütemező minta alkalmazásával. Az őrzött felfüggesztés nem használható úgy, hogy a védett műveleteket szinkronizált műveletekből hívnánk, mert a külső zárolás megmaradna. Ennek megfelelően a minta használatakor erre ügyelni kell, azaz az őrzött műveleteket zárolás (synchronized) mentesen szabad használni.

A minta jellegéből adódóan az őrzött felfüggesztés csak abban az esetben használható, ha nem okoz gondot, hogy egy művelet csak egy állapot fennállása esetén kerül végrehajtásra, illetve a várakozás időtartama korlátok között tartható, becsülhető.

31.6. Duplán ellenőrzött zárolás (Double Checked Locking)

Cél

Egy objektum zárolásának lekérdezése és esetleges zárolása a lehető legkevesebb erőforrás használatával. A kritikus szakaszt csak egyszer kell zárolni, de garantálni kell, hogy párhuzamosan futó szálak esetén csak egyszeres zárolás jöjjön létre.

Felhasználhatóság

Az objektum zárolásának állandó lekérdezése költséges művelet – mivel az is egy szinkronizált művelet, ezért csak akkor engedélyezhető, ha a szál hozzáférhető –, ezért célszerű a zárolás állapotának előzetes lekérdezését nem biztonságos módon végezni, ezáltal jelentős erőforrásokat takaríthatunk meg. Sokszor a Lusta inicializáció tervmintával együtt szokás alkalmazni.

A minta olyan alkalmazások esetén alkalmazható, amikor a következő feltételek teljesülnek.

- Az alkalmazás tartalmas olyan kritikus szakaszokat, amelyeket szekvenciálisan kell végrehajtani.
- Előfordulhat, hogy több szál egyidejűleg próbálja végrehajtani a kritikus szakaszt.
- A kritikus szakasz csak egyszer hajtandó végre.
- A kritikus szakaszt minden esetben zárolni jelentős költség növekedéssel (futási idő) járna.
- Lehetséges egy „könnyű”, de biztonságos ellenőrzést végrehajtani a zárolás előtt.

Az Egyke minta esetén fordulhat elő párhuzamos környezetben, hogy több szál is létrehoz egy-egy objektumot. Ha megvizsgáljuk az Egyke C++ nyelvű megvalósítását (ami emlékeztetőként megismétlünk), akkor látható, hogy a több szál esetén előfordulhat a következő. Az egyik szál az *instance* művelet végrehajtásakor megvizsgálja a példány létezését, és miután nincs még példány létrehozza azt. Azonban a példány létrejötte előtt a másik szál is meghívja a műveletet, elvégzi a vizsgálatot, és miután nincs még példány, az is létre fog hozni egyet. Így két példány jöhet létre, ami ellentmond az Egyke minta céljának. Bizonyos esetekben ez akár végzetes eredménnyel is járhat.

```
class Singleton
{
public:
   static Singleton *instance (void)
   {
      if ( instance_ == 0 ) instance_ = new Singleton;
      return instance_;
   }
private:
   static Singleton *instance_;
};
```

A leírt probléma egyszerű megoldásának tűnhet a kritikus rész zárolása, ahogy azt a következő kódrészlet mutatja.

```
class Singleton
{
public:
   static Singleton *instance (void)
   {
      // A guard konstruktora automatikusan zárolást biztosít
      Guard<Mutex> guard (lock_);
      // Egyidőben csak egy szál tartózkódhat ebben a kritikus szakaszban
      if ( instance_ == 0 ) instance_ = new Singleton;
      return instance_;
      // A guard destruktora automatikusan feloldja a zárolást
   }
private:
   static Mutex lock_;
   static Singleton *instance_;
};
```

Ebben az esetben azonban a példány művelet minden egyes hívása zárolással és annak feloldásával jár, noha erre csak egyszer (a példány létrehozásakor) lenne szükség. Ez jelentős időveszteséggel jár.

A hatékonyságon javíthatunk, ha a példány létezését a zárolás előtt vizsgáljuk, és csak szükség esetén zárolunk, ahogy azt a következő kód mutatja, azonban az eredmény már nem szálbiztos.

```
class Singleton
{
public:
   static Singleton *instance (void)
   {
      if ( instance_ == 0 )
      {
         Guard<Mutex> guard (lock_);
         instance_ = new Singleton;
      }
      return instance_;
   }
```

```
private:
   static Mutex lock_;
   static Singleton *instance_;
};
```

Ebben az esetben előfordulhat ugyanaz, amit már vizsgáltunk, azaz két szál „egyidőben" végzi el a létezési vizsgálatot. Mind a kettő ugyanazt az eredményt kapja, és továbblép. Az egyik zárol és létrehoz egy objektumot, amíg a másik blokkolódik. Miután létrejött az objektum a zárolás megszűnik, és a másik szál zárol és hoz létre újabb objektumot.

A megoldás a példány létezésének kétszeri ellenőrzése: egyszer zárolás nélkül, majd a zárolt részen belül újra. Ezzel elkerülhető a többszöri példányosítás.

```
class Singleton
{
public:
   static Singleton *instance (void)
   {
      if ( instance_ == 0 )
      {
         Guard<Mutex> guard (lock_);
         if ( instance_ == 0 ) instance_ = new Singleton;
      }
      return instance_;
   }
private:
   static Mutex lock_;
   static Singleton *instance_;
};
```

Az előzőekkel ellentétben, ha az első ellenőrzés több esetben is teljesülne, utána a zárolás miatt, csak egy szál hozhat létre objektumot, mert csak akkor engedi ezt a második ellenőrzés. A további szálak a zárolás után léphetnek csak be a kritikus szakaszba, és a példány ekkor már létezik, ezért a második ellenőrzés miatt nem jöhet létre újabb objektum. Ugyanakkor ez a megoldás nem okoz felesleges zárolásokat, hiszen a példány létrejötte utáni hívásokban csak az első vizsgálatot kell végrehajtanunk, amihez nem tartozik zárolás.

Szerkezet

A szinkronizált kritikus szakasz elé behelyezünk egy aszinkron lekérdezést, és a zárolt részen belül megismételjük a tesztet.

```
if ( !flag )
{
   // zárolás
   if ( !flag )
   {
      // kritikus szakasz
      flag = true;
   }
   // zárolás feloldása
}
```

Az alkalmazás során biztosítanunk kell, hogy a feltétel vizsgálata atomi (megszakíthatatlan) legyen, ellenkező esetben a zárolás nem kerülhető el. Figyelni kell arra is, hogy bizonyos fordítók az optimalizálás során a második (zárolt) vizsgálatot nem értékelik ki, hanem az első eredményét használják, ami nyilvánvalóan nem kívánt működést eredményez. Ennek elkerülésére használható a *volatile* kulcsszó, amint azt a Java példában szemléltetjük.

A minta előnyei:

- Garantálja a kritikus szakasz egyszeri végrehajtását többszálú környezetben is a feltétel kétszeri ellenőrzésével.
- Minimalizálja a zárolást, ha a feltétel atomi.

Java példa

Vizsgáljuk meg a minta alkalmazását Java nyelven. A következő program szemlélteti az egyszerű implementációt.

```
public class Example
{
    private ExObject obj = null;
    public ExObject getObject()
    {
        if ( obj == null )
        {
            synchonized(this)
            {
                if ( obj == null ) obj = new ExObject();
            }
        }
        return obj;
    }
}
```

Ez a megoldás nem biztonságos JDK 6-nál korábbi változatokban. Ha az objektum létrehozása (*ExObject* konstruktora) időben elhúzódik, akkor egy másik szál hozzáférhet az objektumhoz, ugyanis az értékadás már megtörtént. Így olyan objektumot próbál majd használni, ami még nincs megfelelően inicializálva.

A probléma megoldására szolgál a *volatile* kulcsszó. Ennek hatása hasonlít a *synchronized* blokkhoz kulcsszóéra, a következő eltérésekkel:

- A *volatile* változó lehet objektum, és lehet elemi típus is, akár *null* is lehet az értéke.
- A szinkronizáció a változó minden egyes hozzáférésekor (írás, olvasás, lekérés) bekövetkezik.

A *volatile* változók a szálak által közösen használt memória részre kerülnek, és a környezet biztosítja, hogy minden ilyen változóra vonatkozó író vagy olvasó művelet kizárólag a közös területet használja.

A biztonságos megoldás ennek alapján a következő:

```
public class Example
{
    private volatile ExObject obj = null;
```

```
    public ExObject getObject()
    {
        if ( obj == null )
        {
            synchonized(this)
            {
                if ( obj == null ) obj = new ExObject();
            }
        }
        return obj;
    }
}
```

31.7. Író-olvasó zárolás (Read-Write Lock)

Cél

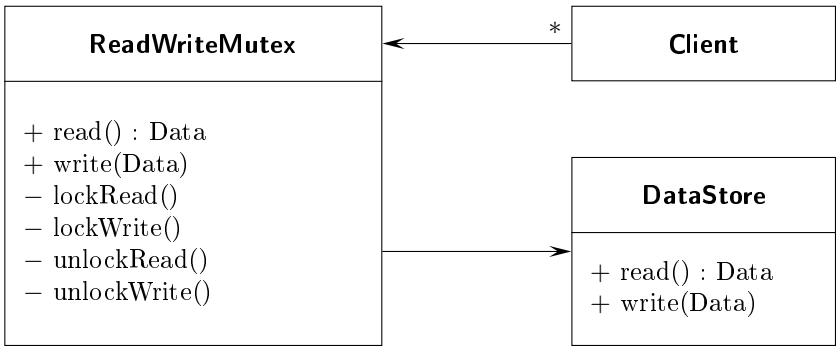
Olyan zárolás biztosítása egy objektumnak, ahol egyszerre tetszőlegesen sokan olvashatják az adatokat, de egyszerre csak egy írhatja, továbbá írás közben olvasás sem engedélyezett.

Felhasználhatóság

Amennyiben konkurens hozzáférést akarunk biztosítani olyan adatokhoz, amelyeket időnként módosítani akarunk, és a módosítás folyamata elhúzódhat.

Szerkezet

Beiktatunk egy köztes objektumot (ReadWriteMutex) az adatforrás és a kliens közé, amely nyilvántartja, hogy milyen állapotban van az adatforrás, és ennek megfelelően ad engedélyt olvasásra, illetve írásra.



31.8. Szálkészlet (Thread Pool)

Cél

Sok, rövid feladat párhuzamos végrehajtását gyorsítani. Műveletek egy halmazára korlátozott számú szál biztosítása, ezáltal mindig csak korlátolt számú művelet fut egyszerre. Objektumkészlethez hasonló, csak most nem objektumokat, hanem szálakat kezelünk.

Felhasználhatóság

Túl sok műveletet kell elvégezni, és nem akarjuk mindegyiket külön szálban futtatni – hiszen a szálak létrehozása és bezárása is erőforrást igényel –, ugyanakkor szekvenciális futtatásuk lassú lenne. Akkor is használhatjuk, ha a teljesítendő feladatok nem egyszerre keletkeznek, csak egy részük érhető el egyszerre.

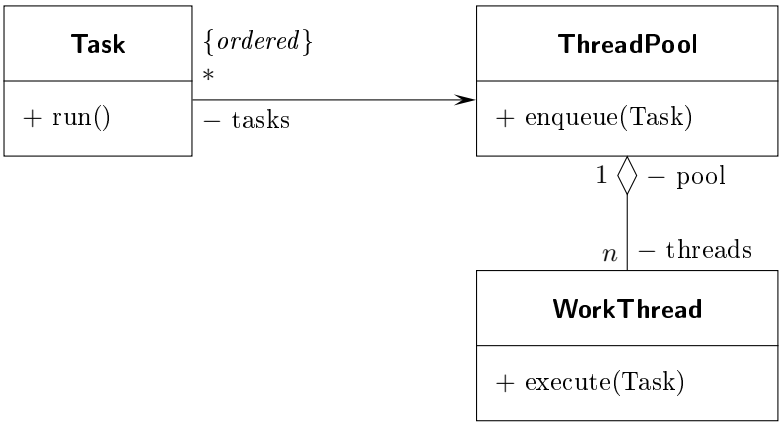
Ilyen eset lehetséges, ha például egy szervernek sok kérést kell kiszolgálnia, és minden igény viszonylag egyszerű, gyorsan elvégezhető feladat. Ekkor minden igényt egy külön szálban hajthatunk végre, azonban egy szál létrehozása és megszüntetése túl sok erőforrást igényel a feladathoz viszonyítva. Ezért egy előre adott számú szállal dolgozunk, amelyek közül egyet rendelünk egy igény feldolgozásához. A feladat befejezése után a szál visszakerül a felhasználható szálak közé.

Az eddigiek alapján a minta akkor használható, ha:

- a használható szálak száma korlátozott;
- egy szál létrehozása (megszüntetése) túl sok költséget jelent a feladathoz képest, ezért szálak létrehozását kerülni kell;
- egy feladatot végrehajtott és befejeződött szálak újrafelhasználhatóak;
- egy központi objektum képes kezelni a szálakat, azokhot feladatokat rendelni.

Szerkezet

A készletért egy objektum felel (ThreadPool), amely a kapott kéréseket (Task) elhelyezi egy sorban, és kezeli a teljesítésért felelős szálakat (WorkThread). Egy szál, amint befejezi egy kérés végrehajtását, lekéri az első kérést a sorból, és végrehajtja azt.



A minta alkalmazáskor a következő problémák merülhetnek fel.

- A szálak konkurens futása miatt holtpont alakulhat ki, ha azok nem függetlenek. (Például közös erőforrások használata.)
- A szálak számának helye megválasztása. Túl sok szál felvétele erőforrás pazarlást eredményez, túl kevés szál jelentős teljesítmény csökkenést okoz. (Bizonyos megvalósításokban a készlet mérete szükség esetén növelhető.)

- Helytelen megvalósítás miatt a szálak nem kerülnek vissza a készletbe. A szálak megmaradnak, de nem használhatóak, csak erőforrást fognak, illetve egy idő múlva kiürül a készlet.
- A készletet kezelő objektum túlterhelődhet, ha túl sok kérést kell teljesítenie.

31.9. Reaktor (Reactor)

Cél

Egy kiszolgálóhoz párhuzamosan érkező kérések teljesítése oly módon, hogy a kérésnek megfelelő teljesítő objektumhoz kerüljön az igény.

Felhasználhatóság

Különböző típusú igényeket kaphat a rendszer, amelyeket megfelelő teljesítőnek kell továbbítania, akár többnek is. A kezelők külön-külön történő elérése nem lenne hatékony a rendszer számára, ezért felügyelő objektumokat iktatunk a szerkezetbe.

Szerkezet

A kérések egy eseménykezelőhöz (Dispatcher) érkeznek, amely biztosít számukra megfelelő teljesítő erőforrásokat (EventHandler), ha azok szabadok, különben blokkolja a kérést. Az erőforrásokat specializálhatjuk a típustól függően (ConcreteEventHandler). A teljesítőket egy külön objektum felügyeli (Demultiplexer), amely az eseménykezelőnek továbbít egy erőforrást, ha az felszabadul.

31.10. Termelő-fogyasztó

Cél

Termékek (objektumok) előállításának és felhasználásának időbeli elválasztása, aszinkronitás megteremtése.

Felhasználhatóság

Ha el akarjuk kerülni egy elem előállításának és felhasználásának időbeli összekapcsolását. Az aszinkronitás lehetővé teszi a termelő és fogyasztó objektumok párhuzamos működtetését. Várakozni csak akkor kell, ha kiürül, vagy betelik a raktár.

Szerkezet

A termelő egy közbeiktatott raktárba helyezi az objektumokat, ahonnan a fogyasztó kiveheti.

32. Keretek

A tervmintáknál speciálisabb, de nagyobb újrafelhasználható tervezési egységek a *keretek* (frameworks). A keretek egyre elterjedtebbé és fontosabbá válnak. Ezeket használják fel legtöbbször az objektumelvű rendszerekben.

A keret:

- Együttműködő osztályok halmaza, amelyek egy újrafelhasználható tervet alkotnak meghatározott osztályba tartozó szoftverek számára;
- Megszabja a rendszer szerkezetét. Definiálja a teljes szerkezetet, azt osztályokra és objektumokra particionálja, megadja azok feladatait, az együttműködésük módját és a vezérlés folyamatát.
- Magában foglalja a felhasználási terület közös tervezési döntéseit. A keret előre definiálja ezeket a tervezési paramétereket, így használójának csak az adott rendszer specialitásaival kell foglalkoznia.

Ennek megfelelően keretek használata esetén terv-újrafelhasználásról beszélünk, nem pedig kód-újrafelhasználásról, noha rendszerint egy keret tartalmaz konkrét osztályokat, amelyek azonnal használhatók.

Keretek használata esetén:

- A program (szerkezetileg) fő részét használjuk fel.
- Azokat a kódrészleteket kell megírni, amelyeket a szerkezetben hívunk. Azaz adott nevű és paraméterezésű műveleteket kell előállítani, így csökkentve a meghozandó tervezési döntések számát.

Ennek eredményeképpen:

- Gyorsabban készül el a rendszer.
- Hasonló feladatot megoldó programok szerkezete is hasonló lesz.
- Egyszerűbb lesz a programok karbantartása.
- A programok egységesebbeknek tűnnek majd felhasználóik számára is.

A tervminták is és a keretek is tervezési általánosítások, de három lényeges eltérés van köztük:

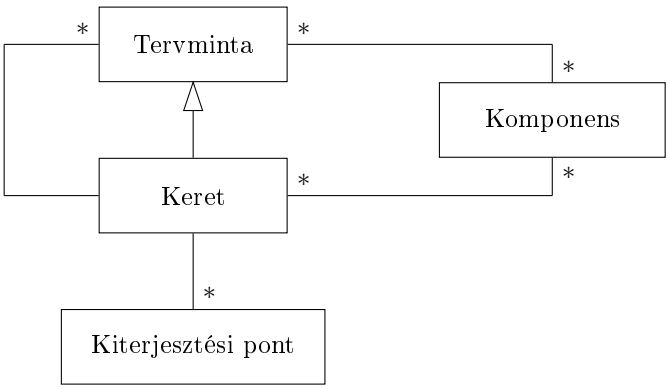
1. *A tervminták absztraktabbak, mint a keretek.* A keretekhez tartozik programkód, a tervminták esetén legfeljebb példákat lehet megadni.
2. *A tervminták kisebb szerkezeti egységek, mint a keretek.* Egy keret rendszerint több tervmintát tartalmaz, de ez fordítva nem áll fenn.

3. *A tervminták kevésbé specializáltak, mint a keretek.* A keretekhez mindig tartozik egy speciális felhasználási terület. Ezzel szemben egy tervminta gyakorlatilag tetszőleges területen használható. A tervminta nem írja elő az egész rendszer szerkezetét, szemben a kerettel.

A tervminták és a keretek is *komponensek* segítségével épülnek fel, és ezek biztosítják az egyszerű újrafelhasználhatóságot. Komponensen értjük egy rendszer (fizikai) kicserélhető részét, amely megfelel egy adott felületnek (interfész), és egyben ilyen hozzáférési felületet biztosít is. Komponenseket könnyebben fel lehet használni, ha kiegészítjük azokat olyan leírással, amely válaszol a következő kérdésekre: milyen szolgáltatásokat biztosít a komponens; milyen elvárásoknak kell megfelelnie; melyek a függőségi viszonyai; ...

A keretek ezenkívül még rendelkeznek *kiterjesztési pontokkal* is. Ezek adják meg a keret azon részeit, amelyeket módosítani kell egy adott felhasználás során, azaz leírják, hogy miként lehet egy keretet kiterjeszteni, illetve testre szabni. Ennek megfelelően, a tervmintákkal ellentétben, egy keretet a felhasználás során ki is lehet terjeszteni, illetve az alapértelmezett viselkedését módosítani lehet.

A tervminták, keretek, komponensek és kiterjesztési pontok kapcsolatát szemlélteti a következő ábra.



Látható, hogy a keretekben specializált tervminták szerepelnek. Egy keret akár több tervmintát is használhat. Komponens előfordulhat tervmintában és keretben egyaránt. Egy komponens több tervmintának, illetve keretnek is alkotóeleme lehet, és egy tervminta, illetve keret felhasználhat több komponenst is. Egy kerethez több kiterjesztési pont tartozhat.

Egy keret létrehozása során a következő tevékenységeket kell elvégezni:

1. A keret felhasználási területének vagy területeinek azonosítása.
2. A keret elnevezése. A névnek utalnia kell a felhasználás módjára és lehetőleg a keret szerkezetére is.
3. A keret által támogatott alapvető használati esetek meghatározása.
4. A kerettel kapcsolatot tartó aktorok azonosítása.
5. Olyan ismert tervminták vagy egyéb működő megoldások azonosítása, amelyek segíthetik a keret fejlesztését.
6. A keret alapvető felületeinek és komponenseinek megtervezése, és feladatok, illetve aktorok hozzárendelése a hozzáférési felület részeihez.

7. A keret felületeinek alapértelmezett implementációja.

8. A keret kiterjesztési pontjainak leírása és dokumentálása.

9. Tesztesetek és tesztterv létrehozása.

A tervminták és keretek jelentősen felgyorsíthatják a tervezés folyamatát, ugyanakkor használatuk kezdetben jelentős ráfordítást igényel, elsősorban a keretek esetében. Ennek oka a keret szerkezetének megismerésében rejlik. Ennek a kezdeti ráfordításnak az idejét lehet csökkenteni ismert elemek felhasználásával. Ezért fontos, hogy minél több közismert tervmintát, illetve megoldási módot használjunk fel, és az egész keretet megfelelően dokumentáljuk. Ebben az esetben ugyanis a keret nagyobb egységekből épül fel, magasabb absztrakciós szintet használ, így a megfelelő ismeretekkel rendelkező szakember számára könnyebben áttekinthető.

33. Program újratervezési minták (Refactoring Patterns)

1. Létrehozó metódusok (Creational Methods)

   - Probléma: Egy osztály sok, különböző paraméterezésű konstruktorral rendelkezik, így nehéz eldönteni, melyiket kell meghívnunk.
   - Megoldás: Konstruktor helyett példányt létrehozó metódusokat használunk, amelyek egy közös konstruktort használnak.

2. Művelet kihelyezés (Compose Method)

   - Probléma: Egy művelet annyira összetett, hogy a működése nem áttekinthető.
   - Megoldás: A művelet törzsében azonosítjuk az összetartozó részeket, és azokat kiemeljük külön rejtett műveletekbe.

3. Típusfelügyelő osztályok (Replace Type Code with Class)

   - Probléma: Egy attribútum típusa nem teszi lehetővé, hogy megvédjük a változót hibához vezető értékadásoktól, illetve összehasonlításoktól.
   - Megoldás: Az eredeti típust lecseréljük egy olyan osztályra, amely magában foglalja az érték tárolását és a speciális hozzáféréseket, ezáltal ellenőrizhető lesz minden művelet.

4. Null objektum (Null Object)

   - Probléma: A kódban az üres referenciák (null) használata gyakran hibákhoz és kivételhez vezet, továbbá külön kezelést igényel, amihez rendszerint elágazás szükséges.
   - Megoldás: A null értékek helyett egy üres viselkedésű objektumot (NullObject) helyezünk a szerkezetbe – amelyet közvetlenül a konstruktorban is értékül adhatunk az összes referenciának –, így egyszerűsödik a null eseteket is magában foglaló kód.

5. Gyűjtő paraméter (Collection Parameter)

   - Probléma: Egy összetett művelet lokális változón végez műveleteket, ezért a változó vizsgálata (debug) nehézkes.
   - Megoldás: A művelet egyes részeit kiemeljük segédműveletekbe, és paraméterátadáson keresztül gyűjtjük a módosításokat a lokális változóba.

6. Paraméter kiemelés (Extract Parameter)

- Probléma: Egy művelet, vagy konstruktor egy attribútumot értékül ad egy lokálisan inicializált változónak, ezáltal bizonytalan, hogy egy lekérdező kliens objektum megkapja-e az értéket.
- Megoldás: Az attribútumot egy, a kliens által szolgáltatott paraméternek adjuk értékül.

34. OCL használata objektumelvű modellekben

Objektumelvű tervek szabványos leírását teszi lehetővé az UML. Ebben különböző szempontok szerint különböző diagramokkal adhatjuk meg a modellt. Ez egy szintaktikailag egyértelmű leírás, ugyanakkor a szemantikát nem mindig adja meg pontosan, bizonyos aspektusok leírása nem lehetséges. Így például az osztálydiagram megadja a szerkezetet, ugyanakkor bizonyos, a feladat szempontjából fontos jellemzők nem írhatóak le UML segítségével. Gyakran ezeket a tulajdonságokat természetes nyelven adják meg a diagram mellett. Ugyanakkor a gyakorlat bebizonyította, hogy a természetes nyelvű leírás félreértéseket eredményez. Ezt elkerülendő úgynevezett formális nyelveket vezettek be a megadásra. Egy nagy hátránya a tradicionális formális nyelvek használatának, hogy csak mély matematikai ismeretekkel rendelkezők boldogulnak vele, így pont az UML egyik előnye vész el.

A probléma megoldására fejlesztették ki az Object Contstraint Language, OCL, leíró nyelvet, amely formális megadást tesz lehetővé, de emellett könnyen érthető, kezelhető. Az OCL egy tiszta specifikációs nyelv, azaz egy OCL kifejezés kiértékelése egy értékkel tér vissza, mindenféle mellékhatás nélkül. A kiértékelés során a modellben semmilyen változás sem következik be, azaz a rendszer állapota nem változik, noha OCL kifejezésekkel specifikálhatunk állapotváltozást.

Az OCL nem programozási nyelv, így nincs lehetőség vezérlési folyamot megadni OCL-ben. Miután az OCL elsősorban modellezési nyelv, a benne felírt kifejezések definíció szerint direkt módon nem végrehajthatóak. Az OCL kifejezések kiértékelése időpillanathoz kötött, azaz a modell objektumainak állapota nem változhat meg a kiértékelés alatt.

OCL kifejezések alkalmasak következőkre:

- a modellben szereplő osztályok és típusok invariánsainak specifikálása,
- műveletek elő- és utófeltételeinek megadása,
- műveletekre vonatkozó korlátozások meghatározása,
- üzenetek és akciók célpontjainak leírása,
- őrök (guards) leírása,
- sztereotípusok típus invariánsainak specifikálása, illetve
- lekérdező nyelvként is használható.

34.1. Az OCL szintaktikus alapjai

Egy OCL kifejezés mindig egy adott környezetben érvényes, azaz minden OCL kifejezés egy adott típus egy példányának környezetében értelmezett. A self kulcsszó minden esetben a környezeti példányra hivatkozik.

A kifejezés UML modellbeli környezetét úgynevezett környezet (context) deklaráció adja meg az OCL kifejezés elején. Ha a kifejezést diagramban adjuk meg és ellátjuk megfelelő sztereotípussal, továbbá szaggatott vonallal összekötjük a környezeti elemével, akkor további környezet megadás nem szükséges. Ha a kifejezést külön adjuk meg, akkor a **context** kulcsszó után következik a környezet megnevezése, amit követ a kifejezés.

Legegyszerűbb esetben háromféle kifejezést adhatunk meg:

- invariánsok segítségével adhatunk meg típusinvariánsokat, amelyek a típus (osztály) összes példányára állandóan érvényes tulajdonságokat adnak meg;
- előfeltételekkel írhatjuk le egy művelet vagy egyéb viselkedési elem végrehajtásához szükséges feltételeket;
- utófeltételekkel definiálhatjuk egy művelet vagy egyéb viselkedési elem végrehajtása után kielégítendő feltételeket.

Invariánsokat a «invariant» sztereotípussal, illetve a környezetet követő **inv** kulcsszóval adhatunk meg. Előfeltételek esetén a «precondition» sztereotípust, illetve **pre** kulcsszót, utófeltételek esetén pedig a «postcondtion» sztereotípust, illetve **post** kulcsszót használhatjuk.

Tekintsük a következő egyszerű példát! Tegyük fel, hogy adott a **Cég** osztály, amelynek attribútuma az alkalmazottak száma (alkalmazottszám), és rendelkezik egy ennek értékét lekérdező függvénnyel (alkalmazottSzám()). A result kulcsszó szolgál egy függvény visszatérési értékének megadására. A modellünk legyen olyan, hogy minden cégnek legalább 10 alkalmazottal kell rendelkeznie. Ezeket a következő módon fejezhetjük ki.

context Cég **inv**:
self.alkalmazottszám >= 10

context Cég::alkalmazottSzám() : Integer
post: result = self.alkalmazottszám

Az előző invariáns leírása megadható úgy is, hogy a self kulcsszót elhagyjuk, ugyanis a környezet egyértelmű. Egy másik lehetőség a self helyett egy név bevezetése.

context c : Cég **inv**:
c.alkalmazottszám >= 10

Ha szükséges a későbbi használathoz, hivatkozáshoz az invariánst elnevezhetjük az **inv** kulcsszó után elhelyezett névvel.

context c : Cég **inv** ElégAlkalmazott:
c.alkalmazottszám >= 10

Hasonló módon rendelhetünk nevet elő-, illetve utófeltételekhez is.

Ha egy kifejezésben (rendszerint utófeltételekben) szükségünk van egy attribútum, tulajdonság végrehajtást megelőző értékére, azt a név után írt @pre kulcsszóval tehetjük meg. Például, ha a cég felvesz új alkalmazottakat, akkor ezt leírhatjuk a következőképpen.

context Cég::felvesz(a : Integer)
pre: a > 0
post: alkalmazottszám = alkalmazottszám@pre + a

Ha egy összetett részkifejezést többször is használunk, akkor lehetőségünk van a részkifejezés értékét egy változóban tárolni, és a változót használni, a kifejezés többszöri leírása helyett. Erre szolgál a **let** kulcsszó. Feltételeket használhatunk a jól ismert **if** ... **then**

... **else** ... **endif** szerkezet segítségével. Az **implies** kulcsszó használható a következmény kifejezésére.

context ... **inv**:
  let *változó* : *Típus* = *kifejezés*
  if *felt* **then**
    ... *változó* ...
  else
    ... *változó* ...
  endif

A kifejezéseken belül egy objektum kapcsolataira az asszociációban szereplő név segítségével hivatkozhatunk. (Ha nincs szerepnév, a kapcsolt osztály kisbetűs nevét használhatjuk.) Ha legfeljebb egy objektum lehet kapcsolatban (0..1 vagy 1 multiplicitás), akkor az eredmény egy objektum, ellenkező esetben objektumok halmaza. Az objektumra attribútumai, műveletei alkalmazhatóak, a hivatkozni pont után lehet ezekre. Halmaz esetén -> jel után alkalmazhatóak a halmazokra értelmezett műveletek:

size() a halmaz elemeinek száma,

isEmpty() üres-e a halmaz,

notEmpty() nem üres a halmaz,

select(*felt*) a halmazból kiválasztja a *felt* logikai feltételt kielégítő elemeket egy halmazba,

reject(*felt*) a halmazból elhagyja a *felt* logikai feltételt kielégítő elemeket,

forAll(*felt*) a halmaz összes elemére teljesülni kell a *felt* logikai feltételnek,

exists(*felt*) a halmaz legalább egy elemére teljesülni kell a *felt* logikai feltételnek,

collect(*felt*) kigyűjti azokat az elemeket, amelyek kielégítik a *felt* logikai feltételt, de az eredmény elemek típusa nem feltétlen egyezik meg a kiindulási halmazzal. (Nem csak halmaz esetén alkalmazható.)

Ha szükséges az egyetlen objektum is kezelhető halmazként, és a -> jellel hivatkozhatunk a halmaz lehetséges műveleteire. 0..1 multiplicitás esetén például a notEmpty() művelettel ellenőrizhetjük, hogy van-e kapcsolt objektum.

Az előzőeken kívül léteznek az OCL-ben előre definiált tulajdonságok, amelyek minden objektumra használhatóak.

oclIsTypeOf(t) igaz, ha az objektum t típusú (az osztály példánya).

oclIsKindOf(t) igaz, ha az objektum t vagy abból származtatott típusú (az adott osztály, vagy abból származtatott osztályok példánya).

oclInState(s) igaz, ha az objektum s állapotban van. Az állapot bármelyik állapot lehet, amely az objektum osztályának állapotdiagramjában szerepel. Összetett állapotok beágyazott részállapotaira a :: jelöléssel hivatkozhatunk.

oclIsNew() egy művelet utófeltételében igaz, ha az objektumot a művelettel hoztuk létre.

34.2. Példák

A következőkben néhány egyszerű példány szemléltetjük az OCL használatának lehetőségeit egy modell jelentésének pontosítására.

34.2.1. Házasság

Tegyük fel, hogy személyek közötti házassági kapcsolatot szeretnénk modellezni. Ekkor a személyek közötti reflexív kapcsolat a házasság. Egy személynek egyszerre legfeljebb egy házastársa lehet. Minden személynek ismerjük a nemét és a korát, és két művelet hajtható végre: a születésnap, ami a kort növeli eggyel, illetve az esküvő, amely az adott személlyel házastársi kapcsolatot hoz létre. Egy lehetséges osztálydiagramot mutat a 34.1. ábra.

Az osztálydiagram megadja a szükséges ismereteket, azonban nem tartalmaz olyan információkat, amelyek a házassággal kapcsolatban teljesülnek, mint például:

- senki sem lehet önmaga házastársa,
- a házastársak különböző neműek,
- egy személy házastársának házastársa a személy,
- csak 18 évet betöltött személy lehet házas.

A felsorolt feltételeket formálisan is megadhatjuk az osztálydiagram kiegészítéseként az OCL segítségével.

Senki sem lehet önmaga házastársa:

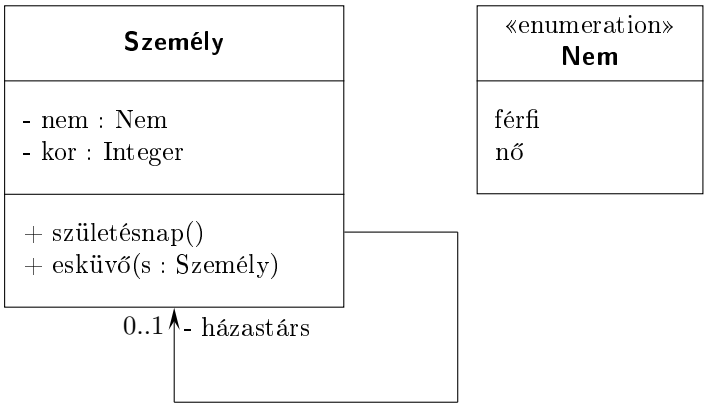
context Személy **inv**:
házastárs->notEmpty() **implies** házastárs <> self

A házastársak különböző neműek:

context Személy **inv**:
házastárs->notEmpty() **implies** házastárs.nem <> nem

Természetesen a különböző nem garantálja, hogy senki sem lehet önmagának házastársa, így ennek használata tulajdonképpen redundánssá teszi az előző feltételt.

Egy személy házastársának házastársa a kiinduló személy:



34.1. ábra. A házasság osztálydiagramja

context Személy **inv**:
házastárs->notEmpty() **implies**
házastárs.házastárs.notEmpty() **and** házastárs.házastárs = self

Csak 18 évet betöltött személy lehet házas:

context Személy **inv**:
házastárs->notEmpty() **implies** kor >= 18

A felírt invariánsokat összevonhatjuk egyetlen invariánssá:

context Személy **inv**:
házastárs->notEmpty() **implies**
házastárs <> self **and**
házastárs.nem <> nem **and**
házastárs.házastárs.notEmpty() **and**
házastárs.házastárs = self **and**
kor >= 18

Ha külön ki szeretnénk fejezni, hogy házastárs nélküli személy nem lehet senkinek sem a házastársa, akkor ezt is megtehetjük. (Ez az előzőekből következik, nevezetesen az, hogy a házastárs házastársa a kiinduló személy, garantálja ezt a feltételt.)

context Személy **inv**:
házastárs->isEmpty() **implies**
Személy.allInstances()->forAll(s : Személy |
s.házastárs.isEmpty() **or** s.házastárs <> self)

A személyekre vonatkozó műveleteket is pontosíthatjuk, azaz születésnap esetén a kor eggyel nő, esküvő pedig csak akkor lehetséges, ha még egyedülállóak, és ekkor a házastárs a másik személy lesz.

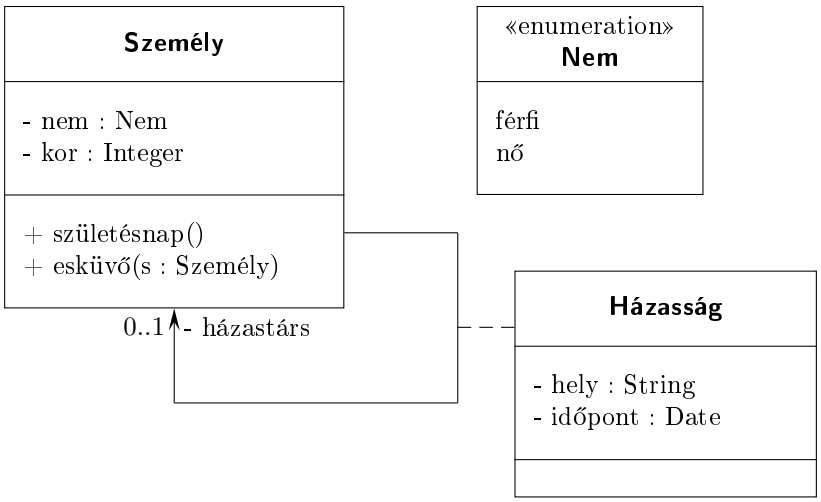
context Személy::születésnap()
post: kor = kor@pre + 1

context Személy::esküvő(s : Személy)
pre: házastárs.isEmpty() **and** s.házastárs.isEmpty()
post: házastárs = s **and** s.házastárs = self

Felírt megszorítás miatt az esküvő csak az egyik személy esetén hajtható végre, a másikra már nem. Ha `s1` és `s2` összeházasodik csak `s1.esküvő(s2)` vagy csak `s2.esküvő(s1)` használható. Mindkét művelet végrehajtása esetén a második előfeltétele nem teljesülne. Ugyanakkor az előfeltétel enyhítésével (csak self-re teszünk megkötést) a személyekre vonatkozó invariáns nem állna fenn a másik művelet befejeződéséig.

Módosítsuk a feladatot úgy, hogy a házasságokról nyilván kell tartanunk azok helyszínét és időpontját! Egy kézenfekvő megoldásnak tűnik, hogy a modellünkben a házasságot kifejező relációhoz társult osztályként vegyük fel a házasság osztályt, amelynek attribútumai a megadott értékek (34.2. ábra).

Ebben a megoldásban az előzőleg alkalmazott megszorítások ugyanúgy szükségesek. Ezeket ki kell egészíteni azonban egy újabbal, miszerint a házastársakhoz ugyanaz a házasság objektum tartozik. (Ez az olvasó feladata.) Értelemszerűen a személyek esetén pontosítani lehetne a házastársi kapcsolat előfeltételében szereplő 18 éves korhatárt úgy, hogy a házasság megkötésének időpontjában kellene legalább 18 évesnek lennie a személynek. Ehhez azonban ismerni kellene a kor helyett, vagy mellett a születési időpontot.



34.2. ábra. A házasság módosított osztálydiagramja

Egy másik lehetséges megközelítés, hogy nem társult osztályként tekintünk a házasságra, hanem olyan osztályként, amelynek objektumai pontosan két személyt személyt kapcsolnak össze. Ekkor két megoldás közül választhatunk:

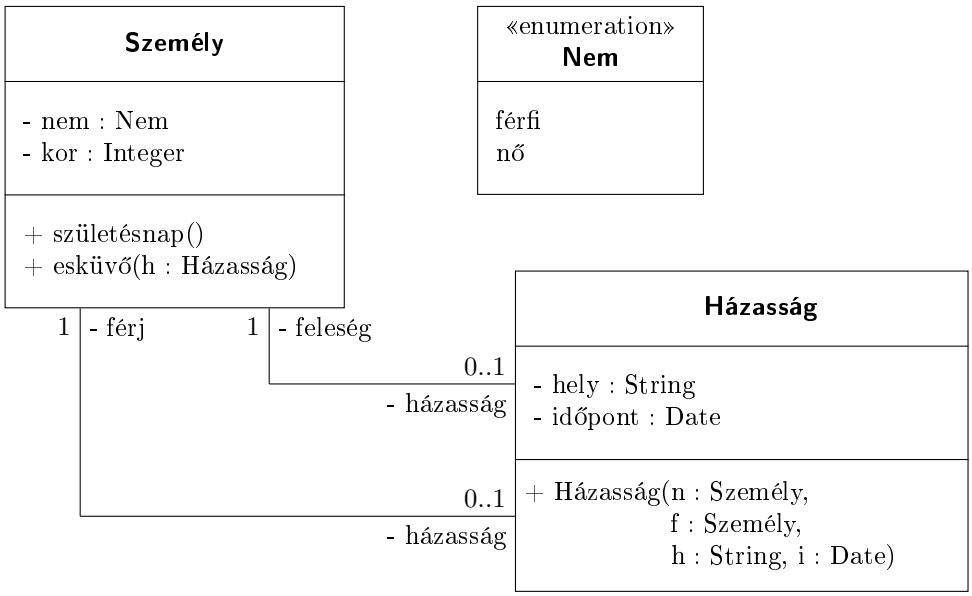
1. Egy házasság objektum kétféleképpen kapcsolódik személyekhez, mindkét kapcsolatban pontosan egy személy vesz részt. Az egyik kapcsolatban a személy a feleség, a másikban a férj szerepét tölti be. (Ez lényegében egyenértékű azzal, hogy az előző megközelítésben a társult osztályhoz tartozó kapcsolat irányítását megszüntetjük – két irányban navigálható –, és az egyik végén a feleség, a másik végén a férj szerepeket használjuk, mindkét oldalon 0..1 multiplicitással.)

2. Egy házasság objektum egyféleképpen kapcsolódik személyekhez, a kapcsolatban pontosan két személy vesz részt.

Mindkét megoldás biztosítja, hogy egy személy házastársának házastársa a kiinduló személy, ezt nem kell külön invariánsban kikötnünk. A második megoldás például garantálja, hogy senki sem lehet önmaga házastársa. Ugyanakkor mindkét esetben ki kell fejeznünk, hogy a házasságban részt vevő személyek neme különböző, amit az első esetben egyszerűen megtehetünk, ha kikötjük, hogy a feleség nő, a férj pedig férfi. A második esetben ez a feltétel kicsit nehézkesebben fejezhető ki. Ugyanakkor, ahogy azt már az előzőekben láttuk, a nemek különbözősége biztosítja azt is, hogy senki sem lehet önmaga házastársa, ezért azt nem kell külön feltüntetnünk. Miután a többi szempont alapján a két megoldási mód között lényeges eltérés nincs, az elsőt választjuk. Ennek osztálydiagramja látható a 34.3. ábrán.

Vegyük észre, hogy az esküvő ebben a modellben módosult. Egy házasság objektum létrejötte eredményezi a kapcsolat létrejöttét, amelynek során a személyeket a házassághoz kell kapcsolnunk, és így az előző modellben aszimmetrikus esküvő művelet most szimmetrikussá válik.

A személyekre most már csak egyetlen invariáns vonatkozik: legalább 18 évesek, ha házasok. Ezen kívül a két művelet utófeltételét adhatjuk még meg.



34.3. ábra. A házasság módosított osztálydiagramja

context Személy **inv**:
házasság->notEmpty() **implies** kor >= 18

context Személy::születésnap()
post: kor = kor@pre + 1

context Személy::esküvő(h : Házasság)
post: házasság = h

A többi feltétel a házasság objektumokhoz tartozik. A konstruktor előfeltételében vizsgáljuk azokat a feltételeket, amelyek az invariánsokban szerepelnek. Ezeket tulajdonképpen el is hagyhatnánk, azonban a modell implementációját nagyban egyszerűsíti a feltételek feltüntetése.

context Házasság **inv**:
feleség.nem = nő **and** férj.nem = férfi

context Házasság::Házasság(n:Személy, f:Személy, h:String, i:Date)
pre: n.nem = nő **and** f.nem = férfi **and**
n.házasság.isEmpty() **and** f.házasság.isEmpty() **and**
n.kor >= 18 **and** f.kor >= 18
post: self.oclIsNew() **and** feleség = n **and** férj = f **and**
hely = h **and** időpont = i **and**
feleség.házasság = self **and** férj.házasság = self

Látható, hogy a modell megváltoztatása lényegesen befolyásolja az alkalmazott OCL kifejezéseket és azok mennyiségét is. Kézenfekvőnek tűnik azt a modellt előnyben részesíteni, amelyben kevesebb OCL feltételt kell használnunk, hiszen ekkor maga a modell pontosabban írja le a problémát.

34.2.2. Járművek

Vizsgáljuk személyek és közlekedési eszközök (járművek) kapcsolatát! Az egyszerűség érdekében kétféle járművet különböztetünk meg: autót és kerékpárt, amelyeknek ismerjük a színét. Csak azzal kívánunk foglalkozni a modellben, hogy személyek járműveket birtokolhatnak. Személyeknek ismerjük a nevét és a korát. Ha feltesszük, hogy egy járművet csak egy személy birtokolhat, és egy személy tetszőleges számú jármű tulajdonosa lehet, akkor a 34.4. ábrán látható osztálydiagramhoz jutunk.

Bizonyos esetekben ez a modell nem eléggé pontos. Ha például ki szeretnénk fejezni, hogy egy jármű tulajdonosa legalább 18 éves, akkor azt a következő OCL kifejezéssel tehetjük meg:

context Jármű **inv**:
tulajdonos.kor >= 18

Lehet, hogy ez túl erős megszorítás, hiszen gyerekek is birtokolhatnak kerékpárokat. Az autó osztály a jármű osztályból származik, így annak kapcsolatait is átveszi, ezért a fenti invariáns megfogalmazható csak az autókra, megfelelően gyengítve a feltételt.

context Autó **inv**:
tulajdonos.kor >= 18

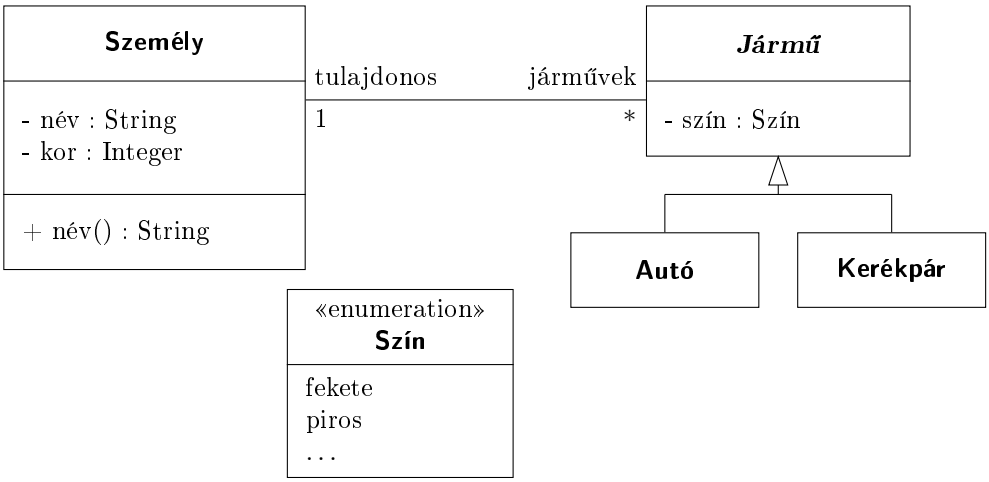
Ugyanezt a tulajdonságot megfogalmazhatjuk a személy osztály invariánsával is, noha ez bonyolultabb.

context Személy **inv**:
kor < 18 **implies** járművek->forAll(j | not j.oclIsTypeOf(Autó))

Ha azt szeretnénk biztosítani, hogy senki se rendelkezhessen háromnál több járművel, akkor ezt is megtehetjük a következő OCL kifejezéssel. (Értelemszerűen a 0..3 multiplicitás használata a $*$ helyett ugyanezt eredményezi.)

context Személy **inv**:
járművek.size() <= 3

Azt is kifejezhetjük, hogy a személy osztály név művelete a név attribútum értékét adja vissza:



34.4. ábra. A járművek és személyek kapcsolata

context Személy::név()
post: result = név

Ugyanez másképpen:

context Személy **inv**:
név() = név

A következőkben néhány „mesterséges” követelményt fogalmazunk meg OCL segítségével.

Személyeknek csak fekete járműveik lehetnek:

context Személy **inv**:
járművek.forAll(j | j.szín=fekete)

Senkinek sem lehet háromnál több fekete járműve:

context Személy **inv**:
járművek.select(j | j.szín=fekete)->size() <= 3

Létezik piros autó:

context Autó **inv**:
Autó.allInstances->exists(a | a.szín=piros)

34.2.3. Sportklub

Egy sportklub személyeket szerződtethet. A klubnak ismerjük a nevét, a személyeknek a nevét és a születési idejét. A személyek lehetnek játékosok vagy edzők. Egy klub legfeljebb egy edzőt alkalmazhat, tetszőleges számú játékosa lehet. A játékosok, edzők és a klub szerződést kötnek, aminek ismert a lejárati dátuma és a fizetés. Egy személy legfeljebb egy klubbal állhat szerződéses viszonyban. Az edző 11 fős csapatot állít össze egy mérkőzésre. A csapatba csak egészséges játékos lehet beállítani. A játékosokról tudjuk játszanak-e a csapatban és, hogy sérültek-e.

A 34.5. ábra egy lehetséges osztálydiagramot mutat. A dátum típust a dátumok kezelése miatt vezettük be. Most egyetlen műveletet használunk dátumokra, hogy egy adott érték megelőzi-e a mai napot (elmúlt). A diagramban szereplő további elemek adódnak a leírásból.

Nyilvánvaló, hogy az osztálydiagram nem pontos abban az értelemben, hogy megenged olyan eseteket is, amelyek nem fordulhatnak elő. A továbbiakban ezeket az eseteket zárjuk ki OCL kifejezésekkel.

Értelemszerű, hogy csak olyan személyek létezhetnek, akik már megszülettek, azaz a születési időpont minden személy esetében megelőzi az aktuális dátumot.

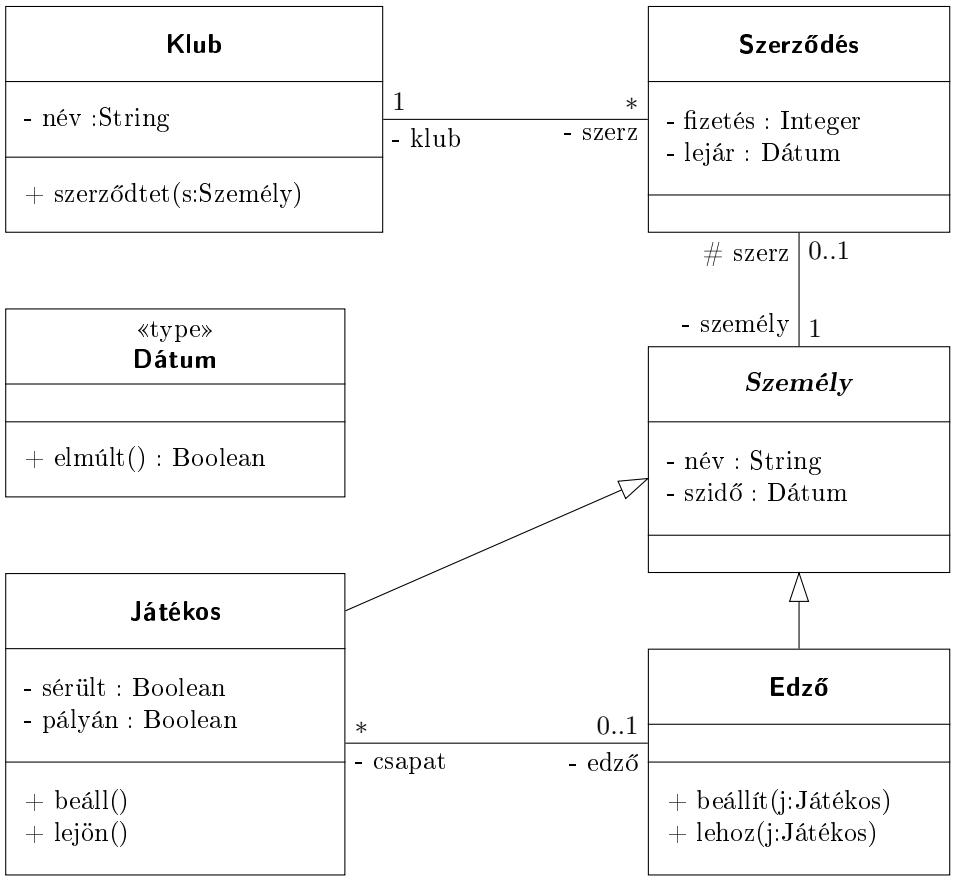
context Személy **inv**:
szidő.elmúlt()

Minden szerződés esetén a fizetésnek pozitívnak kell lennie, és a lejárat dátuma még nem múlhatott el. (Ha elmúlt volna, akkor a szerződés megszűnne.)

context Szerződés **inv**:
fizetés > 0 **and not** lejár.elmúlt()

Ha egy edzőnek nincs szerződése, akkor csapata sem lehet.

context Edző **inv**:
szerz->isEmpty() **implies** csapat->isEmpty()



34.5. ábra. A sportklubok osztálydiagramja

Egy edző által irányított csapat összes játékosa ugyanahhoz a klubhoz tartozik, mint az edző.

context Edző **inv**:
csapat->forAll(j : Játékos | j.szerz->notEmpty() **and**
j.szerz.klub = szerz.klub)

Egy szerződés nélküli játékosnak nincs edzője, illetve ha van szerződése, akkor ugyanahhoz a csapathoz tartozik, mint az edzője.

context Játékos **inv**:
(szerz->isEmpty() **implies** edző->isEmpty()) **and**
(szerz->notEmpty() **imlies** edző->notEmpty() **and**
edző.szerz->notEmpty() **and** edző.szerz.klub = szerz.klub)

Ha egy klub szerződtetett személyeket, akkor azok között pontosan egy edzőnek kell lennie.

context Klub **inv**:
szerz->notEmpty() **implies**
szerz->collect(s:Szerződés | s.személy.oclIsTypeOf(Edző)).size()=1

A játékos osztály sérült és pályán attribútumai a játékosok állapotainak leírására szolgálnak, azaz azt fejezik ki, hogy a játékos egészséges-e, illetve a kiválasztott csapatban a pályán van-e. Ennek megfelelően a továbbiakban az értékekhez tartozó állapotokat (egészséges és pályán) használjuk az OCL kifejezésekben.

Egy edző a pályára küldhet egy játékost, ha az a csapatában van, nincs még a pályán és egészséges, valamint a pályán nem lesz túl sok játékos.

context Edző::beállít(j:Játékos)
pre: csapat->exists(c : Játékos | c = j) **and**
j.oclInState(egészséges) **and not** j.oclInState(pályán) **and**
csapat->select(c : Játékos | c.oclInState(pályán)).size() < 11
post: j.beáll()

Egy edző lehozhat egy játékost a saját csapatából, ha az a pályán van.

context Edző::lehoz(j:Játékos)
pre: csapat->exists(c : Játékos | c = j) **and**
j.oclInState(pályán)
post: j.lejön()

Egy játékos beállhat, ha egészséges és nincs a pályán, illetve lejöhet, ha a pályán van.

context Játékos::beáll()
pre: oclInState(egészséges) **and not** oclInState(pályán)
post: oclInState(pályán)

context Játékos::lejön()
pre: oclInState(pályán)
post: **not** oclInState(pályán)

A klub szerződtethet egy személyt, ha annak nincs szerződése, illetve edző esetén, ha még nincs edző a klubban. (Most eltekintünk a szerződés konkrét adataitól.)

context Klub::szerződtet(s:Személy)
pre: s.szerz->isEmpty() **and**
s.oclIsTypeOf(Edző) **implies**
szerz->select(személy.oclIsTypeOf(Edző))->isEmpty()
post: sz.oclIsNew() **and** sz.oclIsTypeOf(Szerződés) **and**
sz.klub = self **and** sz.személy = s

Az utófeltétel garantálja

s.szerz.klub = self **and** szerz->exists(személy = s)

feltétel teljesülését.

Ebben a példában több művelet elő- és utófeltételét adtuk meg OCL segítségével. Ugyanakkor hasonló eredményre jutnánk, ha elkészítenénk a dinamikus modellből az állapotdiagramot, és abban az egyes állapotátmenetek feltételeit írnánk fel. (A műveletek állapotváltozást okoznak, így tulajdonképpen mindegy, hogy bizonyos feltételek hol jelennek meg.)

A példák mutatják, hogy az OCL alkalmas az objektumelvű modell pontosítására, de azt is érdemes szem előtt tartani, hogy a modell módosításával, vagy az osztálydiagramon kívül egyéb diagramok használatával a szükséges OCL kifejezések mennyisége csökkenthető.