Programozási Technológia II.

Tartalomjegyzék

1. 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**;

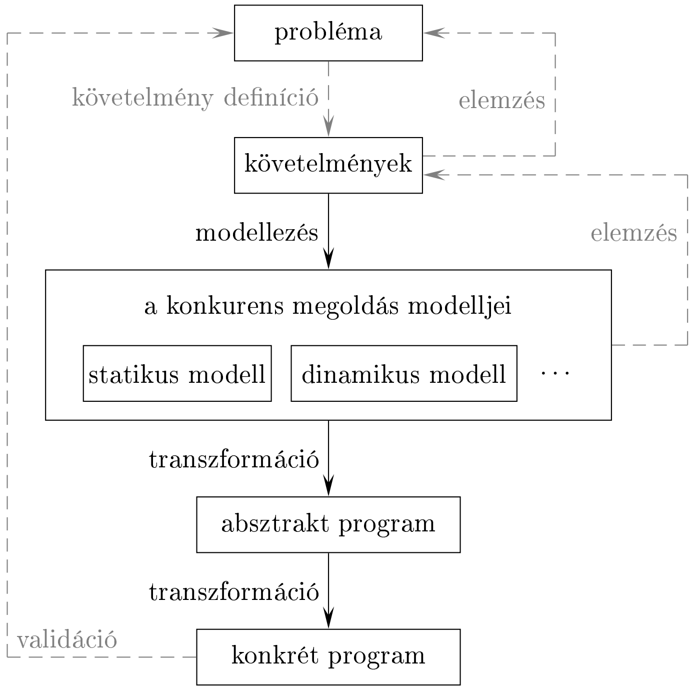
ahol <*feltétel*> egy logikai kifejezés, és <*utasítássorozat*> nem tartalmazhat iterációt vagy szinkronizációs (várakozó) utasításokat.

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 (1.1. 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.



1.1. ábra. Konkurens programok UML alapú előállítása

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

1.1. 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!

1.2. Első esettanulmány

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

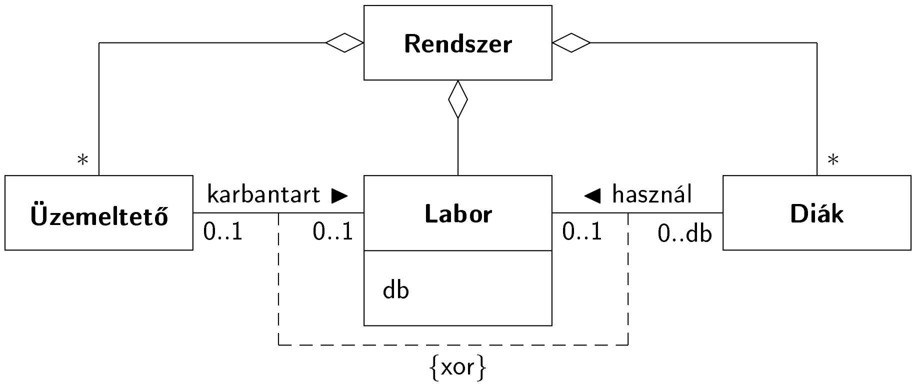
A feladatban egy számítógépes laboratórium használatát kell programmal szimulálnunk. Egyetlen laborral foglalkozunk, amelyben adott számú számítógép található. A labort hallgatók akarják használni. Egy hallgató végzi a tanulmányait, és amikor számítógépre van szüksége, akkor a laborhoz megy, és kint 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.

1.2.1. 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.



1.2. 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.

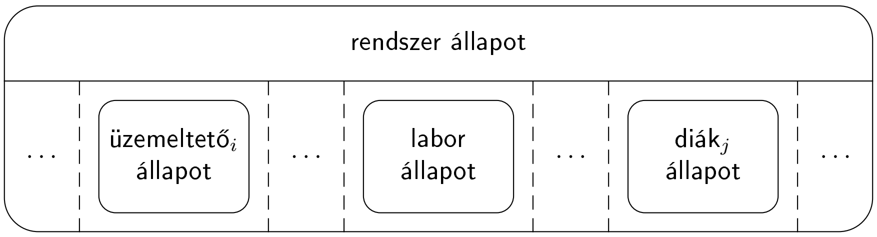
A használ és a karbantart relációk egymást kölcsönösen kizárják, amit megszorítással fejezhetünk ki. Így az 1.2. ábrán látható osztálydiagramhoz jutunk.

1.2.2. Dinamikus modell

A rendszer állapotait az üzemeltetők, a diákok és a labor állapotai együtt határozzák meg (1.3. 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.



1.3. 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.

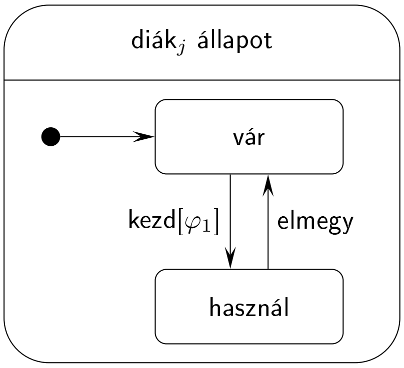
- Senki sem igényel karbantartást, és senki sem tartja karban a labort. Az állapot neve: OK, invariánsa: $\mathsf{r} = 0 \wedge \mathsf{w} = 0$.
- Karbantartást igényeltek, és senki sem tartja karban a labort. Az állapot neve: igényelt, invariánsa: $\mathsf{r} > 0 \wedge \mathsf{w} = 0$.
- A labort karbantartják.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

Az állapotátmenetek feltételei a következők.

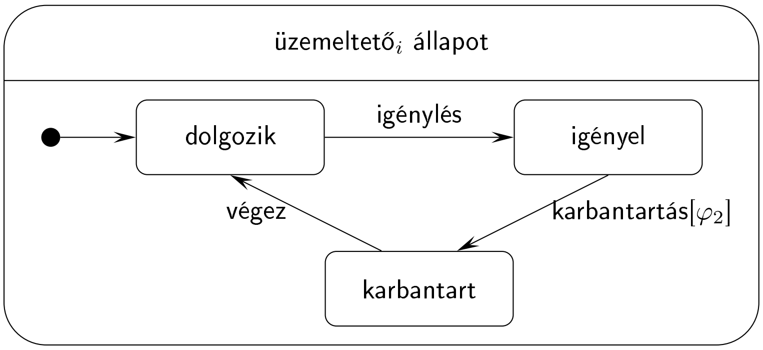
- Egy diák beléphet a laborba, és elkezdhet egy gépet használni (kezd esemény), ha a labor nincs tele, és az állapota OK, azaz $\varphi_1 : \neg\, in\ \mathsf{tele} \wedge in\ \mathsf{OK}$ feltétel fennáll. Másképpen: $\varphi_1 : \mathsf{t} < \mathsf{db} \wedge \mathsf{r} = 0 \wedge \mathsf{w} = 0$.
- 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.

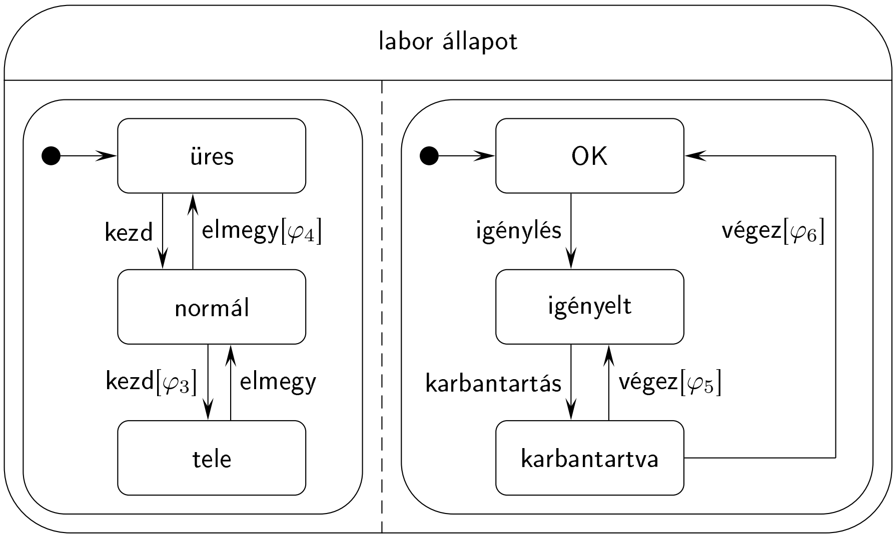
Az állapotdiagramok az 1.4–1.6. ábrákon láthatóak.



1.4. ábra. Egy diák állapotdiagramja



1.5. ábra. Egy üzemeltető állapotdiagramja



1.6. ábra. A labor állapotdiagramja

1.2.3. 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 az 1.7. ábrán, a folyamatok vázai pedig az 1.8. ábrán láthatóak.

Az atomi utasításokat, az állapotdiagram megfelelő akcióit és az átmeneti feltételeket (azaz őrfeltételeket) mutatja az 1.9. 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 az 1.10. ábrán látható eredményre jutunk. Az absztrakt programot megkapjuk, ha ezeket a folyamatokat helyettesítjük az 1.7. 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
```

1.7. ábra. A program váza

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

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

1.8. 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

1.9. 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
```

1.10. ábra. Az absztrakt folyamatok

1.3. 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.

1.3.1. 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.

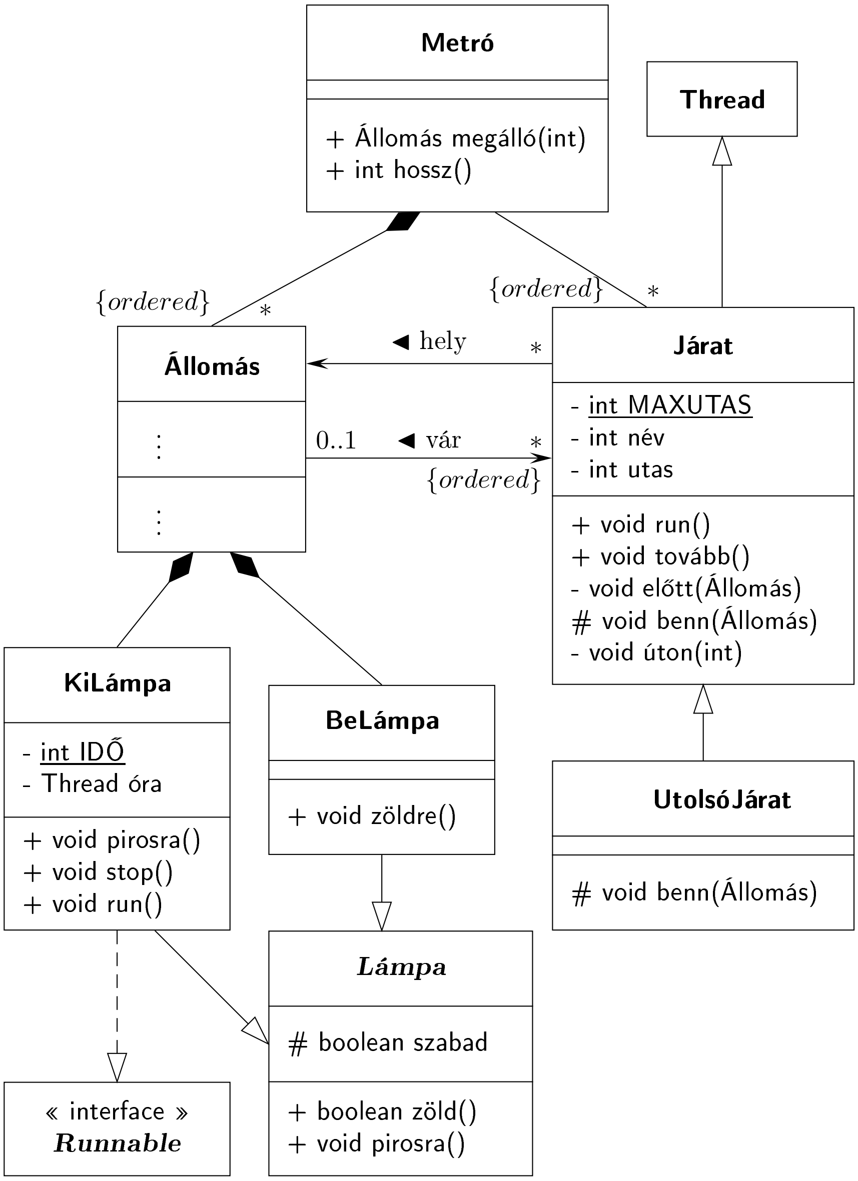
Minden állomást jellemez a neve, az ott várakozó utasok száma, a bizonyos időközönként érkező új utasok száma, a beérkező járatról leszálló utasok aránya, a következő állomás távolsága, illetve, hogy nyitva van-e. Egy állomás műveletei: a megfelelő adattagok értékeit megadó függvények, az állomás bezárása (*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 az 1.11. 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.

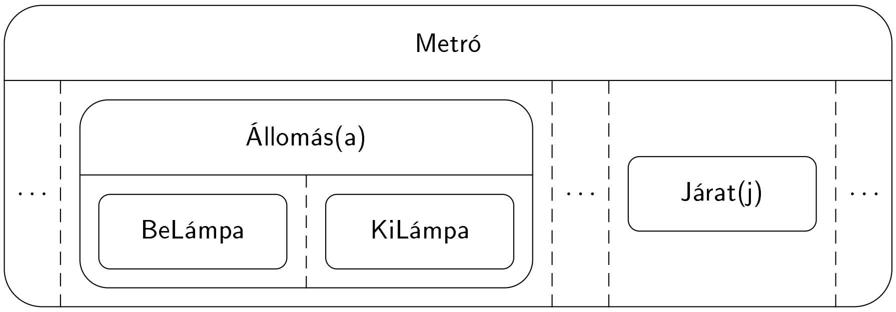


1.11. ábra. A metróvonal osztálydiagramja

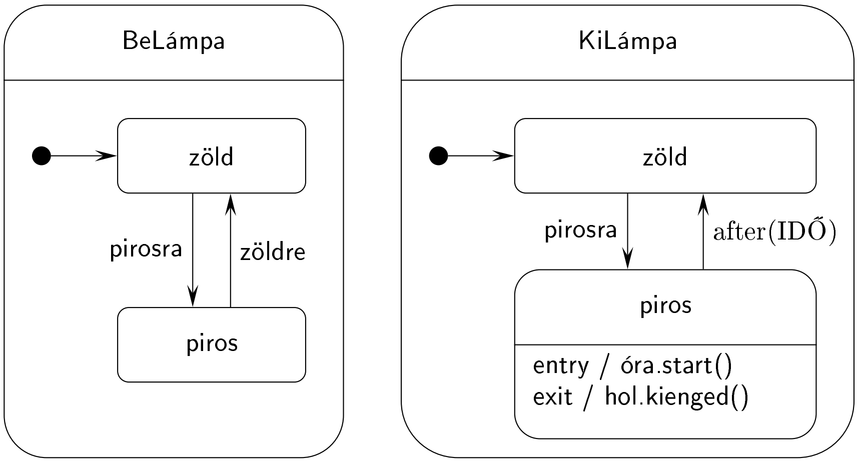
1.3.2. Dinamikus modell

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 (1.12. 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



1.12. ábra. A metróvonal állapotdiagramja



1.13. ábra. A lámpák állapotdiagramja

lámpák esetén ezt kifejezhetjük az állapotinvariánssal:

$$I(\mathsf{piros}) = \quad t < \mathsf{ID\tilde{O}},$$

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 (1.13. 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:

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

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:

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

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.

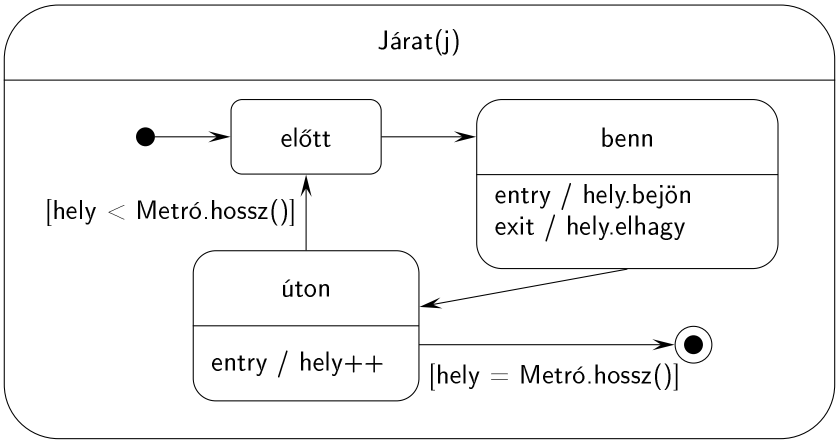
Ha egy járat bemegy egy állomásra, akkor erről értesíteni kell a bevezető lámpát. Ezt az állomáson keresztül tehetjük meg, erre szolgál a *bejön* művelet. Másképpen fogalmazva a bevezető lámpa pirosra eseménye akkor következik be, amikor egy járat belép az állomásra:

BeLámpa.pirosra $\equiv$ hely.bejön $\equiv$ benn.entry.

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

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

Egy út megkezdésekor a járat a következő állomással kerül kapcsolatba, azaz az úton állapot belépési akciója a *hely* növelése. Az eddigiek alapján az 1.14. ábrán látható állapotdiagramhoz jutunk.



1.14. ábra. Egy járat állapotdiagramja

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

1.15. 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;
```

1.16. ábra. Egy járat folyamat váza

1.3.3. 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 (1.15. 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 (1.16. ábra).

1.3.4. 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 (1.17. 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

```
o1:
      ...
      await o2.feltétel() then S ta;
      ...
```

1.17. 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(); }
```

1.18. 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;
}
```

1.19. 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 az 1.18. 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 az 1.19. ábrán.

A ***Lámpa*** és a **BeLámpa** osztály megvalósítása (1.20. 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-

```java
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();
    }
}
```

1.20. 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 (1.20. 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.

```java
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; }
```

1.21. 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;
    }
}
```

1.22. 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 az 1.21–1.22. 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

```java
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) {}
        }
    }
```

1.23. 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(); }
}
```

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

esetben az állomás a *tovább* művelettel tudja aktivizálni a járatot. A megvalósítást mutatják az 1.23–1.24. ábrák.

Az **UtolsóJárat** osztály implementációja értelemszerű, a **Járat** *benn* műveletét kell kiegészítenünk az állomás bezárásával (1.25. ábra).

A **Metró** osztályban az **Állomás** objektumokat definiálunk egy konstans tömbben, ezzel implementáljuk a két osztály közötti kompozíciós kapcsolatot. A **Metró** osztály megvalósításából (1.26. ábra) a *main* műveletet nem tartalmazza a modell. Ebben adott számú

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

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

1.25. 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), 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);
            }
        }
    }
}
```

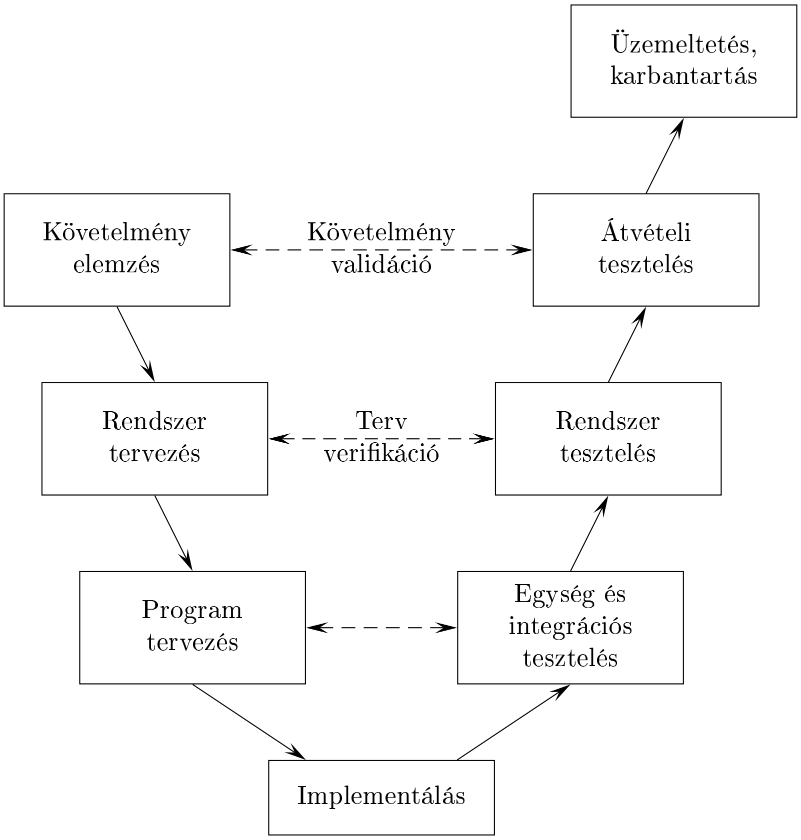
1.26. ábra. A **Metró** osztály megvalósítása

2. Szoftverfejlesztési modellek

Az előző félévben megismertünk három hagyományos szoftverfejlesztési modellt: a vízesés, az evolúciós és a Boehm-féle spirális modellt. Ebben a fejezetben további, újabb modelleket mutatunk be.

2.1. 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 (2.1. ábra).



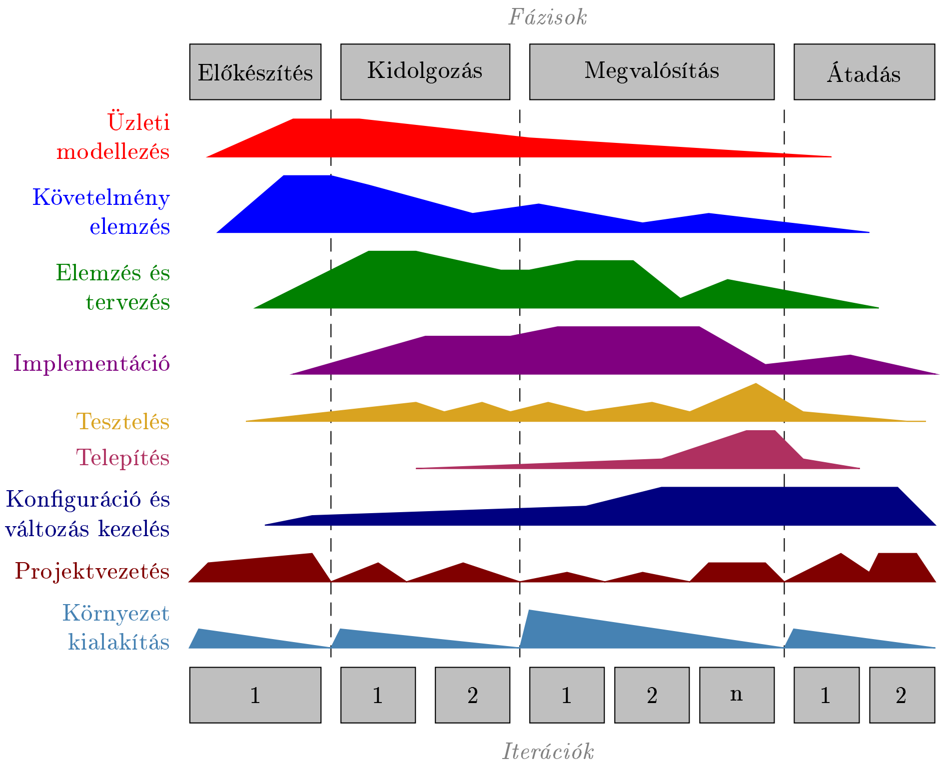
2.1. ábra. A V-modell

2.2. 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. A 2.2. ábra egy átlagos projekt összes elvégzendő munkájának az egyes fázisokra és 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.



2.2. ábra. RUP munkafolyamatok és fázisok

2.2.1. 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.

2.2.2. 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.*

2.2.3. 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?

2.2.4. 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.

2.3. 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.

2.3.1. 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.)

2.3.2. 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.

2.3.3. 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.

2.4. 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 koncentrálhanak, 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.

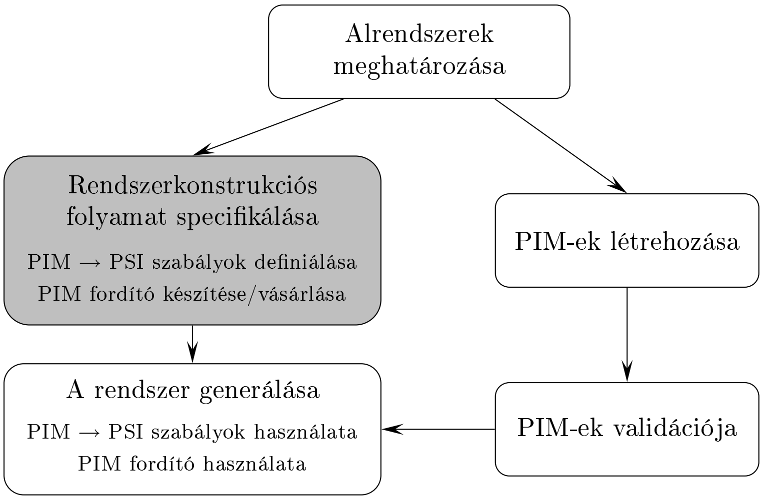
A módszer alkalmazásakor a következő problémák merülhetnek fel.

- 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.



2.3. ábra. Szofverfejlesztés menete xUML-ben

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 (2.3. ábra).

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.

2.4.1. 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

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 (2.4. ábra).

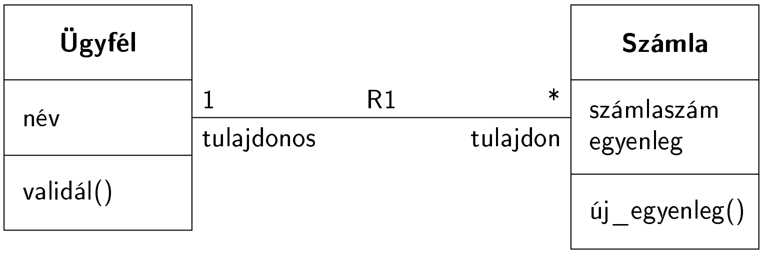
link számla R1 ügyfél

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



2.4. ábra. Az ügyfelek és a számlák közötti kapcsolat

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

3. 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.

3.1. 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.

3.1.1. 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.

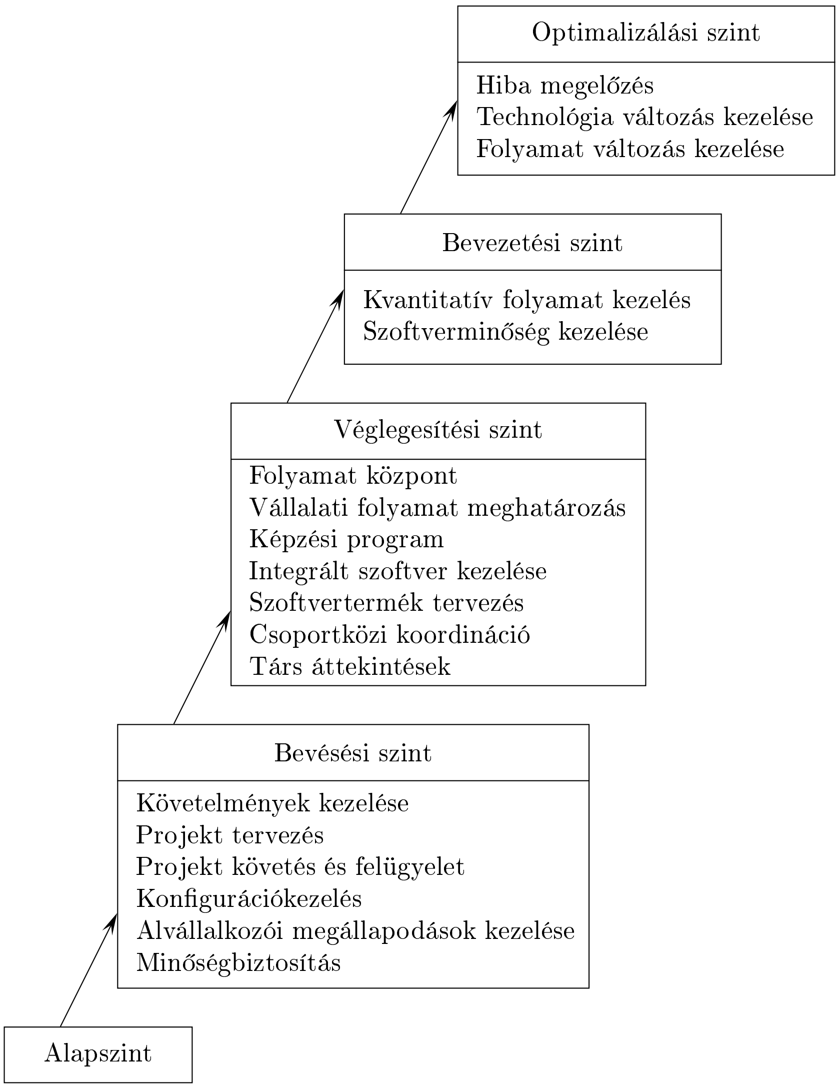
3.2. CMM

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 (3.1. 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.



3.1. 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.)

4. 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.

4.1. 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.

4.2. 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.

4.3. 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.

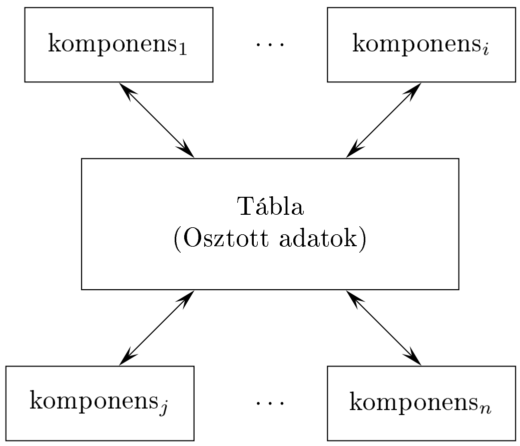
4.4. Réteg szerkezetű rendszer

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.

4.5. Gyűjtemény

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



4.1. 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 (4.1. 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.

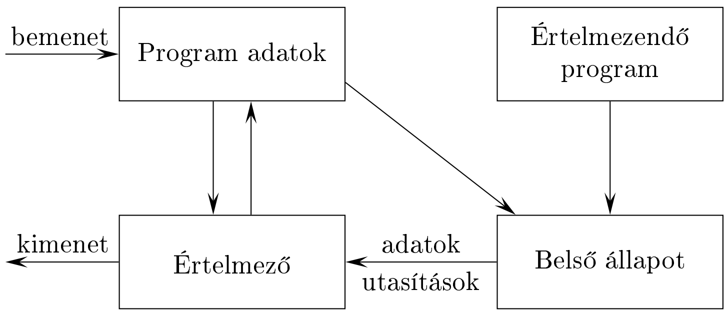
4.6. Virtuális gép, értelmező

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 (4.2. ábra):

1. értelmező (interpreter, state machine, execution engine),

2. belső állapot (internal state, stack),

3. értelmezendő program (utasítások az adott nyelven),



4.2. 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.

4.7. Modell–Nézet–Vezérlő

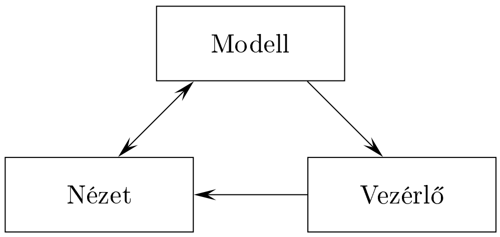
A minta interaktív alkalmazás fejlesztését kívánja támogatni olyan rugalmas szerkezettel, amelyben a felhasználói felületet érintő változtatások egyszerűen megvalósíthatóak. A minta lényege, hogy az adatokat és a rajtuk végzett műveleteket elválasszuk egymástól. Ezt három fő komponens segítségével valósítja meg (4.3. 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



4.3. 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.

4.8. 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.

5. 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?

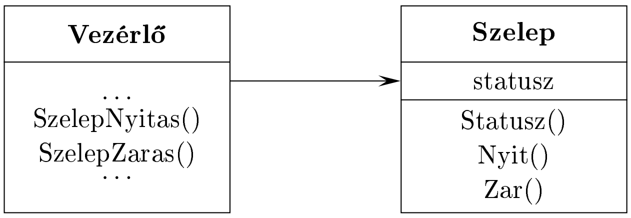
Ennek egyik legfontosabb oka, hogy a tapasztalt tervezők rendszerint bevált tervek alapján hozzák létre egy új rendszer tervét. Egy-egy jó tervet használnak újra meg újra, ezek ismerete és használata teszi őket jó szakemberekké.

Ezeket a terveket, tervrészeket nevezzük *tervmintáknak* (design patterns). Hasonló módon használhatóak a tervezésben, mint ahogy a programozás során a kód egyes részeit kód-újrafelhasználás segítségével állítjuk elő.

5.1. 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.

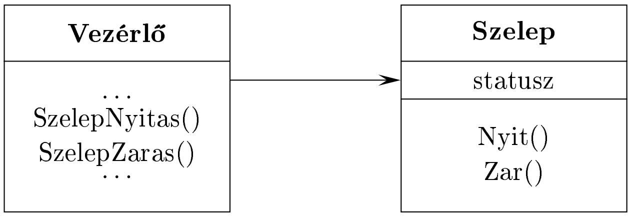
Példa „mindenható osztályra" az alábbi vezérlő:



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;
    ...
}
```

5.2. 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.

5.3. 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.

5.4. 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.

6. 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.

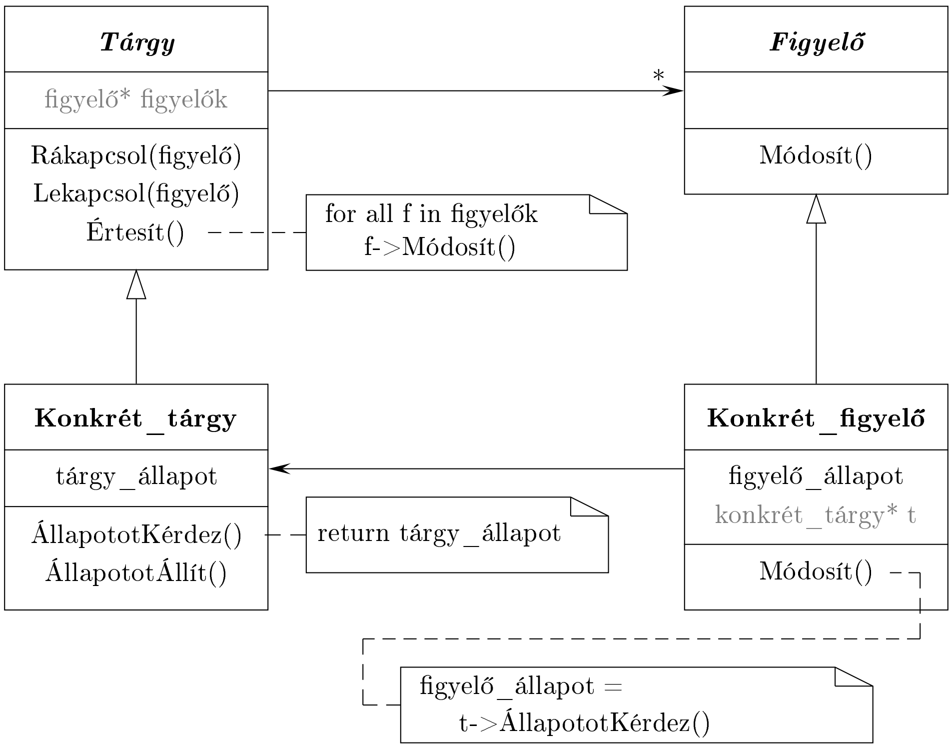
Motiváció: Egy gyakori mellékhatás, ami egy rendszer együttműködő osztályokra bontása során felmerül, a kapcsolatban álló objektumok közötti konzisztencia fenntartásának szükségessége.

Erre egy példa, amikor egy statisztika százalékos adatait akarjuk megjeleníteni táblázat, oszlopdiagram vagy kördiagram formájában. Egyidejűleg több formában is láthatók az adatok. Ekkor a statisztika a megjelenítés tárgya, a diagramoknak pedig mindig annak aktuális állapotát kell mutatniuk, anélkül hogy a diagramok egymásról tudnának. Ha bármelyiken változtatás történik, akkor az megjelenik a statisztikában, és az értesíti az összes diagramot. Ebben a példában a statisztika a *tárgy* (subject), a diagramok pedig a *figyelők* (observers).

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.

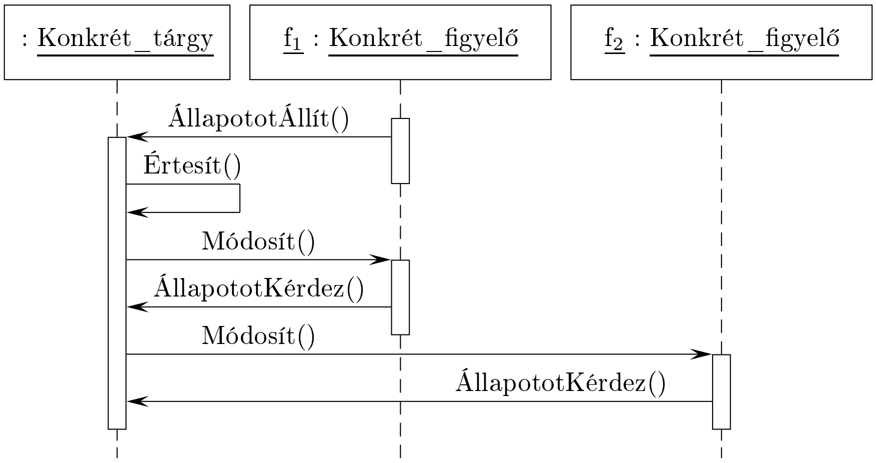
Szerkezet:



Elemek:

- Tárgy: Ismeri a figyelőit, amelyek száma tetszőleges. Lehetőséget biztosít figyelő objektumok hozzávételére és eltávolítására.
- Figyelő: Meghatározza a módosítási felületét azoknak az objektumoknak, amelyeket értesíteni kell.
- Konkrét_tárgy: Tartalmazza a konkrét_figyelők számára érdekes állapotot. Állapotváltozás esetén értesíti a figyelőit.
- Konkrét_figyelő: Hivatkozik a konkrét_tárgy objektumra. Tartalmazza az állapotot, amelynek konzisztensnek kell lennie a tárgyéval. Implementálja a figyelő módosítási felületét, ezzel biztosítva a konzisztenciát.

Együttműködés: A konkrét_tárgy értesíti a figyelőit olyan változás esetén, amely inkonzisztenciát okozhatna az aktuális állapot és a figyelői állapota között. Az értesítés után a konkrét_figyelő lekérdezheti a tárgy állapotát. Ennek segítségével állítja helyre a konzisztenciát. Ezt szemlélteti a következő szekvenciadiagram két figyelő esetén.



6.1. Példa kód

```
class Targy;

class Figyelo
{
public:
    virtual ~Figyelo();
    virtual void Modosit(Targy *valtott_targy) = 0;
protected:
    Figyelo();
};

class Targy
{
public:
    virtual ~Targy();
    virtual void Rakapcsol(Figyelo *f);
    virtual void Lekapcsol(Figyelo *f);
    virtual void Ertesit();
protected:
    Targy();
private:
    List<Figyelo *> *figyelok;
};

void Targy::Rakapcsol(Figyelo *f)
{
    figyelok->Append(f);
}
```

```
void Targy::Lekapcsol(Figyelo *f)
{
    figyelok->Remove(f);
}

void Targy::Ertesit()
{
    ListIterator<Figyelo *> it(figyelok);
    for ( it.First(); !it.Done(); it.Next() )
    {
        it.CurrentItem()->Modosit(this);
    }
}
```

6.2. Esettanulmány

Egy lehetséges alkalmazás az idő megjelenítése a számítógépen. Ekkor az időzítő a megfelelő műveletekkel kiegészítve a tárgy objektumból örököltethető, figyelő pedig lehet ennek digitális vagy analóg megjelenítése.

```
class Idozito : public Targy
{
public:
    Idozito();
    virtual int Ora();
    virtual int Perc();
    virtual int Masodperc();
    void Tikk();
};

void Idozito::Tikk()
{
    // idő állítása
    Ertesit();
}

class DigitalisOra : public Figyelo
{
public:
    DigitalisOra(Idozito *i);
    virtual ~DigitalisOra();
    virtual void Modosit(Targy *t);
    virtual void Rajzol();
private:
    Idozito  *targy;
};
```

```
DigitalisOra::DigitalisOra(Idozito *i)
{
    targy = i;    targy->Rakapcsol(this);
}

DigitalisOra::~DigitalisOra()
{
    targy->Lekapcsol(this);
}

void DigitalisOra::Modosit(Targy *t)
{
    if ( t == targy )    Rajzol();
}

void DigitalisOra::Rajzol()
{
    int ora = targy->Ora();
    int perc = targy->Perc();
    int mp = targy->Masodperc();
    // a digitális óra kirajzolása
}

class AnalogOra : public Figyelo
{
    AnalogOra(Idozito *i);
    virtual ~AnalogOra();
    virtual void Modosit(Targy *t);
    virtual void Rajzol();
private:
    Idozito  *targy;
};
```

Ezek után egy digitális és egy analóg óra, amelyek ugyanazt az időt mutatják:

```
Idozito *idozito = new Idozito();
AnalogOra *a_ora = new AnalogOra(idozito);
DigitalisOra *d_ora = new DigitalisOra(idozito);
```

7. Iterátor

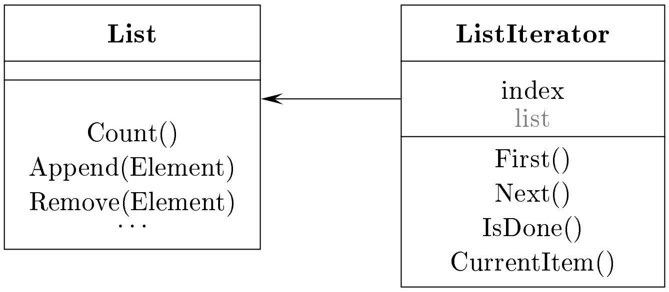
Név, osztály: iterátor (iterator); viselkedési (behavioral).

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

Más nevek: cursor.

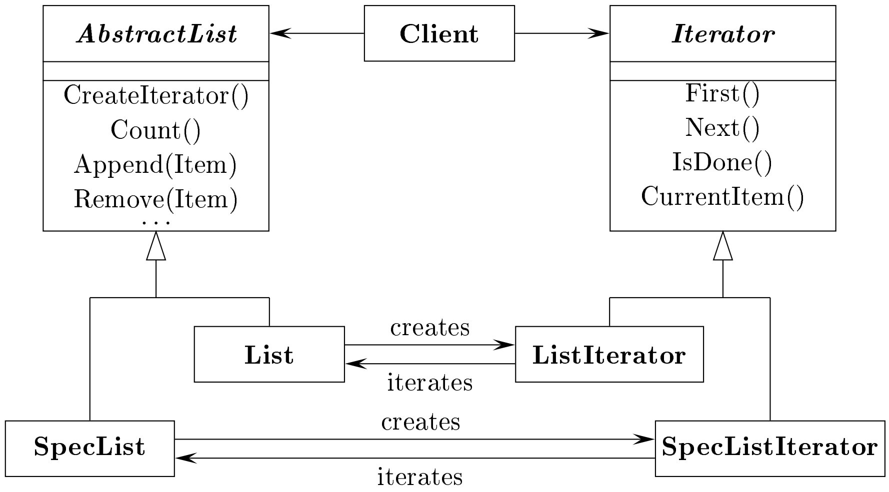
Motiváció: Egy aggregátumnak (például lista) biztosítania kell az alkotóelemeinek elérhetőségét, a belső szerkezet felfedése nélkül. Továbbá lehetséges, hogy különböző módokon akarjuk bejárni az elemeket (normál, fordított). Ugyanakkor nem szerencsés ennek érdekében túl sok művelettel ellátnunk a lista felületét, még akkor sem, ha esetleg minden bejárási módot előre jelezni tudunk. Ugyanazon a listán egyszerre több bejárás is „futhat”.

Erre egy lehetséges megoldás, ha például egy `List` osztály a `ListIterator` osztályt használja a következő ábra szerint.



A `ListIterator` használata előtt meg kell adni a bejárandó listát. A bejárás elválasztása a listától lehetővé teszi, hogy különböző bejárásokat definiáljunk, anélkül, hogy a `List` osztály interfészét nagyon megnövelnénk.

Most a bejáró művelet és a lista szoros kapcsolatban áll. A felhasználónak tudnia kell, hogy egy listát jár be, más szerkezetre a megoldás nem alkalmazható. Ezt öröklődés segítségével oldhatjuk meg, amelyben a bejárást általánosítjuk.



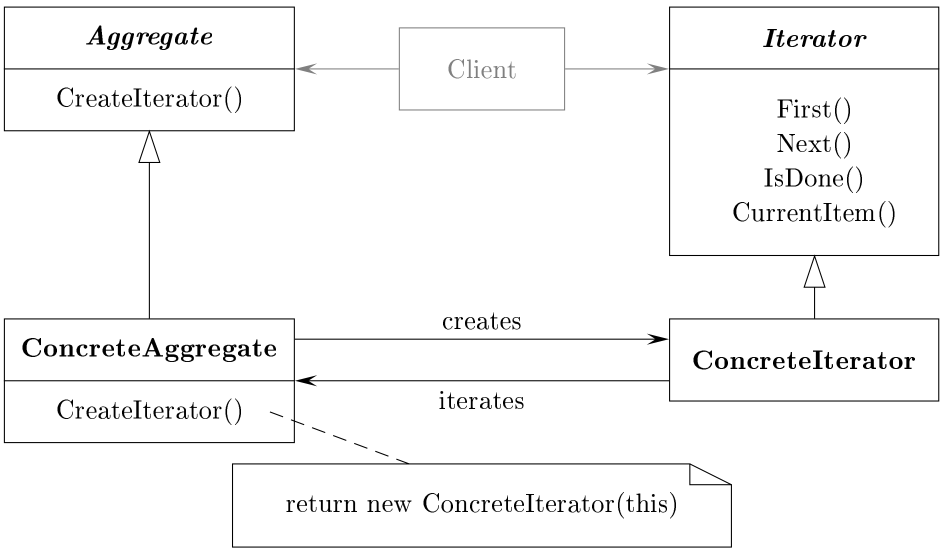
Az iterátort az aktuális lista osztálytól függetlenül kell létrehozni, ezért minden lista objektumnak létre kell hoznia a megfelelő iterátort.

Erre való a `CreateIterator` művelet. (Ez az *Absztrakt gyártó* minta egy esete.)

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:



Elemek:

- Iterator: megadja a bejárás felületét (elemek elérése, továbblépés).
- ConcreteIterator: megvalósítja az Iteratort, és nyilvántartja az aktuális elemet.
- Aggregate: megadja az Iterator objektum létrehozásának a felületét.
- ConcreteAggregate: megvalósítja az Iterator létrehozását egy megfelelő példány megadásával.

Együttműködés: A `ConcreteIterator` példánya nyilvántartja az aggregátum aktuális elemét, és meg tudja határozni a bejárás következő elemét.

7.1. Példa kód

```
template <class Item> class List
{
public:
    List(long size = DEF_CAPACITY);
    long Count() const;
    Item& Get(long index) const;
    ...
};

template <class Item> class Iterator
{
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};

template <class Item> class ListIterator : public Iterator<Item>
{
public:
    ListIterator(const List<Item> *li);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private:
    const List<Item> *_list;
    long _current;
};

template <class Item> ListIterator<Item>::
    ListIterator(const List<Item> *li) : _list(li), _current(0) {}
```

```
template <class Item> void ListIterator<Item>::First()
{
    _current = 0;
}

template <class Item> void ListIterator<Item>::Next()
{
    _current++;
}

template <class Item> bool ListIterator<Item>::IsDone() const
{
    return _current >= _list->Count();
}

template <class Item> Item ListIterator<Item>::CurrentItem() const
{
    if ( IsDone() )    throw IteratorOutOufBounds;
    return _list->Get(_current);
}
```

7.2. Esettanulmány

Egy lehetséges alkalmazás egy vállalat alkalmazottait kinyomtató eljárás. Tegyük fel, hogy rendelkezésünkre áll az alkalmazottak (`Employee`) listája (`List`), és az alkalmazottak osztály rendelkezik egy alkalmazott adatait kinyomtató `Print` eljárással. Ekkor a megfelelő eljárás a következő.

```
void PrintEmployees(Iterator<Employee*> &it)
{
    for ( it.First(); !it.IsDone(); it.Next() )
        it.CurrentItem()->Print();
}
```

Az eljárás használatához létre kell hozni a megfelelő iterátorokat.

```
List<Employee*> *employees;
    ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

8. Állapot

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

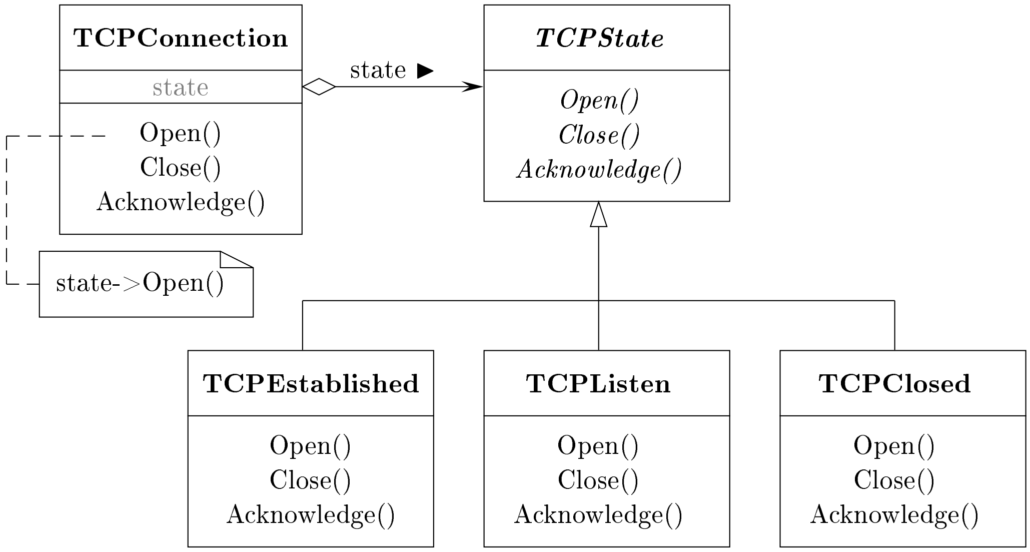
Cél: Lehetővé tenni, hogy egy objektum megváltoztassa a viselkedését, ha az állapota változik. A hatása olyan, mintha az objektum megváltoztatná az osztályát.

Más nevek: objects for states.

Motiváció: Tegyük fel, hogy adott egy `TCPConnection` osztály, amely hálózati kapcsolatot reprezentál. Ennek egy objektuma különböző állapotokat vehet fel: létrejött, figyelő, lezárt. Amikor egy ilyen objektum üzenetet kap más objektumoktól, az aktuális állapot függvényében eltérően viselkedik. Például egy megnyitási igény hatása más ha az állapot lezárt, illetve létrejött.

A minta alapötlete egy absztrakt osztály bevezetése, esetünkben ez legyen a `TCPState`. Ez reprezentálja a különböző (hálózati kapcsolati) állapotokat. Az osztályban deklarálunk egy olyan felületet, amely közös lenne az összes olyan osztályban, amelyek a különböző állapotokhoz tartoznának. Ezen absztrakt osztályból származtatott osztályok valósítják meg az állapotokra jellemző viselkedéseket.

Ha az objektum állapota megváltozik, akkor a változásnak megfelelően cseréli az aggregációs állapot-objektumot.

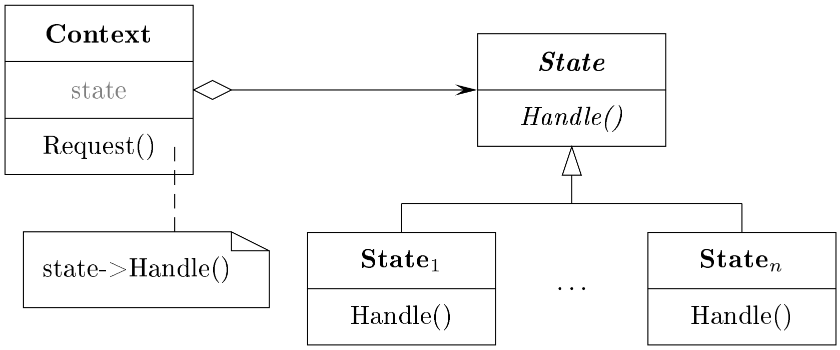


Felhasználhatóság: Az állapot minta használható az alábbi esetekben:

- 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:



Elemek:

- Context: Az ügyfeleknek (kliens objektumok) megadja a felületet. Az aktuális állapot meghatározásával együtt karbantartja a `state` változót úgy, hogy az egy megfelelő $\texttt{State}_i$ osztály példánya legyen.
- State: Meghatározza a speciális állapotok közös felületét.
- State_i: Minden osztály a Context egy állapotához tartozó speciális viselkedést implementál.

Együttműködés:

- A Context osztály objektuma továbbítja az állapotspecifikus igényeket az aktuális State_i osztály objektumának.
- Egy Context objektum átadhatja önmagát, mint paramétert az igényt kezelő State objektumnak. Így szükség esetén a State objektum elérheti azt.
- Context az elsődleges felület a kliensek számára. A kliensek egy ilyen objektumot a State objektumok segítségével konfigurálhatnak. Miután egy Context objektum konfigurált, a klienseknek nem kell direkt módon kezelniük a State objektumokat.
- Vagy a Context vagy a State_i osztályok határozzák meg a következő állapotot, az állapotátmenet feltételével együtt.

8.1. Esettanulmány

```
class TCPOctetStream;
class TCPState;

class TCPConnection
{
public:
    TCPConnection();
    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize()
    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
    TCPState *_state;
};
```

A `_state` adattagban tároljuk a megfelelő viselkedést leíró objektumot.

A `TCPState` osztály szintén tartalmazza az állapotváltoztató műveletet az együttműködés miatt, és minden ilyen objektum műveletének paramétere egy hivatkozás a `TCPConnection` példányára, így biztosítva az adatok elérhetőségét, és az állapot megváltoztathatóságát.

```
class TCPState
{
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};

TCPConnection::TCPConnection()
{
    _state = TCPClosed::Instance();
}
void TCPConnection::ChangeState(TCPState *s)
{
    _state = s;
}
```

```
void TCPConnection::ActiveOpen()
{
    _state->ActiveOpen(this);
}
void TCPConnection::PassiveOpen()
{
    _state->PassiveOpen(this);
}

void TCPConnection::Close()
{
    _state->Close(this);
}

void TCPConnection::Acknowledge()
{
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize()
{
    _state->Synchronize(this);
}
```

TCPState megvalósítja az összes igény alapértelmezett viselkedését (esetünkben ez az üres tevékenység) és az állapotváltoztató műveletet.

```
void TCPState::Transmit(TCPConnection *t, TCPOctetStream *os) {}
void TCPState::ActiveOpen(TCPConnection *t) {}
void TCPState::PassiveOpen(TCPConnection *t) {}
void TCPState::Close(TCPConnection *t) {}
void TCPState::Synchronize(TCPConnection *t) {}
void TCPState::Acknowledge(TCPConnection *t) {}
void TCPState::Send(TCPConnection *t) {}

void TCPState::ChangeState(TCPConnection *t, TCPState *s)
{
    t->ChangeState(s);
}
```

A TCPState osztályból származtatott osztályok valósítják meg az állapotoknak megfelelő viselkedést.

```
class TCPEstablished : public TCPState
{
public:
    static TCPState *Instance();
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};
```

```
class TCPListen : public TCPState
{
public:
    static TCPState *Instance();
    virtual void Send(TCPConnection*);
    ...
};

class TCPClosed : public TCPState
{
public:
    static TCPState *Instance();
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    ...
};
```

A származtatott osztályokban nincs lokális állapot, ezért ezek megoszthatók, így egyetlen példányra van csak belőlük szükség. Ezt adja meg az `Instance` művelet. (Ezért ezek az osztályok *Egykék*.)

Az implementáció:

```
void TCPEstablished::Transmit(TCPConnection *t, TCPOctetStream *os)
{
    t->ProcessOctet(os);
}

void TCPEstablished::Close(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPClosed::Instance());
}

void TCPListen::Send(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::ActiveOpen(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPEstablished::Instance());
}

void TCPClosed::PassiveOpen(TCPConnection *t)
{
    ChangeState(t, TCPListen::Instance());
}
```

9. 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: —.

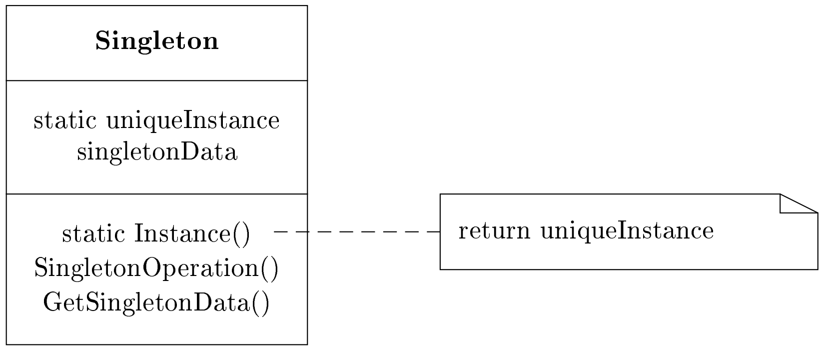
Motiváció: Bizonyos osztályok esetén fontos, hogy pontosan egy példány objektum létezzen. Például egy fájlrendszer, illetve egy ablakkezelő lehet csak egy rendszer esetén. Ezt kell biztosítanunk, és a megfelelő elérhetőséget. Például egy globális változó esetén az elérhetőség biztosított, de a többszöri példányosítás lehetősége megmarad.

Jobb megoldás, ha maga az osztály felel az egyetlen példány létrehozásáért és kezeléséért. Az osztály azt is biztosítja, hogy csak egyetlen példányt lehet létrehozni, többet nem.

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:



Elemek:

- Singleton: Megadja az `Instance` műveletet, amelynek segítségével a kliensek elérhetik az egyetlen példányt. Ez egy osztályművelet. Felelős az egyetlen példány létrehozásáért.

Együttműködés: A kliensek csak az `Instance` műveleten keresztül férhetnek hozzá az osztály egyetlen példányához.

Miért nem lehet az egykét globális vagy statikus objektumként definiálni és az automatikus inicializálásra hagyatkozni?

1. Nem tudjuk biztosítani, hogy csak egy példányt deklarálnak a statikus objektumból.
2. Lehet, hogy nem áll rendelkezésre elegendő információ ahhoz, hogy minden egykét inicializáljunk a statikus inicializációs időben. Lehet, hogy olyan értékekre van szükségünk, amelyeket a program végrehajtása során később ismerünk meg.
3. A C++ nyelv nem definiálja a különböző egységekben található globális objektumok konstruktorainak meghívási sorrendjét. Ezért, ha az egykék között bármilyen függőség áll fenn, akkor hibák léphetnek fel.

9.1. Példa kód

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

Singleton* Singleton::_instance = 0;

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

10. Közvetítő

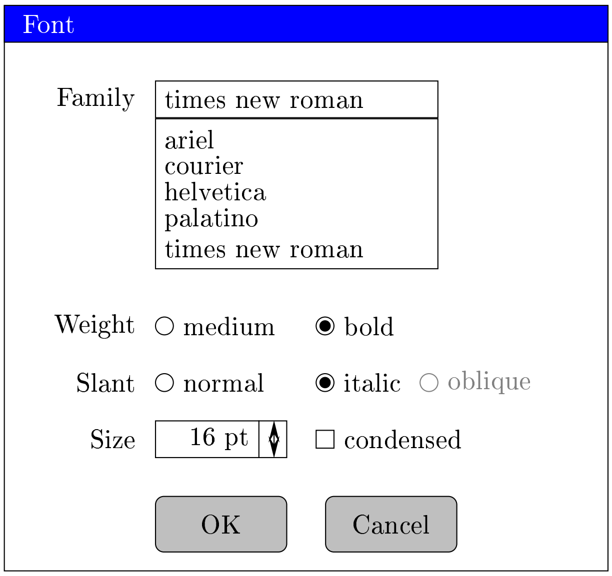
Név, osztály: közvetítő (mediator), viselkedési (behavioral).

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

Más nevek: —.

Motiváció: Az objektumelvű tervezés során a rendszer viselkedését az objektumok között osztjuk szét. Ennek eredménye lehet egy olyan szerkezet, amelyben sok kapcsolat jön létre az objektumok között. A legrosszabb esetben minden objektum ismeri az összes többit. Ez a nagymértékű összekapcsolódás gátja lehet az újrafelhasználhatóságnak, a rendszer jellege monolitikussá válik. A viselkedés módosítása is nehézkes, mert az túl sok objektum között oszlik el.

Példaként tekintsünk egy fontválasztó dialógusablakot, amelyben több elem (gomb, menü, lista) szerepel.

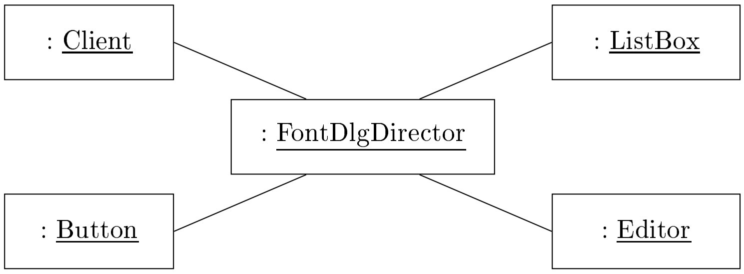


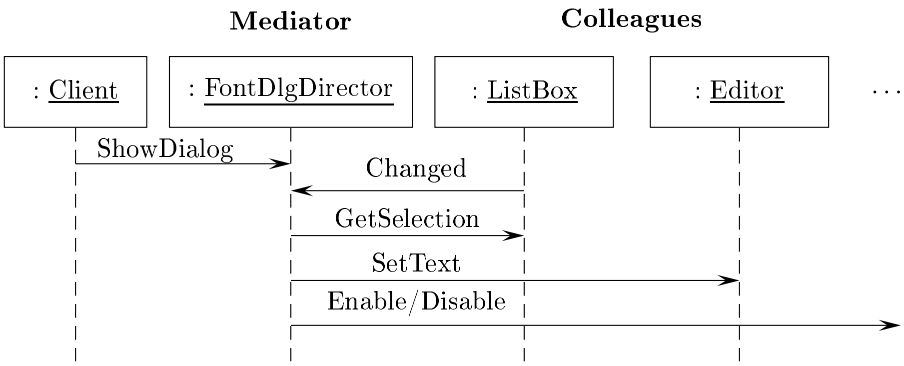
Az elemek között összefüggés van. Ennek megfelelően nem lehet minden esetben aktivizálni egy elemet, illetve nem lehet akármit választani benne.

Különböző dialógusablakok esetén az elemek közötti összefüggés eltérő, ezért az elemeket minden esetben az adott alkalmazás szerint kell specializálni, hogy megfelelően működjenek együtt. Ha ezt származtatással oldanánk meg, akkor túl sok osztály jönne létre.

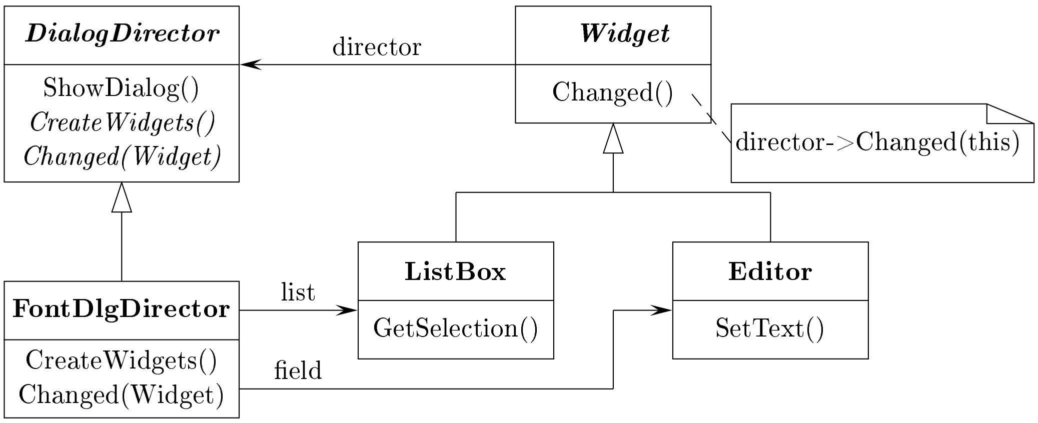
Ezt megoldhatjuk egy közvetítő objektum bevezetésével, amely tartalmazza az elemek együttes viselkedését, egymásra hatását. Egy közvetítő felelős egy csoportba tartozó objektumok kölcsönhatásainak vezérléséért és irányításáért. Ez egy közbeeső elem, amely lehetővé teszi, hogy a csoport tagjai direkt módon ne hivatkozzanak egymásra. Az objektumok (kollégák) csak a közvetítőt ismerik, így csökken a kapcsolatok száma.

Esetünkben egy `FontDlgDirector` objektumot vezethetünk be közvetítőként, amely ismeri az elemeket.





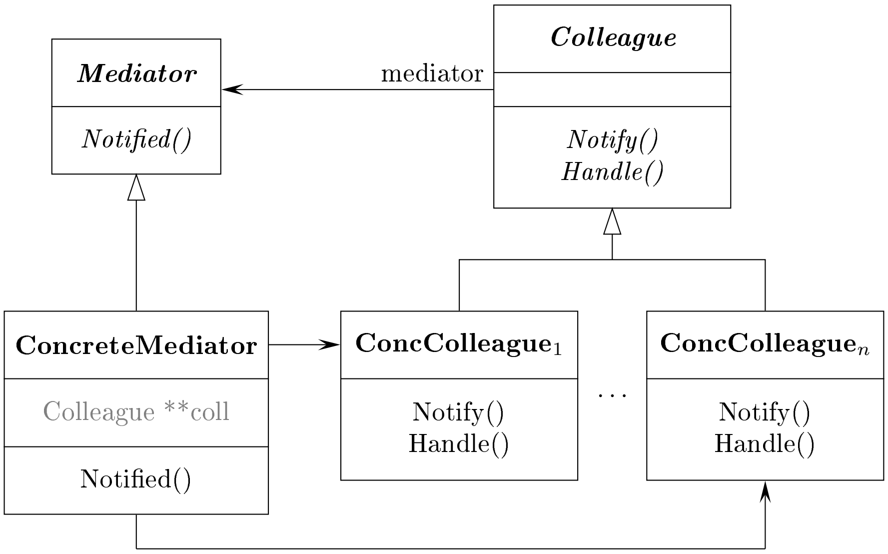
Az osztálydiagram, felhasználva az absztrakt `DialogDirector` és `Widget` osztályt:



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:



Elemek:

- Mediator: A csoportba tartozó objektumok kommunikációs felületét definiálja. (Erre nincs szükség, amikor csak egyetlen közvetítőt használnak az objektumok.)
- ConcreteMediator: Az együttes viselkedést implementálja az objektumok koordinálásával. Ismeri és kezeli a csoport objektumait.
- Colleague: Absztrakt osztály a csoportba tartozó objektumokhoz.
- ConcColleague$_i$: Az csoportba tartozó objektumok konkrét osztályai. Minden ilyen osztály ismeri a közvetítő objektumát. Minden objektum a közvetítőn keresztül kommunikál a csoport többi objektumával a direkt kommunikáció helyett.

Együttműködés: A kollégák a közvetítő objektumnak küldenek és attól fogadnak üzeneteket. A közvetítő az együttes viselkedést úgy valósítja meg, hogy az egyes igényeket a megfelelő kollégáknak továbbítja. A kommunikáció megvalósítható a figyelő minta segítségével is. Ekkor a közvetítő a figyelő, a kollégák pedig a tárgyak.

10.1. Példa kód

```
class Colleague;
class Coll_1;
...
class Coll_n;

class Mediator
{
public:
    virtual ~Mediator();
    virtual void Notified(Colleague*) = 0;
protected:
    Mediator();
    virtual void CreateColleagues() = 0;
};

class Colleague
{
public:
    virtual ~Colleague();
    virtual void Notify()    { mediator->Notified(this); }
    virtual void Handle() = 0;
protected:
    Colleague(Mediator*);
    Mediator *mediator;
};

class Coll_i
{
public:
    Coll_i(Mediator *m);     { mediator = m; }
    void Handle();
};

class ConcreteMediator : public Mediator
{
public:
    ConcreteMediator();
    virtual ~ConcreteMediator();
    void Notified(Colleague*);
protected:
    void CreateColleagues();
private:
    Colleague **coll;
};

void ConcreteMediator::CreateColleagues()
{
    coll = new Colleague*[k];
```

```
    ...
    coll[j] = new Coll_i(this);
    ...
   // coll elemeinek inicializálása
}

void ConcreteMediator::Notified(Colleague *c)
{
    for ( int i = 0; i < k; i++ )
        if ( coll[i] != c )
            // ha kell az objektum értesítése: coll[i]->Handle();
}
```

10.2. Esettanulmány

Egy lehetséges alkalmazás a már bemutatott font meghatározás.

```
class DialogDirector
{
public:
    virtual ~DialogDirector();
    virtual void ShowDialog();
    virtual void Changed(Widget*) = 0;
protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};

class Widget
{
public:
    Widget(DialogDirector*);
    virtual void Changed();
    virtual void HandleMouse(MouseEvent &e);
    ...
private:
    DialogDirector *_director;
};

void Widget::Changed()
{
    _director->Changed(this);
}

class ListBox : public Widget
{
public:
    ListBox(DialogDirector*);
    virtual const char *GetSelection();
```

```
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent &e);
    ...
};

class Editor : public Widget
{
public:
    Editor(DialogDirector*);
    virtual const char *GetText();
    virtual void SetText(const char *text);
    virtual void HandleMouse(MouseEvent &e);
    ...
};

class Button : public Widget
{
public:
    Button(DialogDirector*);
    virtual void SetText(const char *text);
    virtual void HandleMouse(MouseEvent &e);
    ...
};

void Button::HandleMouse(MouseEvent &e)
{
    ...
    Changed();
};

class FontDlgDirector : public DialogDirector
{
public:
    FontDlgDirector();
    virtual ~FontDlgDirector();
    virtual void Changed(Widget*);
protected:
    virtual void CreateWidgets();
private:
    Button *ok;
    Button *cancel;
    ListBox *fontlist;
    Editor *fontname;
};

void FontDlgDirector::CreateWidgets()
{
    ok = new Button(this);
    cancel = new Button(this);
    fontlist = new ListBox(this);
```

```
    fontname = new Editor(this);
    // lista feltöltése, elemek elhelyezése az ablakban
}

void FontDlgDirector::Changed(Widget *w)
{
    if ( w == fontlist )
        fontname->SetText(fontlist->GetSelection());
    else if ( w == ok )
        // a font kiválasztása, és a dialógus bezárása
    else if ( w == cancel )
        // dialógus bezárása
}
```

11. Feljegyzés

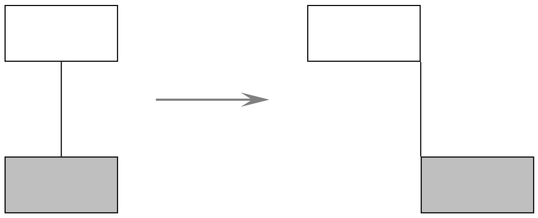
Név, osztály: feljegyzés (memento), viselkedési (behavioral).

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

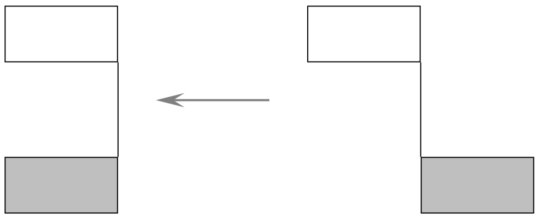
Más nevek: token.

Motiváció: Vannak olyan esetek, amikor szükséges egy objektum belső állapotának megjegyzése. Ezt kell tennünk, ha a rendszerben támogatni akarjuk a visszavonás (undo) műveletet. Ehhez el kell mentenünk egy megelőző állapotot, amit vissza lehet majd állítani. Ugyanakkor az objektumok elrejtik az adattagjaikat, azaz az állapotukat, ezért ezt kívülről nem lehet menteni az elrejtés megsértése nélkül.

Tekintsünk például egy grafikus szerkesztőt, amely támogatja az objektumok közötti kapcsolatokat. Két téglalapot például összeköthetünk vonallal, majd az egyik téglalapot eltolhatjuk. Az összekötő vonal megfelelően változik (a program gondoskodik erről).



Az eltolás visszavonásánál ügyelni kell arra, hogy nem elegendő csak a téglalapot mozgatni.



Ezt a problémát oldhatjuk meg a feljegyzés minta használatával. A feljegyzés egy objektum, amely egy másik objektum, a feljegyzés eredete, belső állapotának egy pillanatra vonatkozó képét tárolja. A visszavonás az eredet objektumtól igényli a feljegyzés objektumot. Az eredet inicializálja a feljegyzést az aktuális állapotnak megfelelő értékekkel. Csak az eredet képes kommunikálni a feljegyzéssel, a feljegyzés mások számára átlátszatlan.

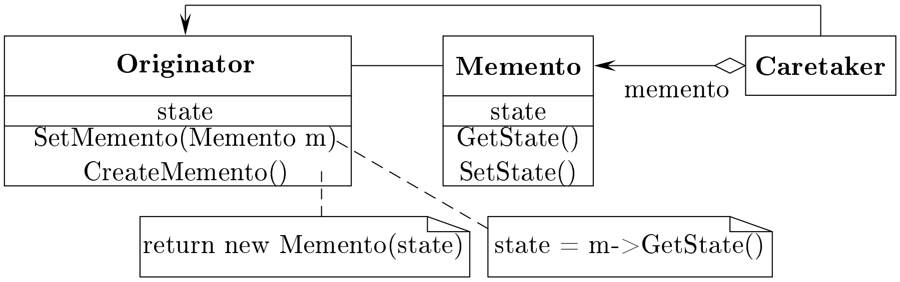
A példánkban az alakzatok közötti kapcsolatokat kezelő objektum az eredet. A visszavonás menete:

1. A program a kapcsolatkezelőtől egy feljegyzést igényel a mozgatás művelet mellékhatásairól, amit az létrehoz.
2. A visszavonás kiadásakor a program ezt a feljegyzést adja meg a kapcsolatkezelőnek.
3. A feljegyzés alapján a kapcsolatkezelő visszaállítja a megelőző állapotot.

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.

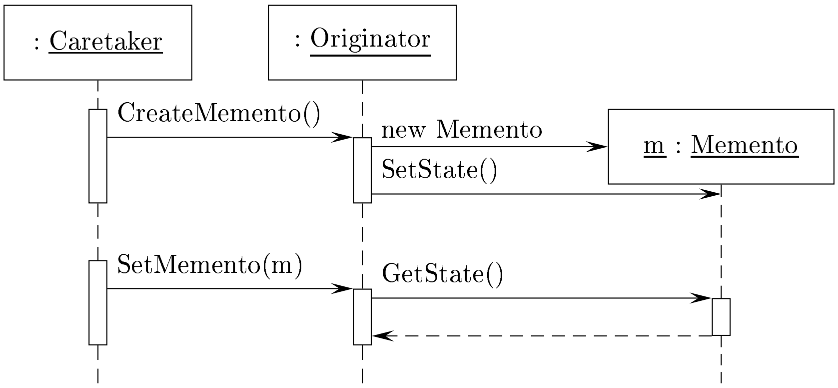
Szerkezet:



Elemek:

- Memento: Az eredet objektum belső állapotát vagy annak egy részét tárolja. Csak az eredet férhet hozzá a tárolt állapothoz. Gyakorlatilag két felülettel rendelkeznek.
- Originator: Létrehoz egy feljegyzést, amelyben a belső állapotát tárolja, és felhasználja a feljegyzést az állapot visszaállításához.
- Caretaker: Felel a feljegyzések tárolásáért, de nem hajt végre műveletet (állapot vizsgálat) a feljegyzésen. Ez a tulajdonképpeni visszavonó művelet.

Együttműködés: Az objektumok együttműködését írja le a következő szekvenciadiagram.



11.1. Példa kód

```
class State;

class Memento;

class Originator
{
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State *_state;
    // ...
};

class Memento
{
public:
    // szűk felület, bármely objektum számára
    virtual ~Memento();
private:     // ezeket csak az Originator éri el
    friend class Originator;
    Memento();
    void SetState(State*);
    State *GetState();
    // ...
private:
    State *_state;
    // ...
};
```

11.2. Esettanulmány

Egy lehetséges alkalmazás a grafikus szerkesztő program. A kapcsolatkezelő objektum a `ConstraintSolver` osztály egyetlen példánya (egyke). Most csak a mozgatást vizsgáljuk, ami egy parancs mintának felel meg.

```
class Graphic;   // grafikai objektumok a szerkesztőben

class ConstraintSolverMemento;

class MoveCommand
{
public:
    MoveCommand(Graphic *target, const Point &delta);
    void Execute();
    void Unexecute();
private:
```

```
    ConstraintSolverMemento *_state;
    Point _delta;
    Graphic *_target;
};

class ContraintSolver
{
public:
    static ConstraintSolver *Instance();
    void Solve();
    void AddConstraint(Graphic *start, Graphic *end);
    void RemoveConstraint(Graphic *start, Graphic *end);
    ConstraintSolverMemento *CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // ...
};

class ConstraintSolverMemento
{
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();
    // és még az állapot
};
```

Az eddigiek alapján a mozgatás parancs és annak visszavonása:

```
void MoveCommand::Execute()
{
    ConstraintSolver *solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento();
    _target->Move(delta);
    solver->Solve();
}

void MoveCommand::Unexecute()
{
    ConstraintSolver *solver = ConstraintSolver::Instance();
    _target->Move(-delta);
    solver->SetMemento(_state);
    // állapot helyreállítása _state alapján
    solver->Solve();
}
```

12. Parancs

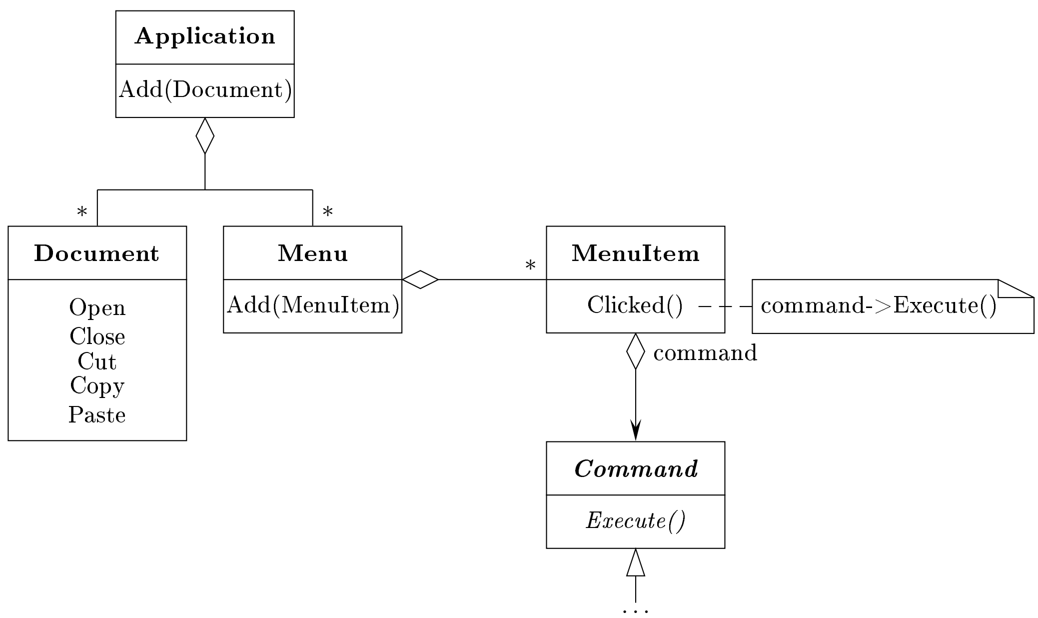
Név, osztály: parancs (command), viselkedési (behavioral).

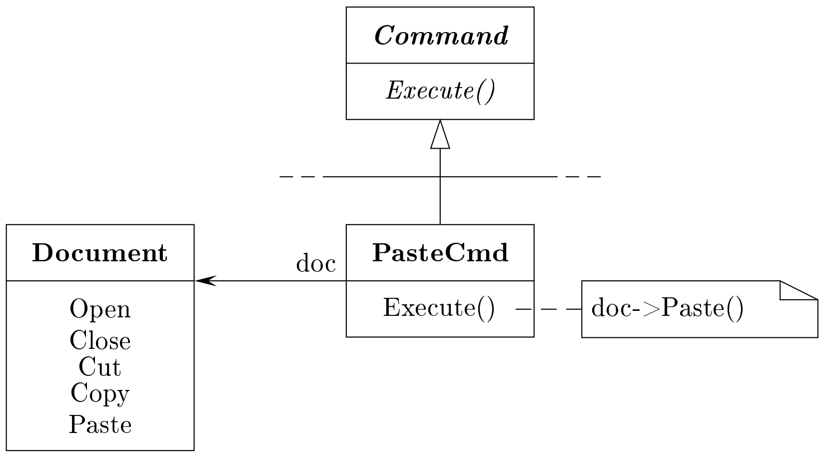
Cél: Egy igény objektumba ágyazása, így lehetővé téve, hogy a kliensek különböző igényeket adjanak ki, és ezeket tároljuk, visszavonjuk.

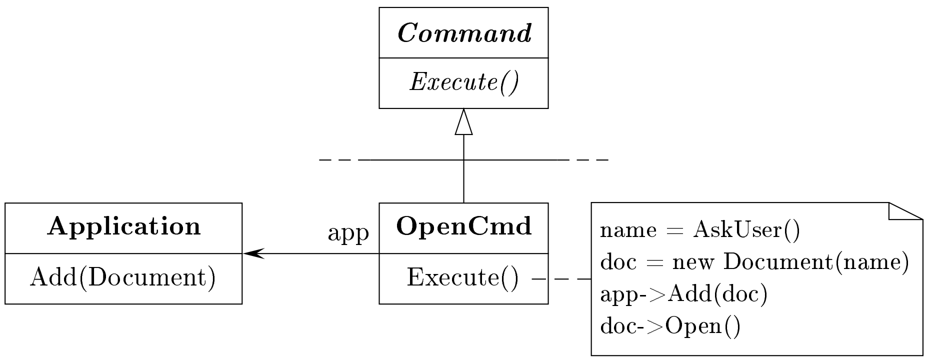
Más nevek: action, transaction.

Motiváció: Van amikor úgy kell egy igényt (parancsot) feldolgoznunk, hogy nem tudunk semmit magáról a műveletről vagy a művelet fogadójáról. Ilyen esetre példa, ha felhasználói felület készítő eszközt szeretnénk létrehozni. Ebben szerepelnek gombok, menük, amelyekhez akciók tartoznak. Ugyanakkor az eszköz nem valósíthatja meg az akciót, mert csak az eszközt használó alkalmazás ismeri azt.

A parancs minta segítségével az igényeket objektumokká alakítjuk, amelyeket tárolhatunk és továbbíthatunk igény szerint. Az alapötlet az absztrakt `Command` osztály, amely egy felületet ad a végrehajtandó műveletekhez. A legegyszerűbb esetben tartalmaz egy absztrakt `Execute` műveletet. Az ebből származtatott konkrét osztályok határozzák meg a fogadó–akció párosokat úgy, hogy a fogadót példányváltozóként tárolják, és az `Execute` műveletet megvalósítják.



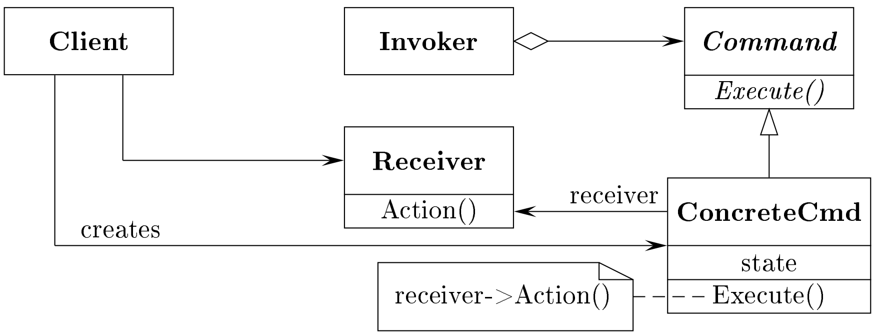




Felhasználhatóság: A parancs minta használható az alábbi esetekben:

- 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:

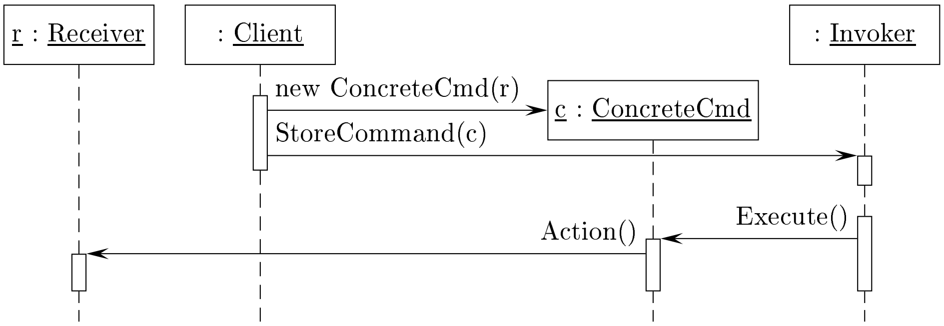


Elemek:

- Command: A művelet végrehajtási felületének deklarálása.
- ConcreteCmd: A fogadó (Receiver) objektum és az akció közötti összekapcsolás definíciója. (PasteCmd, OpenCmd)
- Client: Létrehozza a konkrét parancs objektumot és beállítja a fogadóját. (Application)
- Invoker: A parancs végrehajtását kezdeményezi. (MenuItem)
- Receiver: Ismeri az igényhez tartozó művelet végrehajtásának módját. Bármelyik osztály betöltheti ezt a szerepet. (Document, Application)

Együttműködés:

- A kliens létrehozza a konkrét parancsot és meghatározza a fogadóját.
- Egy kibocsátó (Invoker) objektum tárolja a konkrét parancsot.
- A kibocsátó kiad egy igényt az `Execute` művelet meghívásával. (Visszavonási lehetőség esetén a konkrét parancs tárolja a szükséges állapotokat.)
- A konkrét parancs meghívja a fogadó műveletét az igény kielégítése céljából.



12.1. Példa kód

```
class Command
{
public:
    virtual ~Command();
    virtual void Execute() = 0;
protected:
    Command();
};

class Receiver
{
public:
    Receiver();
    virtual ~Receiver();
    void Action()
```

```
    // ...
};

class ConcreteCmd : public Command
{
public:
    ConcreteCmd(Receiver *r)    { rec = r; }
    virtual void Execute();
private:
    Receiver *rec;
    // ...
};

void ConcreteCmd::Execute()
{
    // visszavonási feljegyzések
    rec->Action();
};
```

(Egyszerű, argumentum nélküli és visszavonást nem igénylő parancsok esetén sablonként is elkészíthető a parancs osztály. A sablon paramétere a fogadó osztály.)

12.2. Esettanulmány

Egy lehetséges alkalmazás a már bemutatott felhasználói felület készítés. Csak az `OpenCmd` és a `PasteCmd` parancsokat vázoljuk. Felhasználjuk a már ismert `Command` osztályt.

```
class OpenCmd : public Command
{
public:
    OpendCmd(Application *a)    { app = a; }
    virtual void Execute();
protected:
    virtual const char *AskUser();
private:
    Application *app;
};

void OpenCmd::Execute()
{
    char *name = AskUser();
    if ( name != 0 )
    {
        Document *doc = new Document(this);
        app->Add(doc);
        doc->Open();
    }
}
```

```
class PasteCmd : public Command
{
public:
    PasteCmd(Document *d)       { doc = d; }
    virtual void Execute();
private:
    Document *doc;
};

void PasteCmd::Execute()
{
    doc->Paste();
}
```

13. 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: –

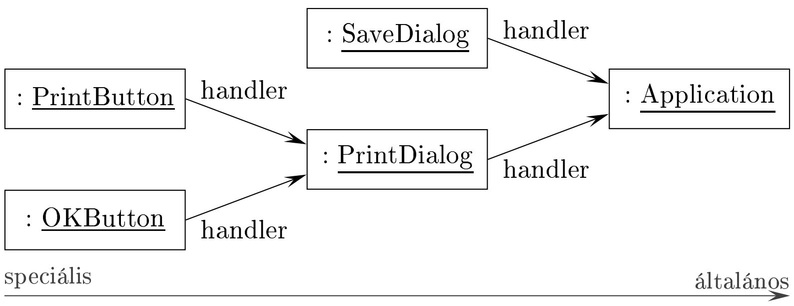
Motiváció: Tekintsük példaként egy grafikus felhasználói felület súgó rendszerét. A felhasználó a felület bármelyik részéről kérhet információt, például egy kattintással. A megjelenített információ függ attól, hogy a felület melyik elemét választottuk ki, és annak a környezetétől. Például egy nyomógombhoz eltérő súgó szöveg tartozhat egy dialógusablakban és az alkalmazás fő ablakában. Ha egy elemhez nem rendeltünk szöveget, akkor a súgónak egy általánosabb információt kell az adott elemről megjelenítenie a környezet függvényében. (Például a gomb helyett a dialógusablakról.)

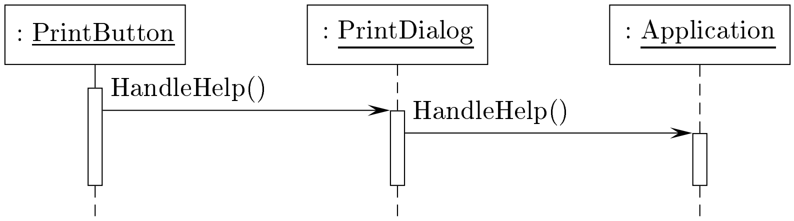
Ennek megfelelően a súgóban szereplő információkat célszerű az általánosság szerint szervezni. Először a speciálisakat, és innen haladhatunk az általánosabbak felé. Egy súgó hívást a számos felületi elem egyike kezel, a környezetnek és a rendelkezésre álló specifikus témáknak megfelelően.

Mi a probléma? Az információt végül is kezelő objektumot nem ismeri a súgási kérelmet kezdeményező objektum (pl:: nyomógomb). Ezért szét kell választanunk a kezdeményező és a kezelő objektumot. Erre szolgál ez a tervminta.

Az alapötlet az, hogy szétválasztjuk a küldőket és a fogadókat megteremtve ezzel az esélyét annak, hogy több objektum is kezelhessen egy igényt. Az igényt a lánc mentén továbbítjuk addig, amíg valamelyik objektum le nem kezeli.

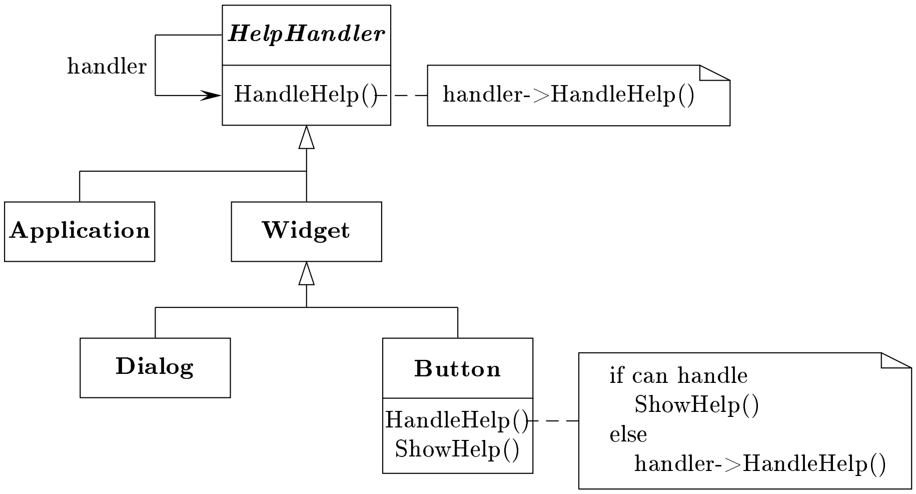
A lánc első objektuma megkapja az igényt. Vagy kezeli, vagy továbbítja a lánc következő tagjának. Az ugyanígy jár el. Az igényt küldő objektum nem tudja pontosan, hogy melyik objektum kezeli az igényt. Azt mondjuk, hogy az igénynek „implicit fogadója” van.





Annak érdekében, hogy az igényt továbbítani lehessen a lánc mentén, és hogy a fogadó implicit maradjon, a láncban szereplő összes objektum közös felülettel rendelkezik. Ez a felület megadja a kezelés módját és a következő elem elérésének mikéntjét.

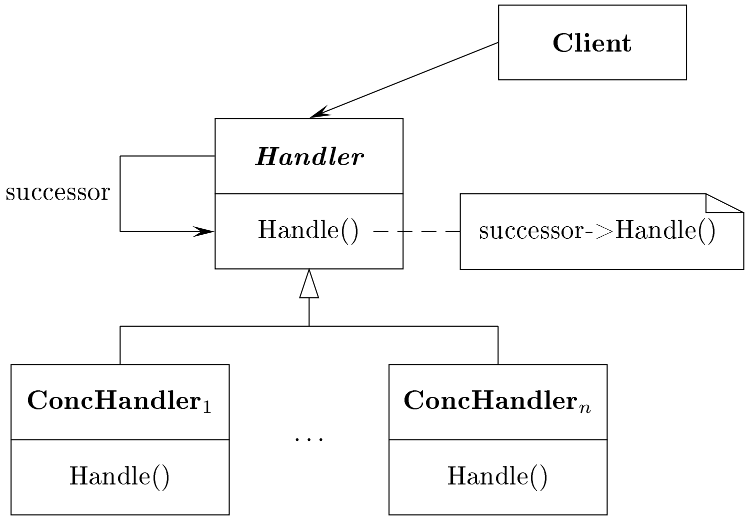
Esetünkben létrehozhatunk egy absztrakt osztályt, amelyből származtatjuk a konkrét kezelőket. Az absztrakt osztály kezelő művelete alapértelmezésben a lánc következő elemének továbbítja az üzenetet. A konkrét osztályokban ezt át lehet definiálni. (A lánc utolsó elemének mindenképpen kezelnie kell az üzenetet.)



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:



Elemek:

- Handler: megadja az igény kezelésének felületét, és implementálja a lánc rákövetkezési relációját.
- ConcHandler$_i$: kezeli a rá vonatkozó igényeket, ha tudja, ellenkező esetben továbbadja; el tudja érni a következő elemet a láncban.
- Client: kiadja az igényt a lánc egy objektumának.

Együttműködés: A kliens kiad egy igényt, amely továbbhalad a láncon, amíg a lánc egy objektuma le nem kezeli azt.

13.1. Példa kód

```
class Request;

class Handler
{
public:
    virtual void ~Handler();
    virtual void Handle(Request *r);
protected:
    Handler(Handler *s = 0) { _successor = s; }
protected:
    Handler *_successor;
};

Handler::Handle(Request *r)
{
    if ( _successor )   _successor->Handle(r);
}
```

13.2. Esettanulmány

Egy lehetséges alkalmazás a bemutatott súgó rendszer.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler
{
public:
    HelpHandler(HelpHandler *h = 0, Topic t = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler *h, Topic t);
    virtual void HandleHelp();
protected:
    HelpHandler *_successor;
    Topic _topic;
};

HelpHandler::HelpHandler(HelpHandler *h, Topic t)
{
    _successor = h; topic = t;
}

bool HelpHandler::HasHelp()
{
    return _topic != NO_HELP_TOPIC;
}

void HelpHandler::SetHandler(HelpHandler *h, Topic t)
{
    _successor = h; topic = t;
}

void HelpHandler::HandleHelp
{
    if ( _successor )   _successor->Handle(r);
}

class Widget : public HelpHandler
{
protected:
    Widget(Widget *parent, Topic t = NO_HELP_TOPIC);
    Widget *_parent;
};

Widget::Widget(Widget *parent, Topic t) : HelpHandler(parent, t)
{
    _parent = parent;
}
```

```
class Button : public Widget
{
public:
    Button(Widget *d, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // overridden operations
};

Button::Button(Widget *d, Topic t) : Widget(d, t) { }

void Button::HandleHelp()
{
    if ( HasHelp() )  // show help
    else              HelpHandler::HandleHelp();
}

class Dialog : public Widget
{
public:
    Dialog(HelpHandler *h, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // ...
};

Dialog::Dialog(HelpHandler *h, Topic t) : Widget(0)
{
    SetHandler(h, t);
}

void Dialog::HandleHelp()
{
    if ( HasHelp() )
        // show help
    else
        HelpHandler::HandleHelp();
}

class Application : public HelpHandler
{
public:
    Application(Topic t) : HelpHandler(0, t) { }
    virtual void HandleHelp();
    // ...
};

void Application::HandleHelp()
{
    // show list of topics
}
```

A használat:

```
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;

Application *app = new Application(APPLICATION_TOPIC);
Dialog *dlg = new Dialog(app, PRINT_TOPIC);
Button *but = new Button(dlg, PAPER_ORIENTATION_TOPIC);

button->HandleHelp();
```

14. Stratégia

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

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

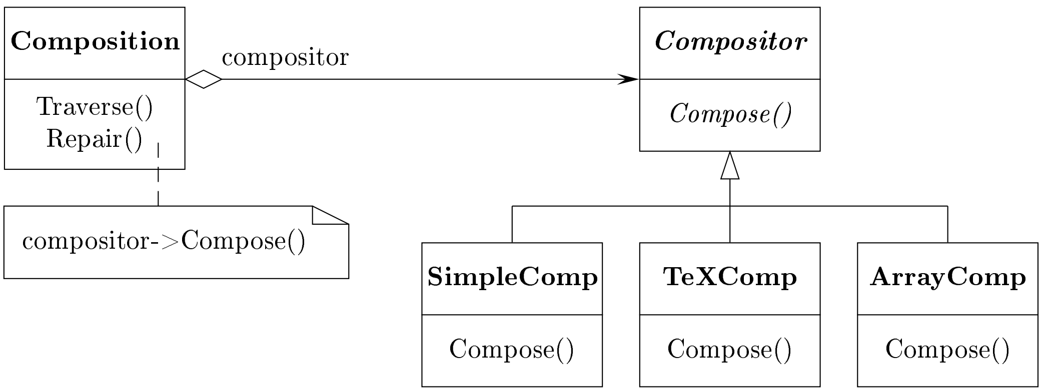
Más nevek: policy

Motiváció: Sok algoritmus létezik szöveg sorokba tördelésére. Egy ilyen algoritmus „kódba drótozása" a felhasználó osztályokban nem célszerű a következők miatt.

- A tördelésért felelős kliensek (osztályok) túl bonyolulttá válnak, ha tartalmazzák a tördelés kódját. Ha többféle tördelést is támogatunk, akkor ezek az osztályok nehezen módosíthatók, karbantarthatók.
- Különböző esetekben eltérő tördelő algoritmusok szükségesek. Az összes lehetséges esetet támogatnunk kell.
- Nehéz új algoritmusokat bevezetni, illetve meglevőket módosítani, ha ezek a kliens belső részét képezik.

Elkerülhetjük ezeket a problémákat, ha olyan osztályokat definiálunk, amelyekbe beágyazhatjuk a különböző tördelő algoritmusokat. Egy így beágyazott algoritmust nevezzünk stratégiának.

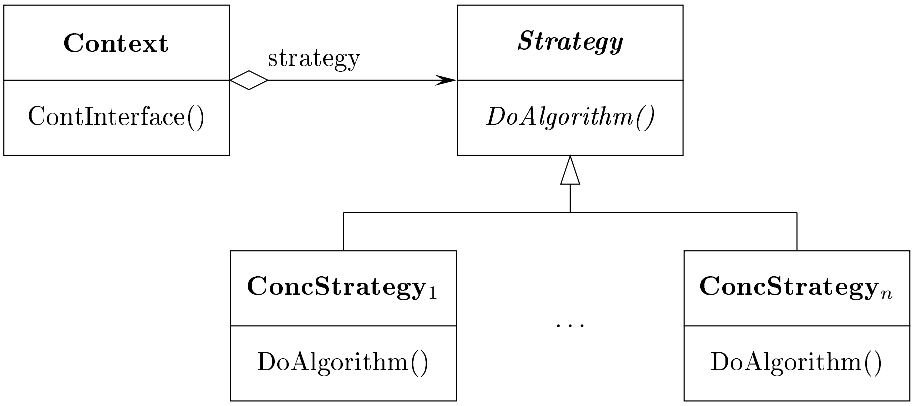
Tegyük fel, hogy a Composition osztály felel egy szöveg tördeléséért. A tördelő algoritmusokat elkülönülten, egy absztrakt Compositor osztály konkrét leszármazottaiban implementáljuk.



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:



Elemek:

- Strategy: megadja az összes felhasználható algoritmus közös felületét; ezt a felületet használjuk a konkrét algoritmusok meghívására.
- ConcStrategy$_i$: az algoritmus implementációja a megadott felülettel.
- Context: egy stratégia objektummal konfigurálható, hivatkozik egy stratégia objektumra, definiálhat egy felületet, amelyen keresztül a stratégia eléri az adatait.

Együttműködés:

- A stratégia és a környezet együttműködve implementálja a megfelelő algoritmust. A környezet átadhat adatokat, vagy akár saját magát is a stratégiának, hogy az algoritmushoz szükséges adatokat biztosítsa.
- A környezet továbbítja a klienstől érkező kérést a stratégiához. A kliensek rendszerint létrehozzák a stratégia objektumot, és átadják azt a környezetnek, de ezután már csak a környezettel állnak kapcsolatban.

14.1. Példa kód

```
class Strategy
{
public:
    virtual ~Strategy();
    virtual void DoAlgorithm() = 0;
protected:
    Strategy();
};

class Context
{
public:
    Context(Strategy *s = 0)   { strat = s; }
    SetStrategy(Strategy *s)   { strat = s; }
    virtual ~Context();
    void ContInterface();
private:
    Strategy *strat;
};

class ConcStrategy : public Strategy
{
public:
    ConcStrategy();
    void DoAlgorithm();
};

void ConcStrategy::DoAlgorithm()
{
    // implementation
};
```

14.2. Esettanulmány

Egy lehetséges alkalmazás a sorok tördelése.

```
class Compositor
{
public:
    virtual int Compose( ... ) = 0;
protected:
    Compositor();
};
```

```
class Composition
{
public:
    Composition(Compositor *c);
    void Repair();
private:
    Compositor *_comp;
    Components *_li;   // the list of components
    int _count;        // number of components
    // data for breaking
};

class SimpleComp : public Compositor
{
public:
    SimpleComp();
    int Compose( ... );
};

class TexComp : public Compositor
{
public:
    TexComp();
    int Compose( ... );
};

class ArrayComp : public Compositor
{
public:
    ArrayComp(int columns);
    int Compose( ... );
};

void Composition::Repair()
{
    // formatting parameters
    int breakCount;
    breakCount = _comp->Compose( ... );
}

Composition *simple = new Composition(new SimpleComp());
Composition *tex = new Composition(new TexComp());
Composition *icon = new Composition(new ArrayComp(10));
```

15. Sablon művelet

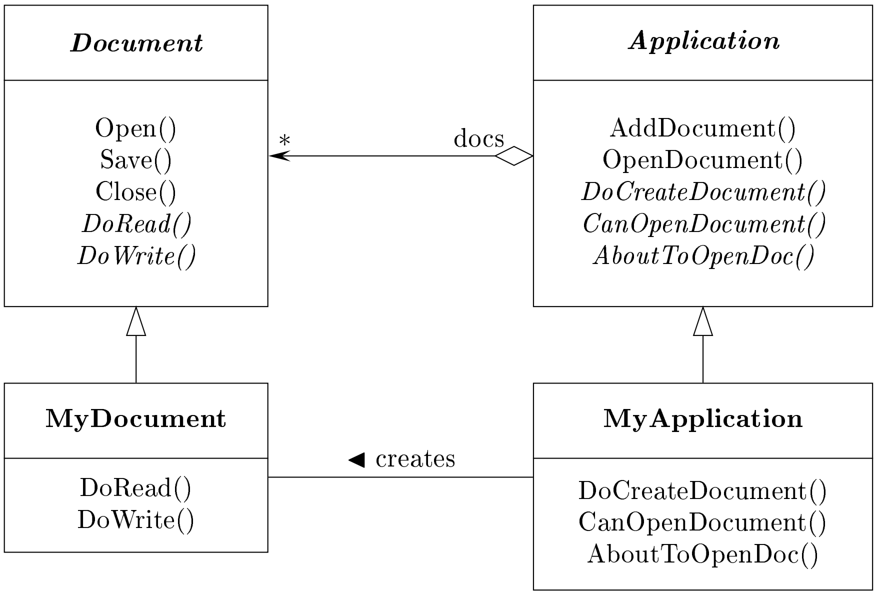
Név, osztály: sablon művelet (template method), viselkedési (behavioral)

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: –

Motiváció: Egy keretrendszerben, amelyben szerepelnek alkalmazás és dokumentum osztályok, az alkalmazás felelős létező dokumentumok megnyitásáért. Egy dokumentumot fájlban tárolunk valamilyen formátumban. A dokumentum objektum tartalmazza az információkat, miután azokat beolvastuk a fájlból.

A keretrendszerrel készített alkalmazásokban az alkalmazás és a dokumentumok osztályokat származtatással hozhatjuk létre az általános osztályokból a speciális igényeknek megfelelően. Például egy rajzoló program egy rajzoló alkalmazást és dokumentumot, egy táblázatkezelő program pedig egy táblázatkezelő alkalmazást és dokumentumot hoz létre.



Az absztrakt `Application` osztály megadja egy dokumentum megnyitásának és beolvasásának algoritmusát az `OpenDocument` műveletben.

```
void Application::OpenDocument(const char *name)
{
    if ( !CanOpenDocument(name) )
    {
        // cannot handle this document
        return;
    }
    Document *doc = DoCreateDocument();
    if ( doc )
    {
        docs->AddDocument(doc);
        AboutToOpenDoc(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

Az eljárás a megnyitás lépéseit definiálja. Ez egy sablon művelet. A származtatott konkrét osztályok adják meg a pontos algoritmust az absztrakt műveletek átdefiniálásával. Esetünkben:

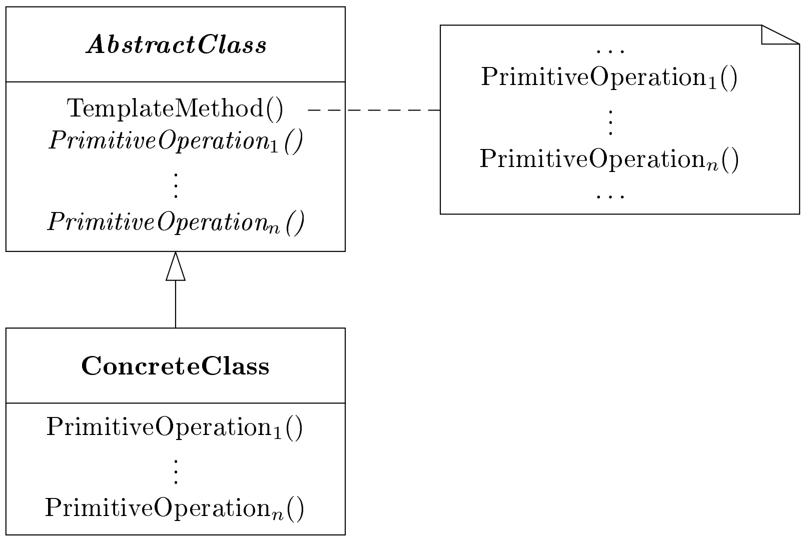
- `CanOpenDocument`,
- `DoCreateDocument`,
- `AboutToOpenDoc` és
- `DoRead`.

A sablon művelet csak ezek sorrendjét adja meg, de a műveletek tartalma a konkrét alkalmazástól illetve dokumentumtól függ.

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:



Elemek:

- AbstractClass: deklarálja az algoritmus lépéseit megadó primitív műveleteket, amelyeket a konkrét osztályokban meg kell valósítanunk; implementálja a sablon műveletet a primitív műveletek felhasználásával.
- ConcreteClass: megvalósítja a primitív műveleteket a konkrét esetnek megfelelően.

Együttműködés: A konkrét osztály az absztrakt osztályra hagyja az algoritmus állandó részének megvalósítását.

15.1. Példa kód

```
class AbstractClass
{
public:
    TemplateMethod();
protected:
    AbstractClass();
    virtual void PrimitiveOperation_1() = 0;
    ...
    virtual void PrimitiveOperation_n() = 0;
};

class ConcreteClass : public AbstractClass
{
public:
    ConcreteClass();
protected:
```

```
    void PrimitiveOperation_1();
    ...
    void PrimitiveOperation_n();
};
```

15.2. Esettanulmány

Egy lehetséges alkalmazás egy képernyőt kezelő program megfelelő része. A képernyő használata előtt biztosítani kell, hogy az elem megkapja a fókuszt, a használat után pedig visszaadja azt. Ezt megtehetjük, ha az összes elemet egy `View` osztályból származtatjuk, amelynek `Display` sablon művelete biztosítja a megfelelő használatot.

```
void View::Display()
{
    SetFocus();
    DoDisplay();
    ResetFocus();
}
```

A kliensek mindig ezt a műveletet hívják meg, a `View` osztályból származtatott osztályokban a `DoDisplay` műveletet kell megvalósítani.

16. Összetétel

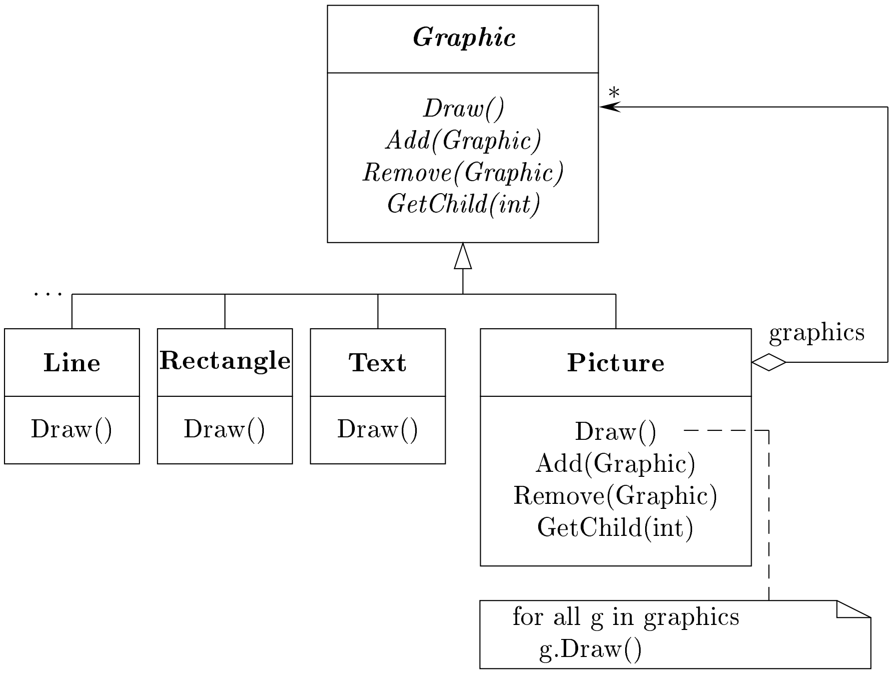
Név, osztály: összetétel (composite), szerkezeti (structural)

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: –

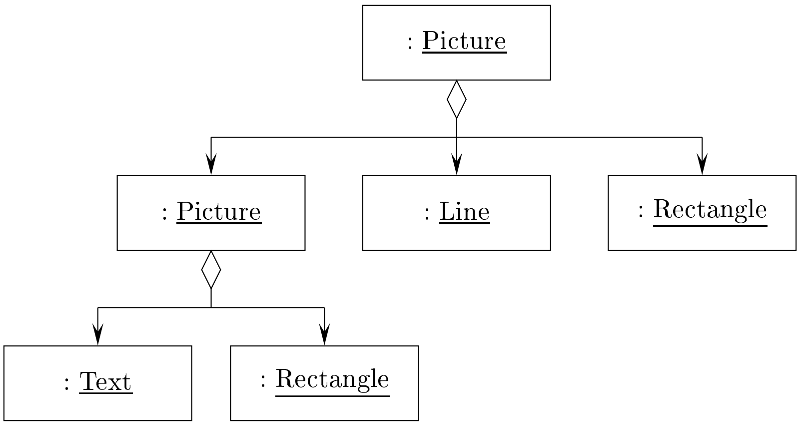
Motiváció: Grafikus alkalmazások lehetővé teszik, hogy a felhasználók bonyolult ábrákat állítsanak elő egyszerű elemekből. Ennek során elemeket csoportosíthatunk, hogy létrehozzunk nagyobb komponenseket, ezekből újabb, még nagyobb komponenseket alkothatunk. A megvalósítás során az egyszerű elemek mellett szükségünk lesz konténerekre is, amelyek a csoportosított komponenseket tartalmazzák.

Ekkor azonban az egyszerű elemeket és a konténereket eltérően kell kezelnünk a megvalósítás során, noha a felhasználó azonosan használja ezeket. A megkülönböztetés bonyolítja a megoldást. A minta megadja miként lehet összetételeket rekurzívan alkalmazni annak érdekében, hogy a megkülönböztetést elkerüljük.



A minta alapötlete egy absztrakt osztály bevezetése, ami tartalmazza az egyszerű elemeket és az összetett szerkezeteket is. Ebben szerepelnek az egyszerű elemek műveletei, és az összetett szerkezetekre vonatkozó műveletek is.

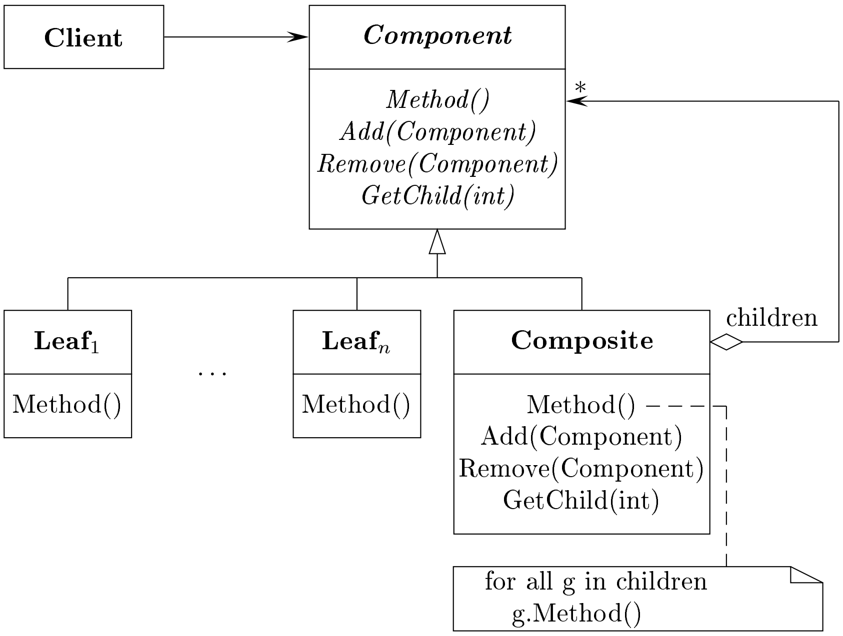
Egy lehetséges objektumdiagram az előző példában:



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:



Elemek:

- Component: az összetételben szereplő objektumok felületét adja meg; megvalósítja az alapértelmezett viselkedését a közös műveleteknek; megadja az összetevő gyerekek elérési és kezelési felületét.

- Leaf$_i$: az összetétel egyszerű elemei, nincsenek leszármazottaik; megadja az egyszerű elem viselkedését.
- Composite: az összetett elemek viselkedését adja meg; tárolja a komponenseket (gyerekek); megvalósítja a gyermekekre vonatkozó műveleteket.
- Client: az összetétel elemeit kezeli.

Együttműködés: A kliensek a `Component` felületén keresztül lépnek kapcsolatba az összetételben szereplő objektumokkal. Levél esetén az igényt azonnal kezeljük, ellenkező esetben az összetétel a gyermekeinek továbbítja az igényt.

16.1. Példa kód

```
class Composite;

class Component
{
public:
    // ...
    virtual Composite *GetComposite() { return 0; }
};

class Composite : public Component
{
public:
    void Add(Component *c);
    // ...
    Composite *GetComposite() { return(this); }
};

class Leaf : public Component
{
};

Composite *comp = new Composite();
Leaf *leaf = new Leaf();
Component *component;
Composite *test;
component = comp;
if ( test = component->GetComposite() )   test->Add(new Leaf());
component = leaf;
if ( test = component->GetComposite() )
    test->Add(new Leaf());    // will not add leaf
```

16.2. Esettanulmány

Egy lehetséges alkalmazás az alkatrészekből összeállított egységek kezelése.

```
class Equipment
{
public:
    virtual ~Equipment();
    const char *Name() { return(name); }
    virtual int Price();
    virtual void Add(Equipment *e);
    virtual void Remove(Equipment *e);
    virtual Iterator<Equipment *> *CreateIterator();
protected:
    Equipment(const char *n);
private:
    const char *name;
};

class FloppyDisk : public Equipment
{
public:
    FloppyDisk(const char *n);
    virtual ~FloppyDisk();
    int Price();
};

class CompositeEquipment : public Equipment
{
public:
    virtual ~CompositeEquipment();
    int Price();
    void Add(Equipment *e);
    void Remove(Equipment *e);
    Iterator<Equipment *> *CreateIterator();
protected:
    CompositeEquipment(const char *n);
private:
    List<Equipment *> equipment;
};

int CompositeEquipment::Price()
{
    Iterator<Equipment *> *i = CreateIterator();
    int total =0;
    for ( i->First(); !i->IsDone(); i->Next() )
        total += i->CurrentItem()->Price();
    delete i;
    return(total);
}
```

```
class Chassis : public CompositeEquipment
{
public:
    Chassis(const char *n);
    virtual ~Chassis();
};
```

`Bus` : `CompositeEquipment` és `Card` : `Equipment` osztályokat az eddigieknek megfelelően adhatjuk meg.

```
Chassis *chassis = new Chassis("PC Chassis");
Bus *bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));
chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The price is " << chassis->Price() << endl;
```

17. Látogató

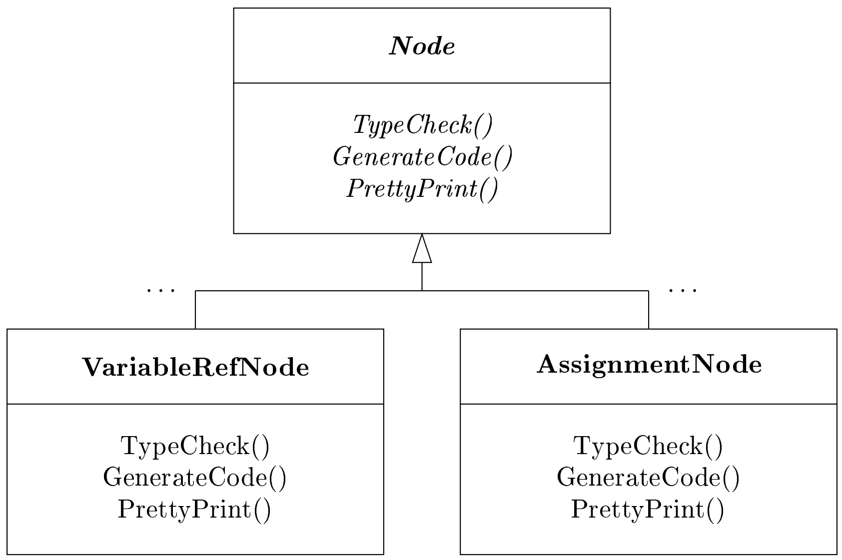
Név, osztály: látogató (visitor), viselkedési (behavioral)

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

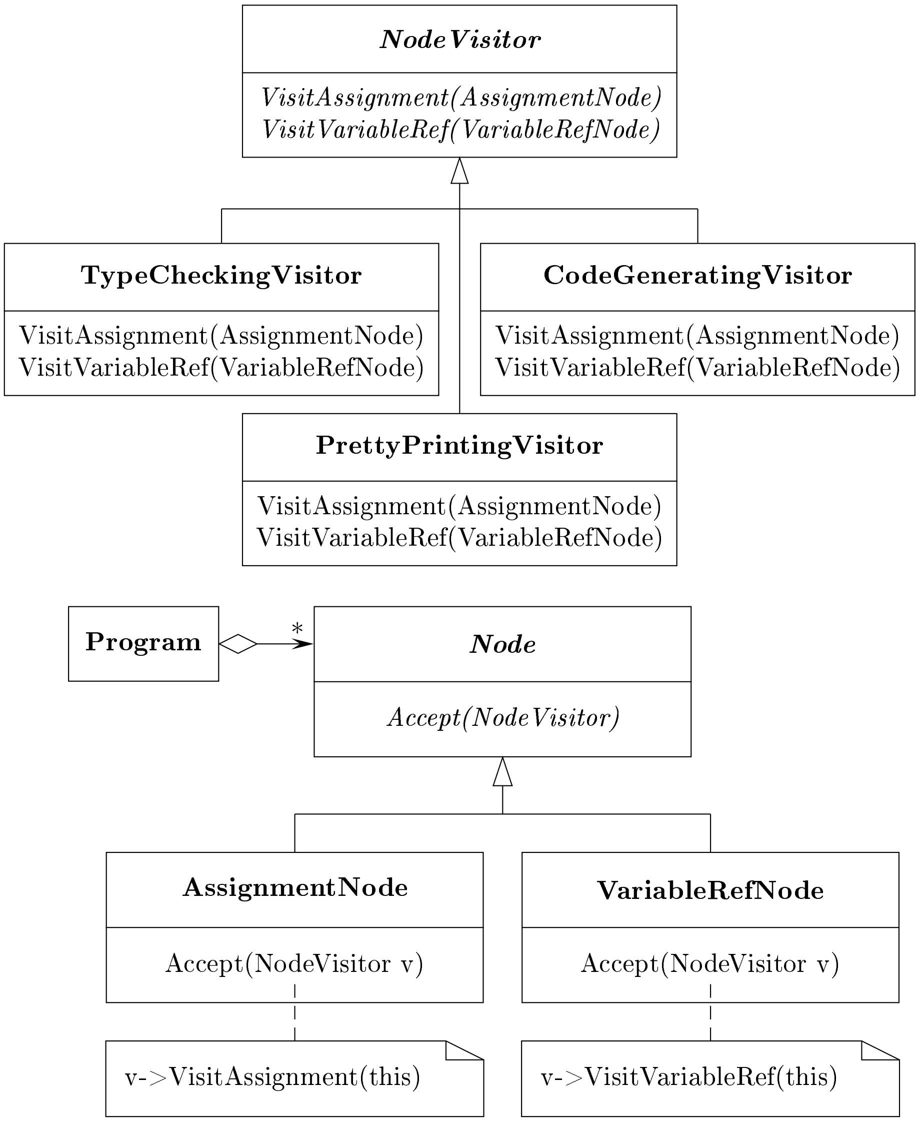
Más nevek: –

Motiváció: Tegyük fel, hogy egy fordítóprogram a programokat egy absztrakt szintaxis fában ábrázolja. Ekkor ezen a fán végre kell hajtani a következő műveleteket: statikus szemantikus elemzés (minden változó definiált-e), kódgenerálás, kód optimalizálás, annak ellenőrzése, hogy minden változó értéket kap-e a felhasználása előtt, ... Ezen kívül felhasználhatjuk a fát szintaxis alapú kiírásra, átszerkesztésre, programmértékek meghatározására.

Majdnem mindig eltérően kell eljárnunk egy változónak megfelelő, illetve egy értékadásnak megfelelő csúcs esetén. Esetleg további osztályokat is megkülönböztethetünk. Így a fa csúcsa egy absztrakt osztály lesz, amelyből származtatjuk a megfelelő osztályokat. (Egy nyelv esetén a szerkezet nem változik.)



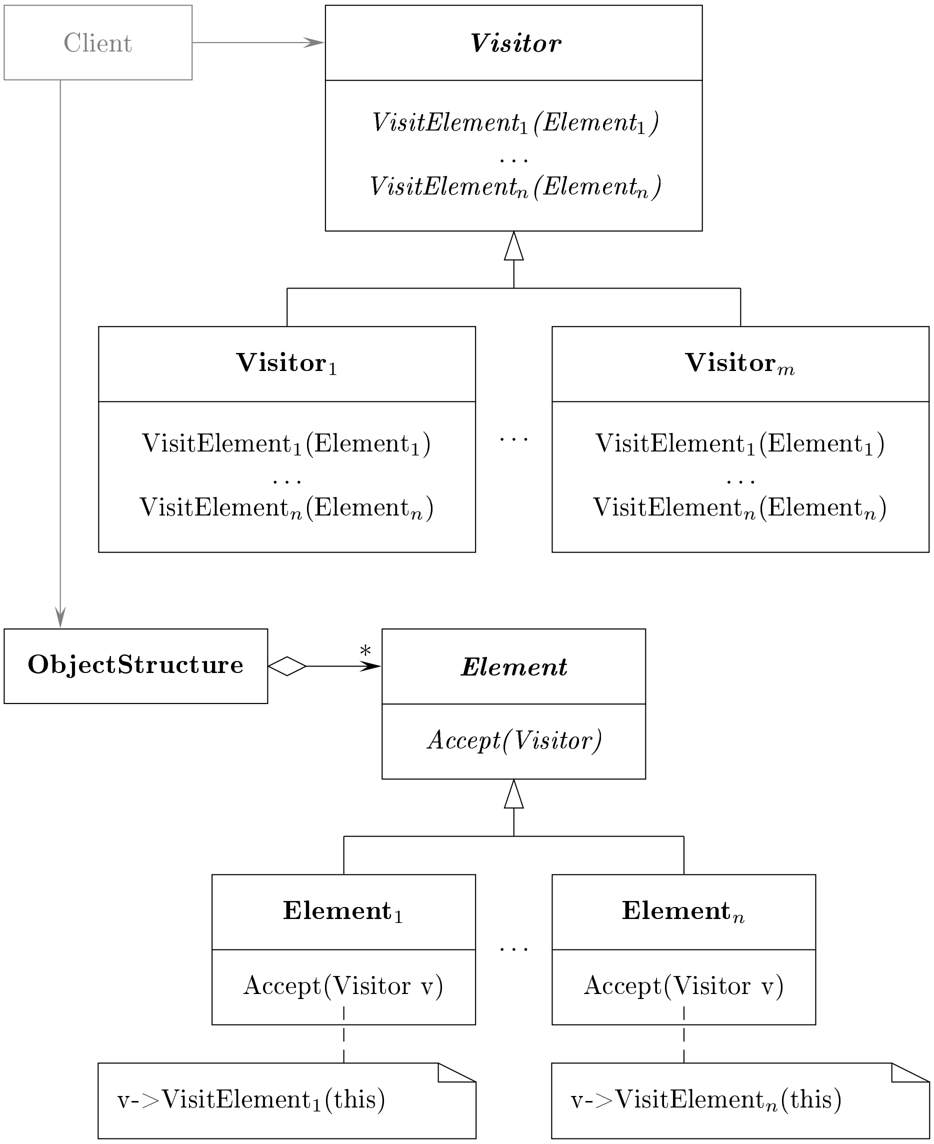
Ebben az esetben új művelet esetén az összes osztályt módosítani kell. Jobb megoldás egy absztrakt műveletosztály bevezetése, amelyben minden típusra megadunk egy „látogató” műveletet, és ebből származtatjuk az egyes konkrét műveleteket. (A szintaxis fa szerkezete nem változik, ezért itt a műveleteket meg lehet adni.)



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:



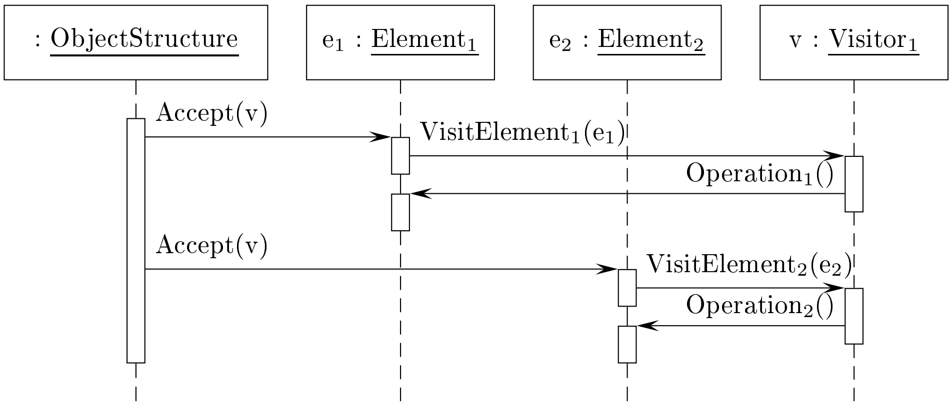
Elemek:

- Visitor: deklarálja a `Visit` műveletet az összes konkrét `Element`$_i$ osztályra, amelynek alapján (szignatúra) meg tudja határozni a meglátogatandó objektum osztályát.
- Visitor$_i$: megvalósítja a megfelelő látogató műveleteket az osztályokra; a művelet egy adott algoritmus megfelelő része, amelynek a környezetét biztosítjuk itt, és nyilvántartjuk a lokális állapotát.
- Element: deklarál egy fogadó (`Accept`) műveletet, amelynek paramétere a látogató.

- Element_j: megvalósítja a fogadó műveletet; megadhat olyan műveletet, amelyet a látogató felhasznál.
- ObjectStructure: el tudja érni az elemeit; biztosíthat egy magas szintű felületet, amelyen keresztül a látogató meglátogathatja az elemeket; lehet egy Composite, vagy tetszőleges gyűjtemény (lista, halmaz).

Együttműködés:

A látogató mintát használó kliensnek először létre kell hoznia egy konkrét látogatót, amelyet felhasználhat a szerkezet bejárása során az elemek meglátogatására.



17.1. Példa kód

```
class Visitor
{
public:
    virtual void VisitElement_1(Element_1 *);
    ...
    virtual void VisitElement_n(Element_n *);
protected:
    Visitor();
};

class Element
{
public:
    virtual ~Element();
    virtual void Accept(Visitor&) = 0;
protected:
    Element();
};

class Element_i : public Element
{
public:
    Element_i();
```

```
    void Accept(Visitor &v) { v.VisitElement_i(this); }
};

class CompositeElement : public Element
{
public:
    virtual void Accept(Visitor &v);
private:
    List<Element *> *children;
};

void CompositeElement::Accept(Visitor &v)
{
    ListIterator<Element *> i(children);
    for ( i.First(); !i.IsDone(); i.Next() )
        i->CurrentItem()->Accept(v);
    v.VisitCompositeElement(this);
}
```

17.2. Esettanulmány

Egy lehetséges alkalmazás az alkatrészekből összeállított egységek kezelése, amelyet kiegészítünk egy látogatóval.

```
class Equipment
{
public:
    virtual ~Equipment();
    const char *Name() { return(name); }
    virtual int Price();
    virtual void Accept(EquipmentVisitor&)
    virtual void Add(Equipment *e);
    virtual void Remove(Equipment *e);
    virtual Iterator<Equipment *> *CreateIterator();
protected:
    Equipment(const char *n);
private:
    const char *name;
};

class EquipmentVisitor
{
public:
    virtual ~EquipmentVisitor();
    virtual void VisitFloppyDisk(FloppyDisk *);
    virtual void VisitCard(Card *);
    virtual void VisitChassis(Chassis *);
    virtual void VisitBus(Bus *);
protected:
```

```
    EquipmentVisitor();
};

void FloppyDisk::Accept(EquipmentVisitor &v)
{
    v.VisitFloppyDisk(this);
}

void Chassis::Accept(EquipmentVisitor &v)
{
    ListIterator<Equipment *> i(equipment);
    for ( i.First(); !i.IsDone(); i.Next() )
        i.CurrentItem()->Accept(v);
    v.VisitChassis(this);
}

class PricingVisitor : public EquipmentVisitor
{
public:
    PricingVisitor()     { total = 0; }
    int GetTotalPrice()  { return(total); }
    void VisitFloppyDisk(FloppyDisk *);
    void VisitCard(Card *);
    void VisitChassis(Chassis *);
    void VisitBus(Bus *);
private:
    int total;
};

void PricingVisitor::VisitFloppyDisk(FloppyDisk *e)
{
    total += e->Price();
}

void PricingVisitor::VisitChassis(Chassis *e)
{
    total += e->Price();
}
```

18. Absztrakt gyártó

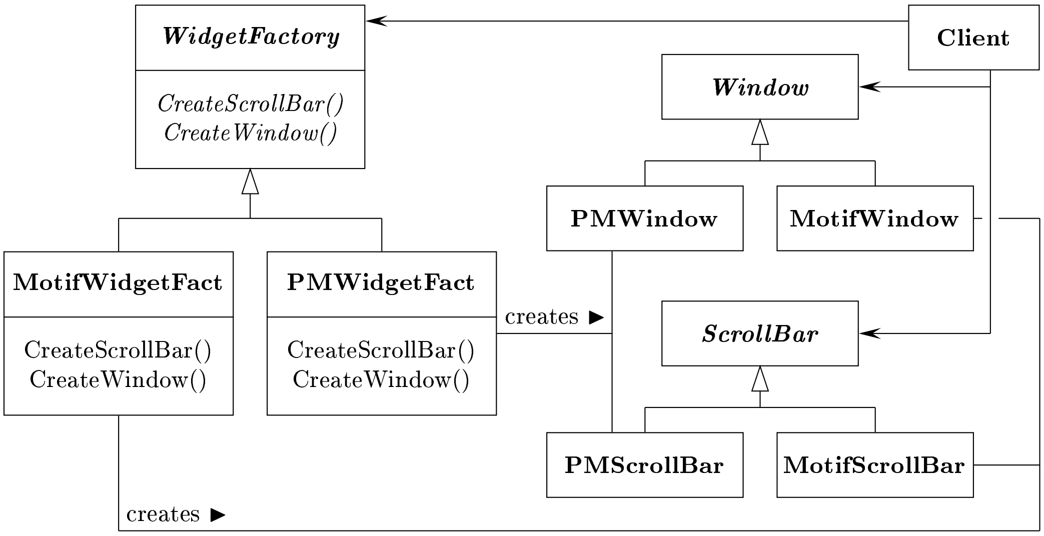
Név, osztály: absztrakt gyártó (abstract factory), létrehozási (object creational)

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

Más nevek: kit

Motiváció: Tegyük fel, hogy olyan felhasználói felületet támogató eszközkészletünk van, amely több szabványt is elfogad. (Motif, Presentation Manager, Windows, ….) Ezekben eltérhet az egyes elemek (widget-ek) kinézete és viselkedése. Ezeket nem szabad a kódba beégetni, ha tényleg hordozható rendszert szeretnénk létrehozni. Azaz nem lehet egy adott szabványnak megfelelő példányokat direkten létrehozni.

A megoldás egy absztrakt WidgetFactory osztály definiálása, amely megadja mindegyik szabványnak megfelelő widget-ek létrehozási felületét. Ugyanakkor minden lehetséges widget-hez egy absztrakt osztályt rendelünk, amelyekből származtatott konkrét osztályok valósítják meg az egyes szabványok megfelelő widget-eit.

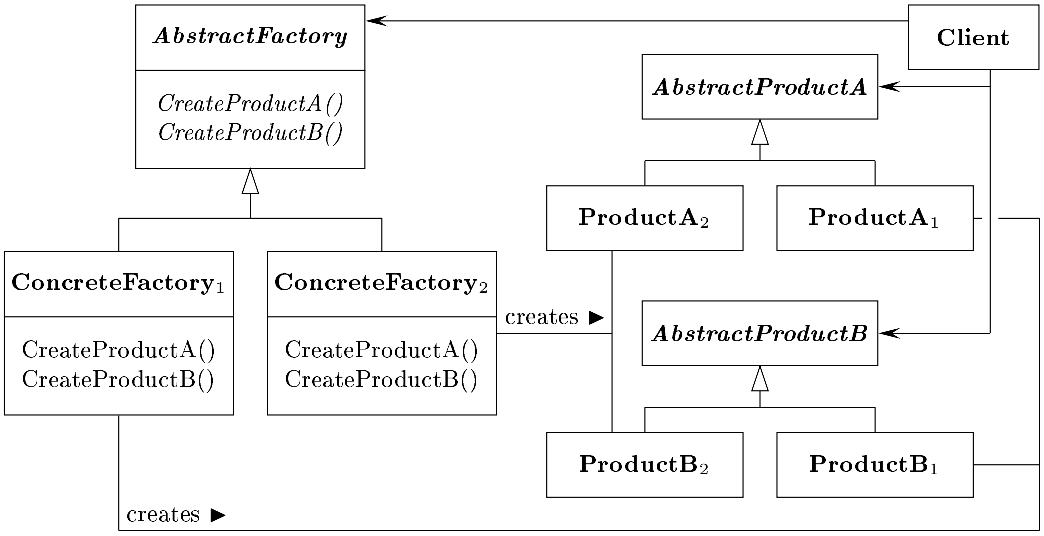


Felhasználhatóság: Az absztrakt gyártó minta használható az alábbi esetekben:

- 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:



Elemek:

- AbstractFactory: deklarálja az absztrakt termékeket előállító műveleteket.
- ConcreteFactory: megvalósítja az előállító műveleteket.
- AbstractProduct: megadja egy adott típusú termékek interfészét.
- ConcreteProduct: definiálja a megfelelő konkrét gyár által létrehozandó termék objektumokat; megvalósítja az AbstractProduct interfészét.
- Client: csak az absztrakt osztályok által meghatározott felületet használja.

Együttműködés: Rendszerint egyetlen konkrét gyártó objektumot (egyke) hozunk létre típusonként futási időben, amely előállítja a megfelelő termék objektumokat. Az absztrakt gyártó a konkrét gyártónak továbbítja a termék létrehozási igényeket.

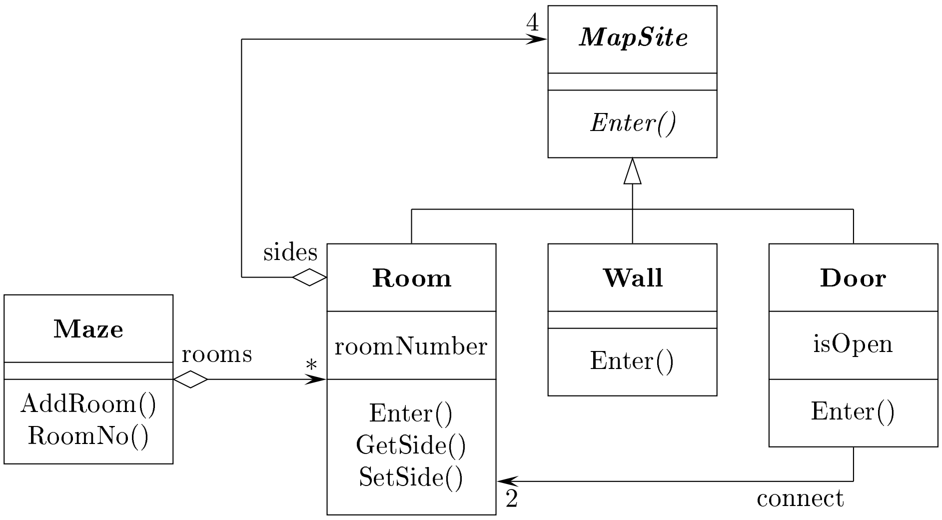
18.1. Példa kód

```
class AbstractProductA;
class AbstractProductB;

class ProductA : public AbstractProductA  { ... };
class ProductB : public AbstractProductB  { ... };
```

```
class AbstractFactory
{
public:
    virtual AbstractProductA *CreateProductA() = 0;
    virtual AbstractProductB *CreateProductB() = 0;
protected:
    AbstractFactory();
};

class ConcreteFactory : public AbstractFactory
{
public:
    static ConcreteFactory *Instance();
    ProductA *CreateProductA();
    ProductB *CreateProductB();
protected:
    ConcreteFactory();
};

ProductA *ConcreteFactory::CreateProductA()
{
    ProductA *a = new ProductA();
    ...
    return(a);
}
```

18.2. Esettanulmány

Egy lehetséges alkalmazás labirintusok (maze) előállítása. A labirintus termekből áll, minden teremnek 4 oldala van. Egy oldal lehet fal vagy ajtó. Az ajtókon át lehet menni, és ekkor a szomszédos terembe juthatunk. Minden ajtó két termet köt össze.



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

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

class Wall : public MapSite
{
public:
    Wall();
    void Enter();
};

class Room : public MapSite
{
public:
    Room(int nr);
    MapSite *GetSide(Direction d) const   { return(sides[d]); }
    void SetSide(Direction d, MapSite *s) { sides[d] = s; }
    void Enter();
private:
    MapSite *sides[4];
    int roomNumber;
};

class Door : public MapSite
{
public:
    Door(Room *r1 = 0, Room *r2 = 0);
    void Enter();
    Room *OtherSideFrom(Room *r);
private:
    Room *r1;
    Room *r2;
    bool isOpen;
};

class Maze
{
public:
    Maze();
    void AddRoom(Room *r);
    Room *RoomNo(int nr) const;
private:
    ...
};
```

Egy kétszobás labirintus felépítése az absztrakt gyártó használata nélkül:

```
Maze *MazeGame::CreateMaze()
{
    Maze *m = new Maze();
    Room *r1 = new Room(1);
    Room *r2 = new Room(2);
    Door *d = new Door(r1, r2);
    m->AddRoom(r1);
    m->AddRoom(r2);
    r1->SetSide(North, new Wall());
    r1->SetSide(East, d);
    r1->SetSide(South, new Wall());
    r1->SetSide(West, new Wall());
    r2->SetSide(North, new Wall());
    r2->SetSide(East, new Wall());
    r2->SetSide(South, new Wall());
    r2->SetSide(West, d);
    return(m);
}
```

A `MazeFactory` a labirintus komponenseit hozza létre:

```
class MazeFactory
{
public:
    MazeFactory();
    virtual Maze *MakeMaze() const    { return(new Maze()); }
    virtual Wall *MakeWall() const    { return(new Wall()); }
    virtual Room *MakeRoom(int nr) const { return(new Room(nr)); }
    virtual Door *MakeDoor(Room *r1, Room *r2) const
                                    { return(new Door(r1, r2)); }
};
```

Ezt felhasználva is elkészíthetjük az előző labirintust.

```
Maze *MazeGame::CreateMaze(Mazefactory &f)
{
    Maze *m = f.MakeMaze();
    Room *r1 = f.MakeRoom(1);
    Room *r2 = f.MakeRoom(2);
    Door *d = f.MakeDoor(r1, r2);
    m->AddRoom(r1);
    m->AddRoom(r2);
    r1->SetSide(North, f.MakeWall());
    r1->SetSide(East, d);
    r1->SetSide(South, f.MakeWall());
    r1->SetSide(West, f.MakeWall());
    r2->SetSide(North, f.MakeWall());
    r2->SetSide(East, f.MakeWall());
    r2->SetSide(South, f.MakeWall());
    r2->SetSide(West, d);
    return(m);
}
```

Ez lényegében ugyanaz mint az első változat. Ugyanakkor most már könnyű különféle (elvarázsolt, aknásított) labirintusokat létrehozni, ha azok megfelelő elemeit megadjuk származtatással az eddigi osztályokból.

```
class EnchantedMazeFactory : public MazeFactory
{
public:
    EnchantedMazeFactory();
    Room *MakeRoom(int nr) const
                { return new EnchantedRoom(nr, CastSpell()); }
    Door *MakeDoor(Room *r1, Room *r2) const
                { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell *CastSpell() const;
};

class BombedMazeFactory : public MazeFactory
{
public:
    BombedMazeFactory();
    Wall *MakeWall() const       { return(new BombedWall()); }
    Room *MakeRoom(int nr) const { return(new RoomWithBomb(nr)); }
};
```

Ezek után egy aknásított labirintus létrehozása a következőképpen történik:

```
MazeGame  game;
BombedMazeFactory f;
game.CreateMaze(f);
```

Hasonlóan lehet elvarázsolt vagy közönséges labirintust is létrehozni.

19. Híd

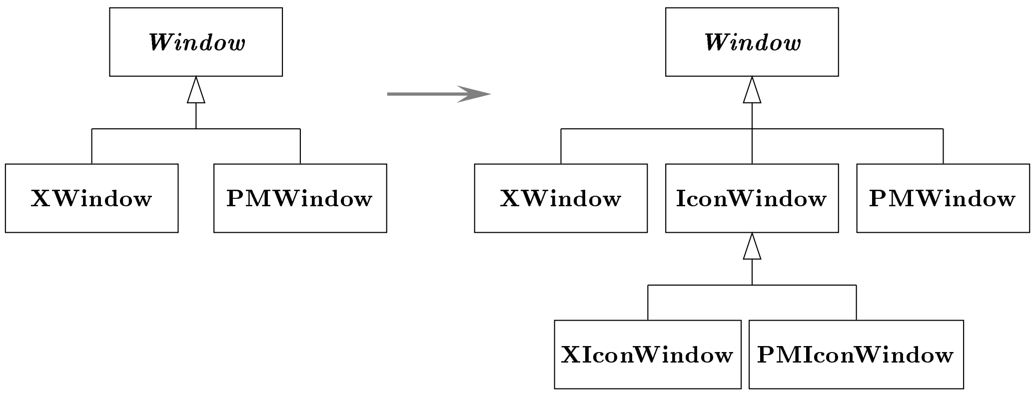
Név, osztály: híd (bridge), szerkezeti (structural)

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

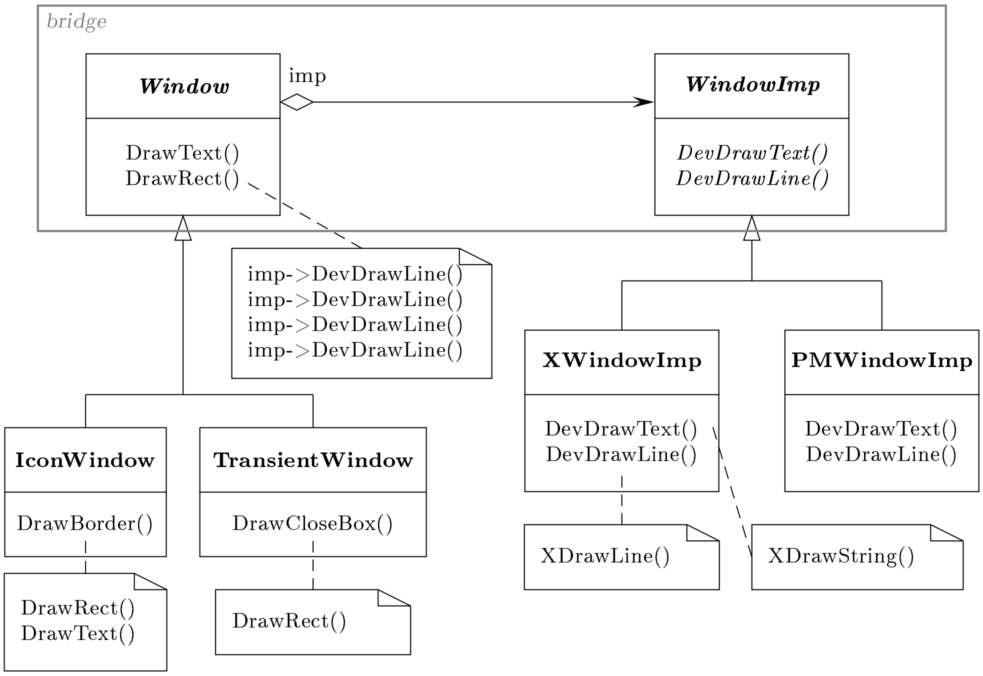
Más nevek: handle/body

Motiváció: Ha egy absztrakcióhoz több lehetséges implementáció tartozik, akkor ezt rendszerint öröklődéssel valósítjuk meg. Az absztrakt osztály megadja a felületet, és a konkrét származtatott osztályok különböző módokon valósítják azt meg. Ez azonban sokszor nem elég rugalmas. Az öröklődés a megvalósítást elválaszthatatlanul az absztrakcióhoz köti, így nehéz módosítani, kiterjeszteni, újrafelhasználni azokat egymástól függetlenül.

Tegyük fel, hogy olyan rendszert kell írnunk, ami például X és PM ablakozó rendszer alatt is használni kívánunk. Ha csak ezt nézzük, akkor be kell vezetnünk egy ablak osztályt, amelyből származtatjuk a megfelelő osztályokat. Ha új felületre akarjuk átvinni a rendszert, vagy több fajta ablak lehet, akkor újabb osztályok bevezetése szükséges.



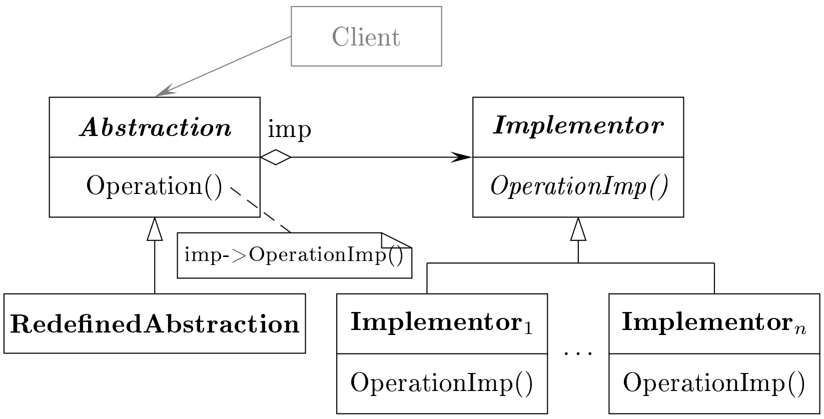
Ráadásul a kliens kódja platformfüggő lesz. Ugyanis, ha létrehozunk egy XWindow objektumot, akkor a Window absztrakciót ehhez kötjük. Ezt elkerülendő két különálló osztályszerkezetet hozunk létre. Az egyik az absztrakciókat, a másik azok megvalósításait tartalmazza.



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:



Elemek:

- Abstraction: megadja az absztrakció felületét; tartalmaz egy hivatkozást az Implementor egy példányára.
- RedefinedAbstraction: kiterjeszti az absztrakciót, annak felületét.
- Implementor: megadja a megvalósító osztályok felületét, ami nem feltétlen egyezik meg az absztrakció felületével (az absztrakció magas szintű, a megvalósítás alacsony szintű műveleteket ad meg).
- Implementor$_i$: megadja a konkrét megvalósítást.

Együttműködés: Az absztrakció a kliens igényeit továbbítja a megvalósítónak.

19.1. Példa kód

```
class Implementor
{
public:
    virtual void OperationImp() = 0;
protected:
    Implementor();
};

class Abstraction
{
public:
    void ChangeImp(Implementor *i) { imp = i; }
    void Operation()               { imp->OperationImp(); }
protected:
    Abstraction(Implementor *i)    { imp = i; }
protected:
    Implementor *imp;
};

class RedefinedAbstraction : public Abstraction
{
public:
    RedefinedAbstraction(Implementor *i) : Abstraction(i) {}
    ...
};

class Implementor_i : public Implementor
{
public:
    Implementor_i();
    void OperationImp();
private:
    ...
};
```

```
void Implementor_i::OperationImp()
{
    ...
}
```

19.2. Esettanulmány

Egy lehetséges alkalmazás a bemutatott ablakozó rendszerekkel kapcsolatos probléma.

```
class WindowImp;

class Window
{
public:
    Window(WindowImp *i)     { imp = i; }
    virtual void DrawContents();
    ...
    // requests forwarded to implementation
    virtual void DrawLine(const Point &p1, const Point &p2);
    virtual void DrawRect(const Point &p1, const Point &p2);
    virtual void DrawText(const char *t, const Point &p);
    ...
protected:
    WindowImp *GetWindowImp();
    View *_contents;
private:
    WindowImp *imp;
};

class WindowImp
{
public:
    virtual void DeviceLine(int x1, int y1, int x2, int y2) = 0;
    virtual void DeviceText(const char *t, int x, int y) = 0;
    virtual void DeviceBitmap(const char *bn, int x, int y) = 0;
    ...
protected:
    WindowImp();
};

void Window::DrawText(const char *t, const Point &p)
{
    GetWindowImp()->DeviceText(t, p.X(), p.Y());
}

void Window::DrawLine(const Point &p1, const Point &p2)
{
    GetWindowImp()->DeviceLine(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

```
class ApplicationWindow : public Window
{
public:
    ...
    void DrawContents();
    ...
};

void ApplicationWindow::DrawContents()
{
    _contents->DrawOn(this);
}

class IconWindow : public Window
{
public:
    ...
    void DrawContents();
private:
    char *bitmapName;
};

void IconWindow::DrawContents()
{
    GetWindowImp()->DeviceBitmap(bitmapName, 0, 0);
}

class XWindowImp : public WindowImp
{
public:
    XWindowImp();
    void DeviceLine(int x1, int y1, int x2, int y2);
    void DeviceText(const char *t, int x, int y);
    ...
private:
    ...
};

class PMWindowImp : public WindowImp
{
public:
    PMWindowImp();
    void DeviceLine(int x1, int y1, int x2, int y2);
    void DeviceText(const char *t, int x, int y);
    ...
private:
    ...
};
```

A megfelelő megvalósítás elérésének műveletét egy absztrakt gyártó felhasználásával implementáljuk. Az absztrakt gyártó tartalmazza az ablakozó rendszer jellemzőit.

```
WindowImp *Window::GetWindowImp()
{
    if ( imp == 0 )
        imp = WindowSystemFactory::Instance->MakeWindowImp();
    return(imp);
}
```

20. Építő

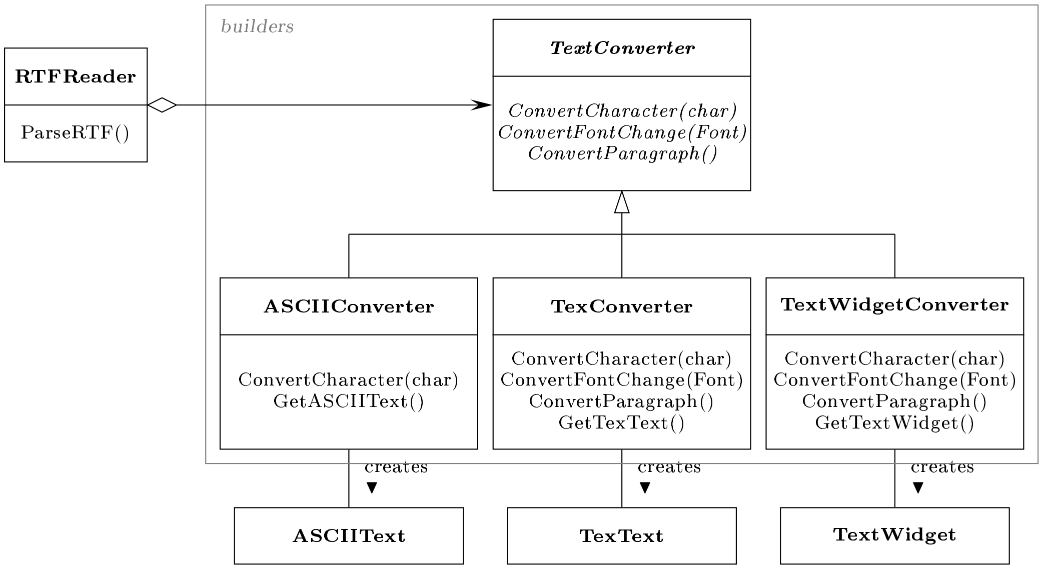
Név, osztály: építő (builder), létrehozási (object creational)

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

Más nevek: –

Motiváció: Egy RTF dokumentumot kezelő programnak tudnia kell az RTF-et több formátumra konvertálni (szöveg, TeX, szerkeszthető text widget). Ugyanakkor a lehetséges formátumok halmaza nyílt, bármikor bővülhet. Ezért a programot könnyen ki kell egészíteni egy új konverzióval, méghozzá módosítás nélkül.

A programot ezért egy TextConverter objektummal konfiguráljuk, ami az RTF-et egy másik formátumra alakítja, pontosabban az átalakítás elemeit tartalmazza. Ahogy a program végig halad az RTF szövegen, meghívja a TextConverter megfelelő konvertáló műveletét. Minden egyes RTF elemre ezt teszi. A TextConverter felel az adat konvertálásáért, és az elem megfelelő formátumú tárolásáért. Ebből származtatjuk az egyes formátumoknak megfelelő konvertáló osztályokat.



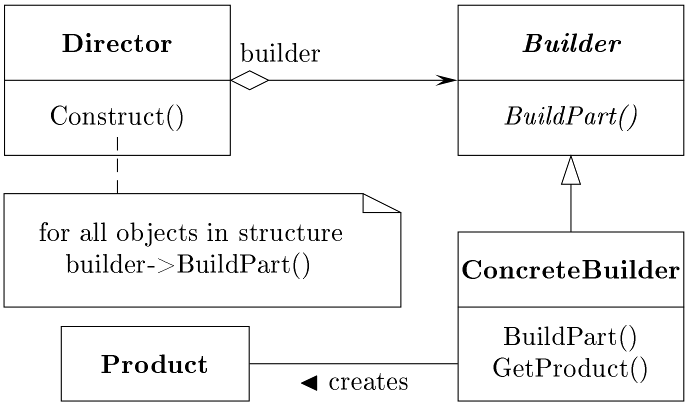
```
void RTFReader::ParseRTF()
{
    while ( t = GetNextToken() )
    {
        switch ( t.Type() )
        {
            case CHAR:  builder->ConvertCharacter(t.Char());
                        break;
            case FONT:  builder->ConvertFontChange(t.Font());
                        break;
            case PARA:  builder->ConvertParagraph();
                        break;
        }
    }
}
```

Normál szöveg esetén csak a karaktert kell átírni, a többi műveletet figyelmen kívül hagyja. Az absztrakt osztályban ennek megfelelően nem tiszta virtuális műveletek vannak, hanem alapértelmezett tevékenységek (SKIP).

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:

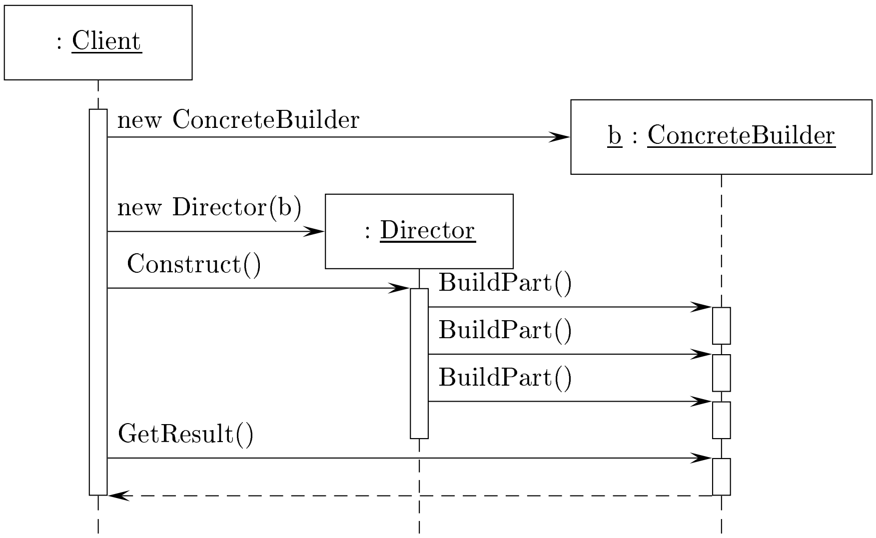


Elemek:

- Builder: a részek létrehozásának felületét adja meg.
- ConcreteBuilder: létrehozza a részeket, és ezekből felépíti a terméket a felületet megvalósítva; megadja a termék lekérdezésének műveletét.
- Director: létrehozza a terméket a Builder felületét felhasználva.
- Product: a létrehozandó összetett termék.

Együttműködés:

A kliens létrehozza a megfelelő konkrét építő objektumot, ezzel konfigurálja a direktort. A direktort utasítja a termék elkészítésére. A direktor értesíti az építőt, amikor egy újabb részt kell előállítani. Az építő ennek alapján új elemekkel bővíti a terméket, amit a végén a kliens lekérdez.



20.1. Példa kód

```
class Builder
{
public:
    virtual void BuildPart()  { }
protected:
    Builder();
};

class Director
{
public:
    Director(Builder *b)    { builder = b; }
    void Construct();
private:
    Builder *builder;
};

class Product
{
    ...
};
```

```
class ConcreteBuilder : public Builder
{
public:
    ConcreteBuilder();
    void BuildPart();
    Product *GetResult()    { return(product); }
private:
    Product *product;
};
```

20.2. Esettanulmány

Egy lehetséges alkalmazás az RTF szöveg konvertálása, amelynek elkészítését most nem részletezzük.

21. Gyártó művelet

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

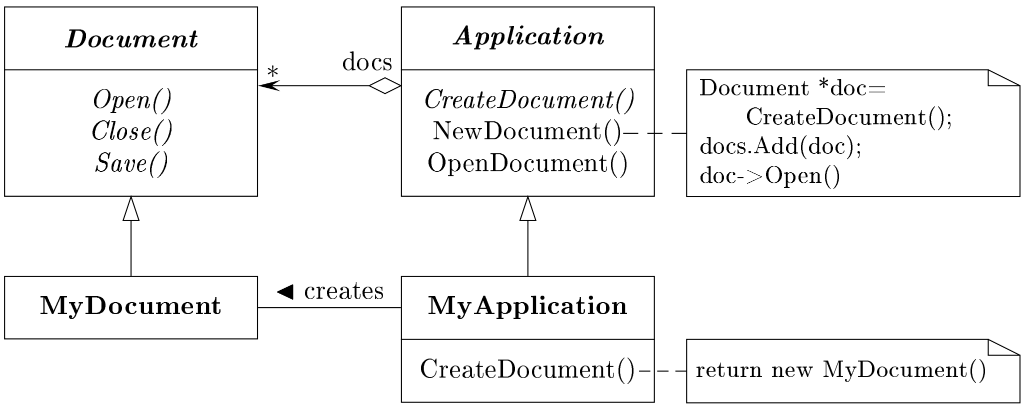
Cél: Egy objektum létrehozási felületének meghatározása úgy, hogy az alosztályok döntsék el, hogy az objektum melyik osztály példánya legyen. A gyártó művelet lehetővé teszi, hogy egy osztály a példányosítást elhalassza az alosztályhoz.

Más nevek: virtual constructor

Motiváció: A már tárgyalt alkalmazás-dokumentum keretrendszerekben merül fel annak a problémája, hogy az alkalmazásnak létre kell hoznia egy megfelelő dokumentumot. Ugyanakkor mindkét osztály absztrakt, ezért csak a konkrét alkalmazás tudja megmondani a tényleges osztályt, amelynek egy példányát elő kell állítani. (Például rajzoló program esetén egy rajzot, szövegszerkesztő esetén egy szöveget, stb..)

Miután a megfelelő dokumentum alosztály az alkalmazástól függ, az absztrakt alkalmazás osztály csak azt tudja, hogy egy dokumentumot létre kell hozni, de annak fajtáját nem ismerheti. Azaz példányosítani kellene egy osztályt, de csak absztrakt osztályról tudunk, amit nem lehet példányosítani.

A gyártó művelet adja a megoldást. Magába zárja annak ismeretét, hogy milyen dokumentum alosztályt kell példányosítani, és kiemeli ezt az ismeretet az alkalmazás osztályából. Az alkalmazás alosztályai átdefiniálják a CreateDocument absztrakt eljárást, és így a megfelelő példányt adják vissza. A CreateDocument műveletet nevezzük gyártó műveletnek.

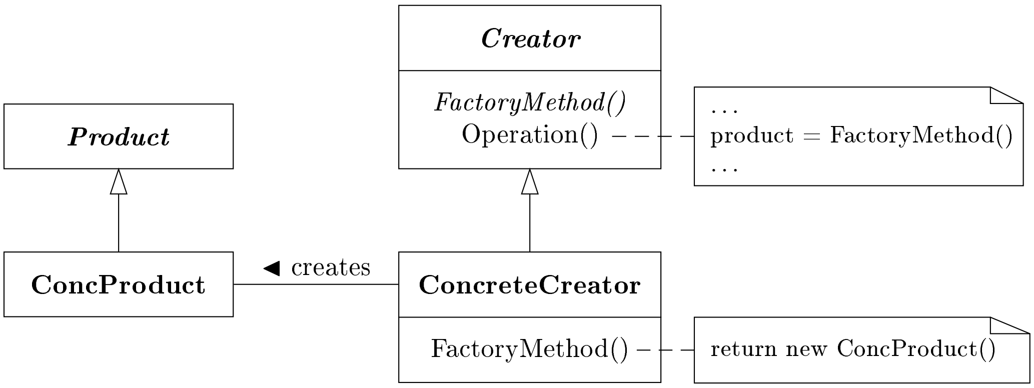


Felhasználhatóság: A gyártó művelet minta használható az alábbi esetekben:

- 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:



Elemek:

- Product: a gyártó művelet által létrehozott objektumok felületét adja meg.
- ConcProduct: megvalósítja a Product felületét.
- Creator: deklarálja a gyártó műveletet, amely egy termék objektumot ad vissza, amit itt felhasználhatunk; megadhatja a gyár művelet alapértelmezett működését is.
- ConcreteCreator: átdefiniálja a gyártó műveletet, és így egy konkrét termék egy példányát adja meg.

Együttműködés: A Creator a konkrét leszármazottaira hagyatkozik a gyártó művelet megvalósításában, így éri el, hogy megfelelő terméket példányosítson.

21.1. Példa kód

Az első változatban a Creator egy absztrakt osztály, amelyben nincs alapértelmezett működés.

```
class Product
{
    ...
};

class Creator
{
public:
    virtual Product *FactoryMethod() = 0;
    void Operation();
protected:
```

```
    Creator();
private:
    Product *prod;
};

class ConcreteProduct : public Product
{
    ...
};

class ConcreteCreator : public Creator
{
public:
    ConcreteCreator();
    ConcreteProduct *FactoryMethod();
};

ConcreteProduct *ConcreteCreator::FactoryMethod()
{
    ConcreteProduct *p = new ConcreteProduct();
    ...
    return(p);
}
```

A második változatban a Creator konkrét osztály, amelyben megadunk alapértelmezett viselkedést, továbbá a létrehozott objektum egy paramétertől függ (paraméterezett gyártó művelet).

```
class Product;

class MyProduct : public Product;

class YourProduct : public Product;

class Creator
{
public:
    Creator();
    virtual Product *FactoryMethod(ProductID id);
};

Product *Creator::FactoryMethod(ProductID id)
{
    if ( id == MINE )  return(new MyProduct());
    if ( id == YOURS ) return(new YourProduct());
    ...
    return(0);
}

class TheirProduct : public Product;
```

```
class MyCreator : public Creator
{
public:
    MyCreator();
    Product *FactoryMethod(ProductID id);
};

Product *MyCreator::FactoryMethod(ProductID id)
{
    if ( id == MINE )   return(new YourProduct());
    if ( id == YOURS )  return(new MyProduct());    // csere
    if ( id == THEIRS ) return(new TheirProduct());
    return(Creator::FactoryMethod(id));
}
```

21.2. Esettanulmány

Egy lehetséges alkalmazás a labirintusok létrehozása.

```
class MazeGame
{
public:
    Maze *CreateMaze();
        // factory methods:
    virtual Maze* MakeMaze() const { return(new Maze()); }
    virtual Room* MakeRoom(int n) const
                               { return(new Room(n)); }
    virtual Wall* MakeWall() const { return(new Wall()); }
    virtual Door* MakeDoor(Room *r1, Room *r2) const
                               { return(new Door(r1, r2)); }
};

Maze *MazeGame::CreateMaze()
{
    Maze *m = MakeMaze();
    Room *r1 = MakeRoom(1);
    Room *r2 = MakeRoom(2);
    Door *d = MakeDoor(r1, r2);
    m->AddRoom(r1); m->AddRoom(r2);
    r1->SetSide(North, MakeWall());
    r1->SetSide(East, d);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());
    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, d);
    return(m);
}
```

Eltérő labirintusok esetén a MazeGame osztályból kell új osztályokat származtatni a gyár műveletek értelemszerű változtatásával.

```
class BombedMazeGame : public MazeGame
{
public:
    BombedMazeGame();
    Wall *MakeWall() const { return(new BombedWall()); }
    Room *MakeRoom(int n) const { return(new RoomWithBomb(n)); }
};

class EnchantedMazeGame : public MazeGame
{
public:
    EnchantedMazeGame();
    Room *MakeRoom(int n) const
                { return(new EnchantedRoom(n, CastSpell())); }
    Door *MakeDoor(Room *r1, Room *r2) const
                { return(new DoorNeedingSpell(r1, r2)); }
protected:
    Spell *CastSpell() const;
};
```

22. Prototípus

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

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

Más nevek: –

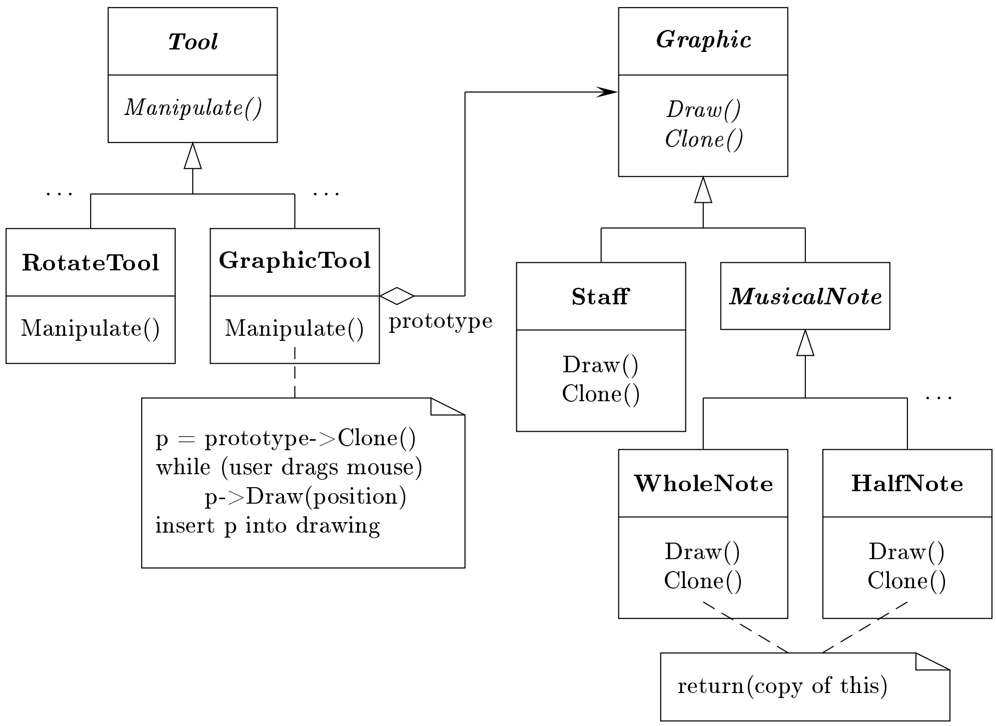
Motiváció: Tegyük fel, hogy kotta szerkesztő programot szeretnénk készíteni. Ezt megtehetjük például egy általános célú grafikai szerkesztő keretrendszer testre szabásával, ha kiegészítjük megfelelő kotta elemeket ábrázoló grafikus objektumokkal. A rendszer tartalmazhat egy elemet (palettát), amelynek segítségével ezeket az objektumokat hozzávehetjük a dokumentumhoz, kiválaszthatjuk, mozgathatjuk azokat.

Tegyük fel, hogy a keretrendszer tartalmaz egy absztrakt *Graphic* osztályt, amelyből származtathatjuk a kotta elemeket. Ezen kívül rendelkezik egy absztrakt *Tool* osztállyal is, amelyből a paletta eszközeit hozhatjuk létre. Adott egy előre definiált *GraphicTool* osztály is, amelynek segítségével grafikus objektumok példányait hozhatjuk létre és adhatjuk a dokumentumhoz.

Ekkor a *GraphicTool* osztállyal kapcsolatban a következő probléma merül fel. A kotta elemeihez készített osztályok az alkalmazásra nézve speciálisak, de a *GraphicTool* osztály a keretrendszerhez tartozik. Ha minden lehetséges objektumhoz egy ebből származtatott alosztályt rendelnénk, akkor túl sok osztály jönne létre, amelyek között csak az a különbség, hogy eltérő példányokat hoznak létre.

Objektumok kompozíciója egy rugalmas alternatíva erre az esetre. A kérdés, hogy miként használhatja ezt a keretrendszer arra, hogy a *GraphicTool* példányait a létrehozandó objektumok osztályával paraméterezzük.

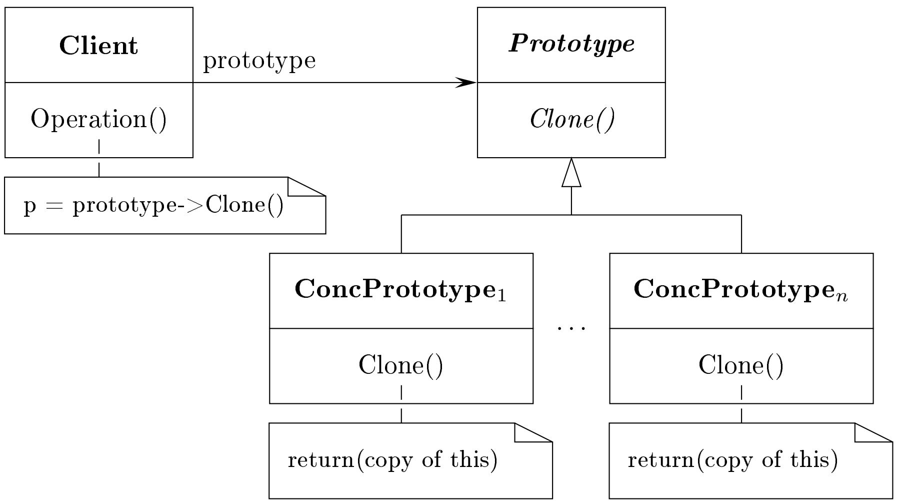
A megoldás, hogy a *GraphicTool* osztály egy *Graphic* objektumot hoz létre a megfelelő származtatott osztály másolásával, klónozásával. Ezt nevezzük prototípusnak, és ezzel paraméterezzük a *GraphicTool*-t. Ha mindegyik származtatott osztály rendelkezik egy klónozó művelettel, akkor a *GraphicTool* bármelyiket klónozni tudja.



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:



Elemek:

- Prototype: megadja a klónozás felületét.
- ConcPrototype$_i$: megvalósítja a klónozás műveletét.
- Client: létrehoz egy új objektumot a prototípus klónozásával.

Együttműködés: A kliens kiad egy klónozási igényt a prototípusnak.

22.1. Példa kód

Az implementáció során figyelni kell a klónozás műveletében szereplő másolásra (Copy). Az alapértelmezett másolás (copy constructor), nem minden esetben megfelelő, ugyanis az csak az adattagok értékeit másolja. Mutatók esetében ez nem biztos, hogy megfelelő (mindkét példány mutatói ugyanarra az objektumra mutatnak). Rendszerint úgynevezett mély másolásra (deep copy) van szükség. Ezt vagy egy saját másoló művelettel valósítjuk meg, vagy egy inicializáló művelet segítségével.

```
class Prototype
{
public:
    virtual Prototype *Clone() const = 0;
protected:
    Prototype();
};

class Client
{
public:
    ...
    void Operation();
private:
    Prototype *prototype;
};

void Client::Operation()
{
    p = prototype->Clone();
    ...
}

class ConcPrototype_i : public Prototype
{
public:
    ConcPrototype_i();
    Prototype *Clone() const;
protected:
    void Initialize(...);
};
```

```
Prototype *ConcPrototype_i::Clone() const
{
    ConcPrototype_i *p = new ConcPrototype_i();
    p->Initialize(...);
    return(p);
}
```

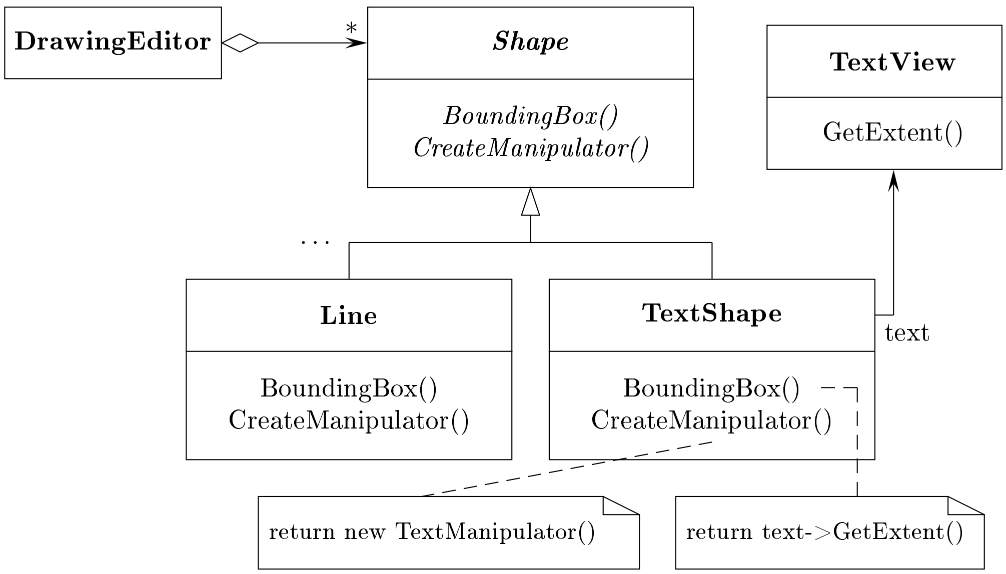
23. Átalakító

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

Motiváció: Egy grafikus szerkesztő programban különböző alakzatokból állítjuk össze a képet. Ezeket egy közös *Shape* osztályból származtatjuk. Vonalak, síkidomok esetén ez nem okoz gondot, de szövegek esetén célszerű lenne egy már meglévő elemet használni, pl. *TextView*. (Egy szöveg jóval bonyolultabb: szerkesztés, stb..) Ugyanakkor a *TextView* és a *Shape* osztályok felülete nem kompatibilis. Ezért bevezetünk egy *TextShape* osztályt, amelyik megfelelően átalakítja a *TextView* felületét (BoundingBox), illetve kiegészíti azt a megfelelő művelettel (CreateManipulator).



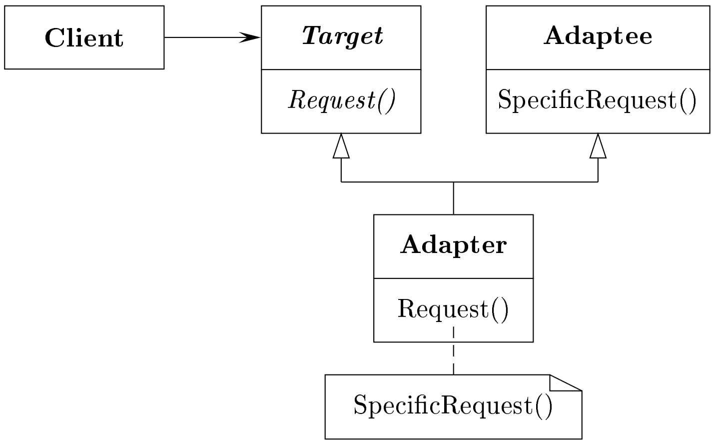
Felhasználhatóság: Az átalakító minta használható az alábbi esetekben:

- 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.

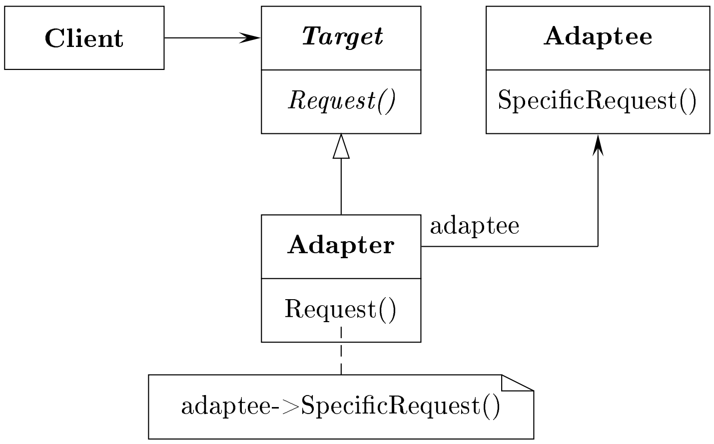
- (*csak objektum átalakítás esetén*) Számos osztályt kell használnunk, de nem célszerű ezek felületét származtatással átalakítanunk. Ekkor egy átalakító objektum megváltoztatja a szülő osztály felületét.

Szerkezet:

Osztály átalakító:



Objektum átalakító:



Elemek:

- Target: megadja az alkalmazás (kliens) által használt felületet.
- Client: Target felületű objektumokkal működik együtt.
- Adaptee: egy létező, átalakítandó felületet ad meg.
- Adapter: Adaptee felületét Target felületté alakítja.

Együttműködés: A kliensek az *Adapter* példányain végeznek műveleteket. A példányok az *Adaptee* műveleteit használják fel az igények kielégítésére.

23.1. Esettanulmány

Egy lehetséges alkalmazás a bemutatott grafikus szerkesztő. Először az osztály átalakítót mutatjuk meg, amelyben többszörös öröklődést használunk.

```
class Shape
{
public:
    Shape();
    virtual void BoundingBox(Point& bl, Point& tr) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView
{
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};

class TextShape : public Shape, private TextView
{
public:
    TextShape();
    virtual void BoundingBox(Point& bl, Point& tr) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};

void TextShape::BoundingBox(Point& bl, Point& tr) const
{
    int bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bl = Point(bottom, left);
    tr = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty() const
{
    return(TextView::IsEmpty());
}

Manipulator* TextShape::CreateManipulator() const
{
    return(new TextManipulator(this));
}
```

Az objektum átalakító egy mutatóval hivatkozik az átalakítandó *TextView* objektumra.

```
class TextShape : public Shape
{
public:
    TextShape(TextView *t)     { _text = t; }
    virtual void BoundingBox(Point& bl, Point& tr) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};

void TextShape::BoundingBox(Point& bl, Point& tr) const
{
    int bottom, left, width, height;
    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);
    bl = Point(bottom, left);
    tr = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty() const
{
    return(_text->IsEmpty());
}

Manipulator* TextShape::CreateManipulator() const
{
    return(new TextManipulator(this));
}
```

Az objektum átalakító rugalmasabb, mint az osztály átalakító, hiszen *TextView*-ból származtatott alosztályokkal is paraméterezhető.

24. Díszítő

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

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

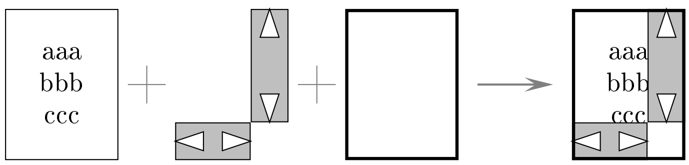
Más nevek: wrapper

Motiváció: Vannak olyan esetek, amikor egy egész osztály helyett egyes objektumokon akarunk új műveletet értelmezni. Egy grafikus felhasználói felület eszközeit kiegészíthetjük például kerettel, vagy a görgetés műveletével.

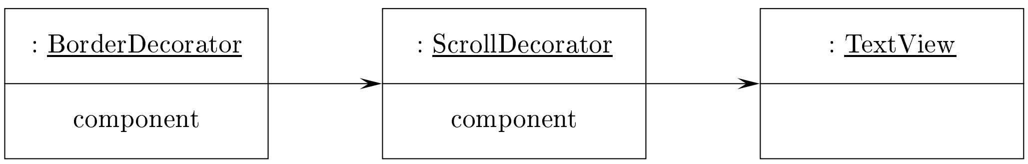
Egy lehetséges megoldás az öröklődés, azonban ez nem elég rugalmas, mert a származtatott osztály minden egyes objektuma rendelkezik az új tulajdonsággal. Például kerettel az előző esetben. Ekkor az, hogy valami rendelkezik-e kerettel statikus (fordítási idő), így egy kliens nem tudja ezt vezérelni.

Egy rugalmasabb megoldás, ha a komponenst beágyazzuk egy másik objektumba, amelyik hozzáadja a tulajdonságot (keretet). A beágyazó objektumot nevezzük díszítőnek. A díszítő felülete megfelel a beágyazott objektuménak, ezért a jelenlétét nem észlelik a komponens felhasználói, azaz átlátszó. A díszítő az igényeket a komponensnek továbbítja, és ezen kívül további műveleteket hajthat végre (keret rajzolása) a továbbítás előtt vagy után. Az átláthatóság miatt a díszítőket tetszőleges mélységben alkalmazhatjuk, így a bővíthetőség nem korlátozott.

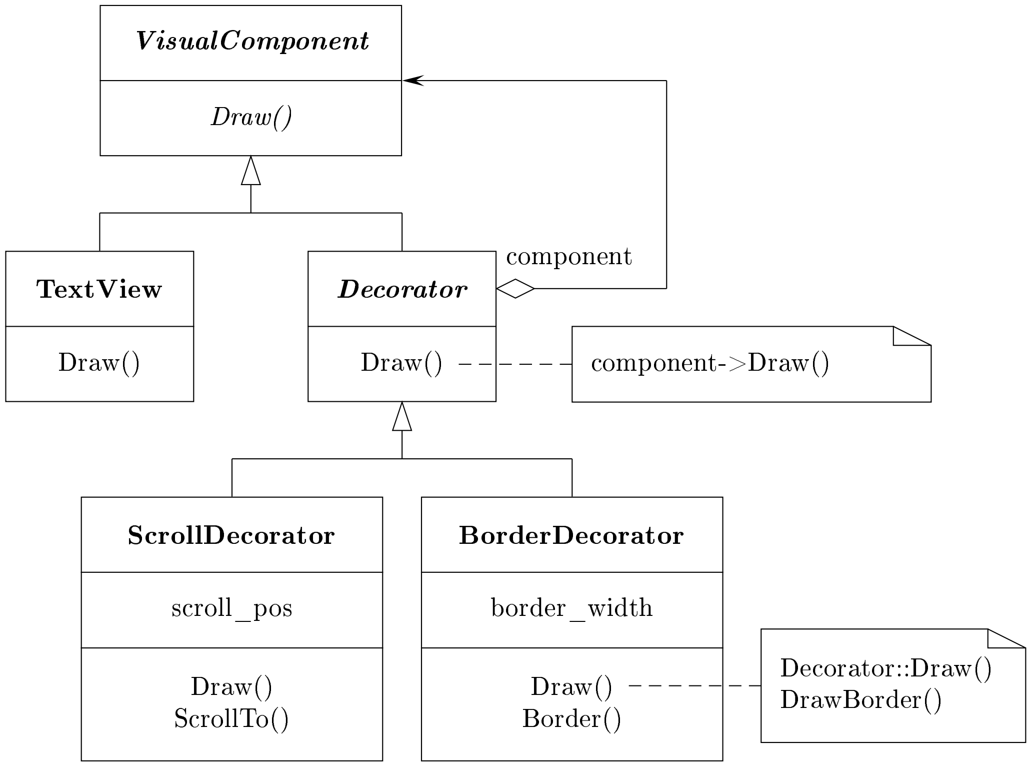
Tegyük fel, hogy egy TextView objektumot szeretnénk görgetősávval és kerettel ellátni.



Ha díszítőkkel valósítjuk ezt meg, akkor az eddigiek alapján a következő objektumdiagramhoz jutunk.



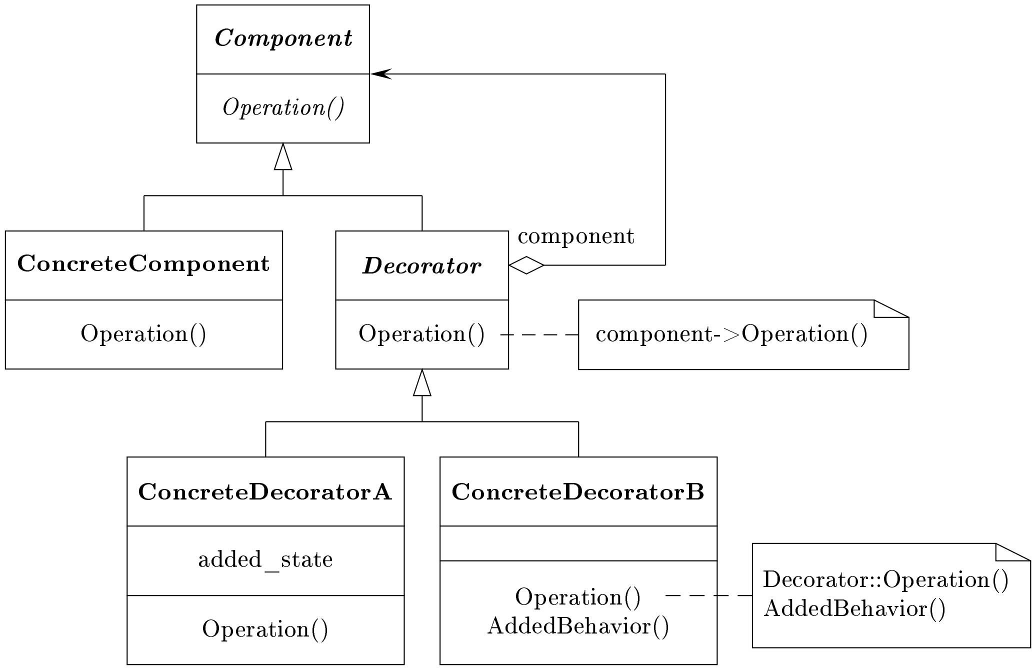
A példában vizuális elemekkel foglalkoztunk. A következő osztálydiagram felel meg a leírt esetnek. Ahol egy komponens szerepelhet, ott lehet díszítő is, ezért a kliensek nem tudják megkülönböztetni a díszített és a sima objektumokat.



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:



Elemek:

- Component: megadja a dinamikusan bővíthető funkcionalitással rendelkező objektumok felületét.
- ConcreteComponent: egy bővíthető objektumot határoz meg.
- Decorator: hivatkozik egy komponens objektumra és annak megfelelő felületet definiál.
- ConcreteDecorator: kibővíti a komponens funkcionalitását.

Együttműködés: A díszítő a komponensnek küldi az üzenetet. A küldés előtt vagy után végrehajthat további műveleteket.

24.1. Esettanulmány

Egy lehetséges alkalmazás bemutatott grafikus felület.

```
const VisualComponent
{
public:
    virtual void Draw() = 0;
    virtual void Resize() = 0;
    ...
};
```

```
const Decorator : public VisualComponent
{
public:
    Decorator(VisualComponent *v) { component = v; }
    virtual void Draw()           { component->Draw(); }
    virtual void Resize()         { component->Resize(); }
private:
    VisualComponent *component;
};

const BorderDecorator : public Decorator
{
public:
    BorderDecorator(VisualComponent *v, int w);
    void Draw();
private:
    DrawBorder(int w);
private:
    int width;
};

BorderDecorator::BorderDecorator(VisualComponent *v, int w)
                         : Decorator(v)
{
    width = w;
}

void BorderDecorator::Draw()
{
    Decorator::Draw();
    DrawBorder(width);
}

const ScrollDecorator : public Decorator
{
public:
    ScrollDecorator(VisualComponent *v);
    void Draw();
private:
    ScrollTo(int pos);
private:
    int pos;
};

const TextView : public VisualComponent
{
    ...
};
```

```
Window *w = new Window;
TextView *t = new TextView;

// sima textview:
w->SetContents(t);

// keretes, görgethető textview:
w->SetContents(new BorderDecorator(new ScrollDecorator(t), 2));
```

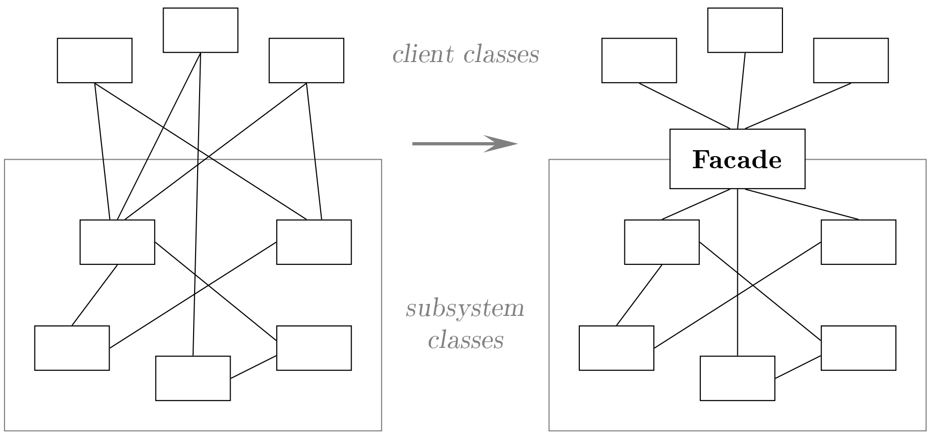
25. Arculat

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.

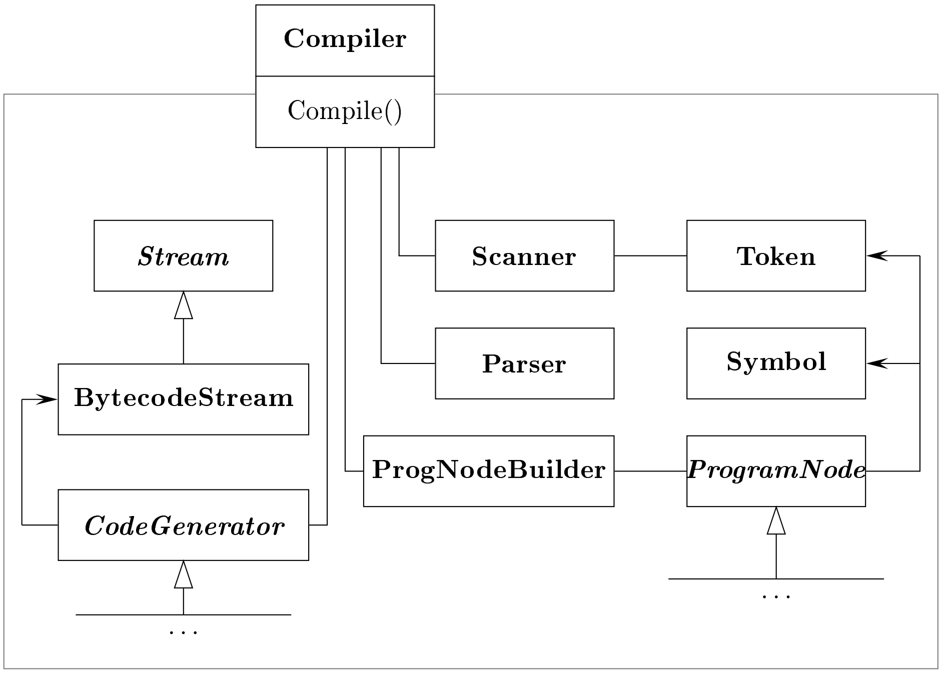
Más nevek: –

Motiváció: Egy rendszer felbontása alrendszerekre csökkenti a bonyolultságot. Egy általános tervezési cél az alrendszerek közötti kommunikáció és függőségek minimalizálása. Egy mód ennek elérésére egy *arculat* objektum bevezetése, ami egyetlen, egyszerűsített felületet biztosít az alrendszer általánosabb lehetőségeihez.



Egy lehetséges példa egy olyan programozási környezet, amelyben az alkalmazások elérhetik a fordító alrendszert. A fordító alrendszer tartalmazza például a következő osztályokat: Scanner, Parser, ProgramNode, BytecodeStream és ProgramNodeBuilder, amelyek megvalósítják a fordítót. Néhány speciális alkalmazás direkt módon érheti el ezeket az osztályokat. A legtöbb esetben a fordító használóit nem érdeklik ezek részletesen (például a kódgenerálás), csak le akarnak fordítani egy kódot. Ezek számára a fordító alrendszer alacsony szintű felületei csak bonyolítják a megoldandó feladatot.

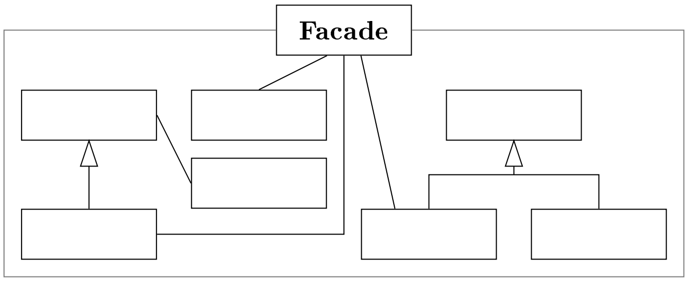
A magasabb szintű felület, amely elrejti a kliensek elől a fenti osztályokat, az alrendszerben található Compiler osztály biztosítja. Ez egységes felületet ad a fordító használatához. A Compiler osztály az arculat. A klienseknek egy egységes, egyszerű felületet ad a fordítóhoz. Összekapcsolja a fordítást megvalósító osztályokat anélkül, hogy elrejtené azokat.



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:



Elemek:

- Facade: tudja, hogy az alrendszer melyik osztálya felelős egy igény feldolgozásáért; továbbítja a kliens igényeit az alrendszer megfelelő objektumának.
- alrendszer osztályok: megvalósítják az alrendszer funkcionalitását; kezelik az arculat által hozzájuk rendelt feladatokat; nem ismerik az arculatot, azaz nem hivatkoznak arra.

Együttműködés: A kliensek az arculaton keresztül kommunikálnak az alrendszerrel. Az arculat továbbítja az igényeket a megfelelő alrendszer objektumnak. Ennek során az arculat a felületet az alrendszer objektumának megfelelőre alakíthatja. Az arculatot használó kliensek nem érik el direkt módon az alrendszer elemeit.

25.1. Esettanulmány

Vizsgáljuk meg a fordító alrendszer megvalósításának vázlatát!

A Scanner osztály egy karaktersorozatból előállítja a tokenek sorozatát egyesével.

```
class Scanner
{
public:
    Scanner(istream&);
    virtual ~Scanner();
    virtual Token& Scan();
private:
    istream& inputstream;
};
```

A Parser osztály a ProgramNodeBuilder felhasználásával felépíti a szintaxis fát a tokenekből. Ennek során a két osztály az építő mintának megfelelően működik együtt.

```
class Parser
{
public:
    Parser();
    virtual ~Parser();
    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};

class ProgramNodeBuilder
{
public:
    ProgramNodeBuilder();
    virtual ProgramNode* NewVariable(const char* name) const;
    virtual ProgramNode* NewAssignment(ProgramNode* var,
                                       ProgramNode* expr) const;
    ...
    ProgramNode* GetRootNode();
private:
    ProgramNode* node;
};
```

A szintaxis fa a ProgramNode osztályból származtatott osztályokból (utasítás, kifejezés, ...) épül fel. A ProgramNode hierarchiája az összetétel minta egy esete. Ebben a ProgramNode osztály adja meg az egységes felületet.

```
class ProgramNode
{
public:
    ...
    virtual void GetSourcePosition(int &line, int& col);
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

A Traverse művelet paramétere egy CodeGenerator objektum, ami előállítja a megfelelő gépi kódot az adott csúcsból. Itt a CodeGenerator osztály egy látogató. Ebből kell az egyes gépeknek megfelelő generátort származtatni.

```
class CodeGenerator
{
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    ...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& output;
};
```

A bejárás (Traverse) művelet egy kifejezés esetén például a következő lehet.

```
void ExpressionNode::Traverse(CodeGenerator& cg)
{
    cg.Visit(this);
    ListIterator<ProgramNode*> it(children);
    for ( it.First(); !it.IsDone(); it.Next() )
        it.CurrentItem()->Traverse(cg);
}
```

Az eddigi osztályok a fordító alrendszer elemei. Ezek után bevezethetjük az arculatot megadó Compiler osztályt, amely a következő lehet.

```
class Compiler
{
public:
    Compiler();
    virtual void Compile(istream&, BytecodeStream&);
};
```

```
void Compiler::Compile(istream& inp, BytecodeStream& out)
{
    Scanner scanner(inp);
    PorgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(out);
    ProgramNode* parsetree = builder.GetRootNode();
    parsetree->Traverse(generator);
}
```

26. Könnyűsúlyú

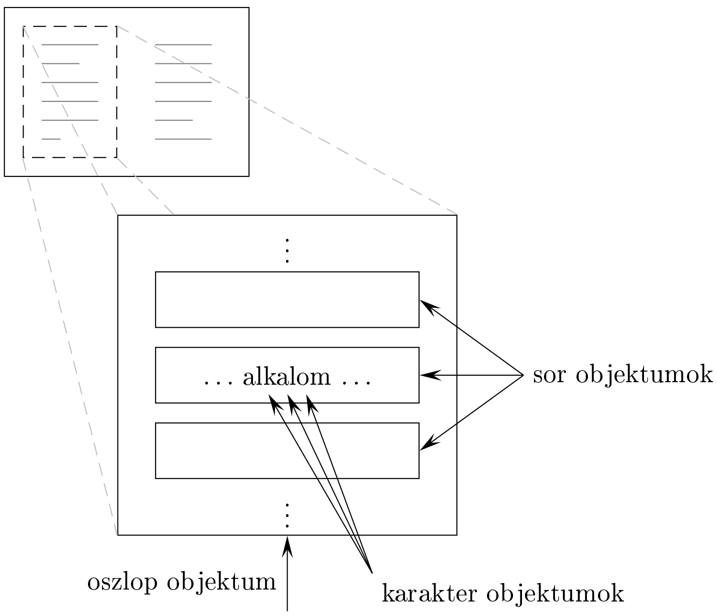
Név, osztály: könnyűsúlyú (flyweight), szerkezeti (structural)

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: –

Motiváció: Vannak olyan alkalmazások, amelyekben hasznos lenne az objektumokat egyesével kezelnünk a teljes tervezés során, de egy naív implementáció túlságosan költséges lenne.

Például egy dokumentum szerkesztő tartalmaz szöveg formázó és szerkesztő lehetőségeket, amelyeket bizonyos mértékig modularizálunk. Objektumelvű szerkesztők objektumokkal ábrázolják a beágyazott elemeket (táblázatok, ábrák). Ugyanakkor nem ábrázolnak minden egyes karaktert külön objektumként, noha az nagyfokú rugalmasságot tenne lehetővé az alkalmazás legfinomabb szintjein. Ekkor ugyanis a karaktereket és a beágyazott objektumokat egységesen kezelhetnénk a megjelenítés és formázás során; az alkalmazást könnyű lenne új karakter halmazokkal kibővíteni anélkül, hogy a funkcionalitást megzavarnánk. Az alkalmazás objektum szerkezete ekkor megfelelne az objektum fizikai szerkezetének.

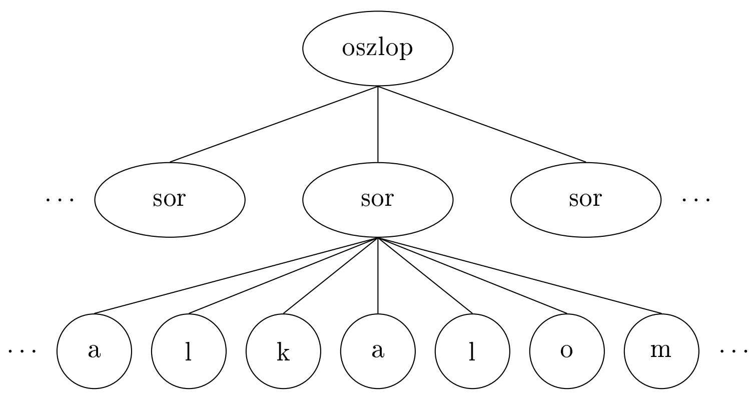


Az ilyen típusú megközelítés hátránya a költség. Még nem túl nagy méretű dokumentumok esetén is százezres nagyságrendű karakter objektumra lenne szükségünk, ami túl nagy memóriát igényelne, és ezért elfogadhatatlan futási igényekkel rendelkezne. A könnyűsúlyú minta megadja miként oszthatunk meg objektumokat úgy, hogy a részletes információkat is tároljuk elfogadható módon.

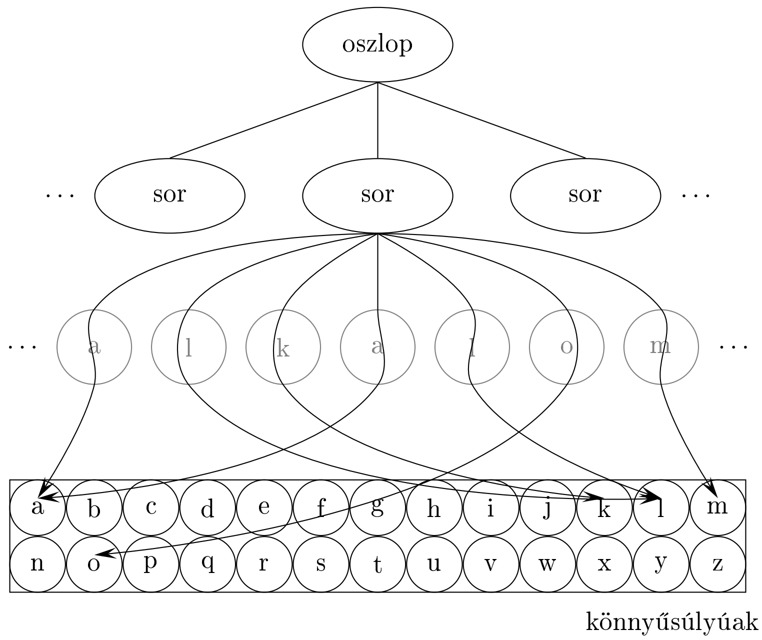
Egy könnyűsúlyú egy osztott objektum, amelyet több környezet is használhat egyidejűleg. A könnyűsúlyú minden környezet számára egy független objektumként jelenik meg, nem különböztethető meg az osztatlan objektum példányától. A könnyűsúlyúak nem tehetnek fel semmit sem a felhasználó környezetről. Az alapötlet a belső és külső állapotok közötti megkülönböztetés. A belső állapotot a könnyűsúlyúban tároljuk, és ez olyan adatokat tartalmaz, amelyek függetlenek a környezettől. A külső állapot a környezettől függ és annak megfelelően változik, ezért nem osztható meg. Szükség esetén a klienseknek kell a külső állapot jellemezőit átadniuk a könnyűsúlyúnak.

Könnyűsúlyúakkal olyan elemeket modellezünk, amelyek túl sok erőforrást igényelnének, ha egyedi objektumokat használnánk a reprezentációban. Például a dokumentum szerkesztő létrehozhat egy könnyűsúlyút minden egyes ábécébeli jelhez. Minden könnyűsúlyú tárolja a karakter kódját, de a dokumentumon belüli helyzetét és szedési stílusát a jel előfordulásánál érvényben levő szöveg formázó parancsok határozzák meg. A karakter kód a belső állapot, a többi információ a külső állapot része.

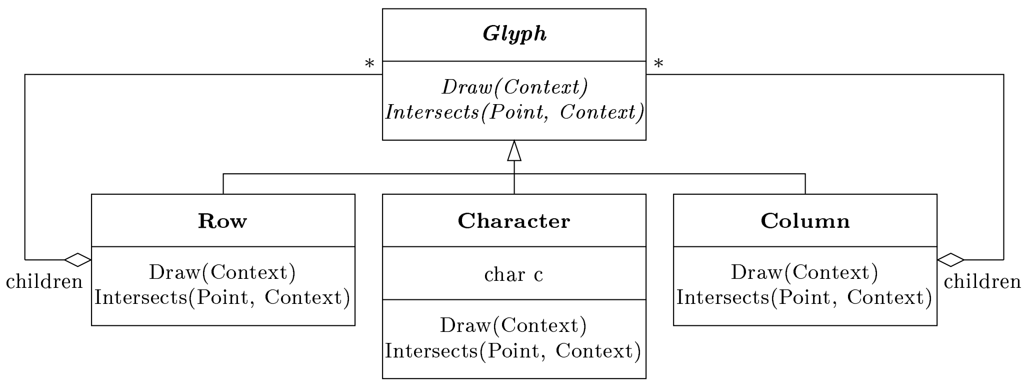
A logikai szerkezetben egy objektum tartozik minden egyes karakterhez. Az előző esetben ez a következőt jelenti.



Ugyanakkor fizikailag egyetlen osztott könnyűsúlyú objektum tartozik egy jelhez, amelyik különböző környezetekben jelenik meg a dokumentumban. Egy karakter minden egyes előfordulása ugyanarra a könnyűsúlyú objektumpéldányra hivatkozik.



Vizsgáljuk meg az osztálydiagramot! A Glyph absztrakt osztály adja meg a grafikus objektumok felületét. Ezek között szerepelhet könnyűsúlyú is. A külső állapotokat leíró adatok a műveletek paraméterei.



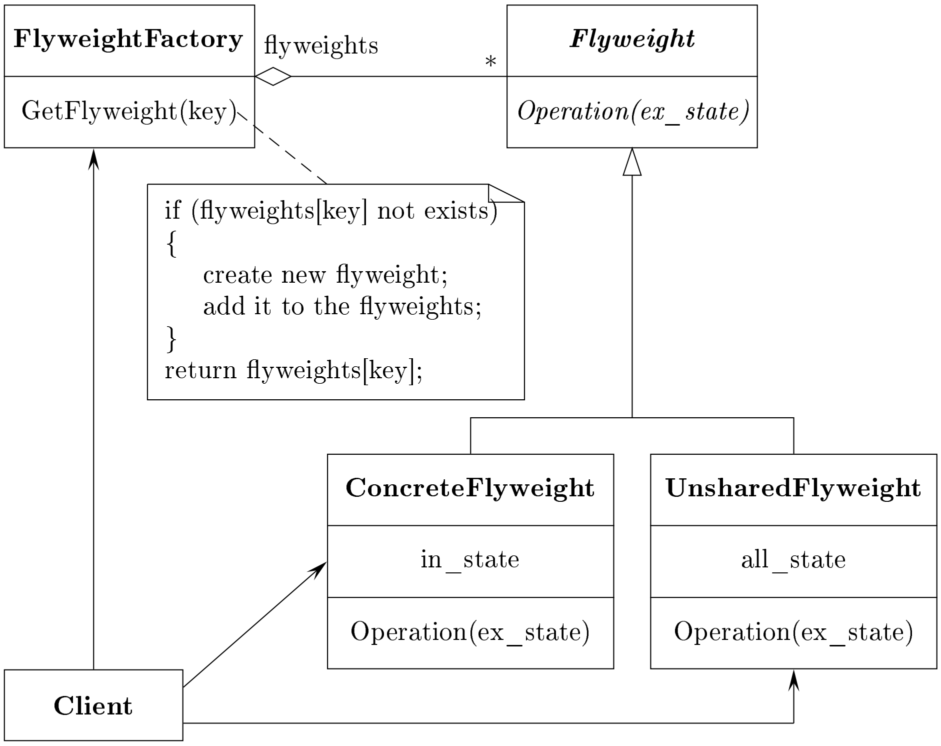
Az a jelet ábrázoló könnyűsúlyú csak a karakter kódot tárolja, a helyzetet és a fontot nem. A kliensek adják meg a környezetfüggő információt, amelynek alapján a könnyűsúlyú kirajzolja önmagát. Például egy sor tudja, hogy a gyerekeinek hova kell kirajzolniuk magukat, hogy vízszintesen illeszkedjenek. Ennek megfelelően minden egyes gyerekének megadja a helyzetét a rajzoláshoz.

A különböző karakterek száma jóval kevesebb, mint a dokumentum jeleinek száma (hossz), így az objektumok száma is jóval kevesebb, mint amit a naív megközelítés eredményezne. Ha ugyanazt a fontot és színt használjuk, akkor egy szöveges dokumentum esetén körölbelül száz objektum elegendő függetlenül a dokumentum hosszától. Ugyanakkor egy átlagos dokumentumban tíznél több font-szín kombinációt nem használunk, ezért ez a szám nem nő elfogadhatatlan mértékben a gyakorlatban.

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:



Elemek:

- Flyweight: megadja az a felületet, amelyen keresztül a könnyűsúlyúak megkaphatják a külső állapotot, és reagálnak arra.
- ConcreteFlyweight (Character): megvalósítja a Flyweight felületét, tárolja a belső állapotot, ha van. Ez mindenképpen megosztható, a tárolt belső állapot független a ConcreteFlyweight objektum környezetétől.
- UnsharedFlyweight (Row, Column): nem minden Flyweight osztályból származtatott osztálynak kell megosztottnak lennie. A felület engedélyezi a megosztást, nem teszi kötelezővé. Az UnsharedFlyweight objektumok gyerekei rendszerint ConcreteFlyweight objektumok a szerkezet valamilyen szintjén.

- FlyweightFactory: létrehozza és kezeli a könnyűsúlyú objektumokat; biztosítja, hogy azok megfelelően legyenek megosztva. Ha a kliens igényel egy könnyűsúlyú objektumot, akkor megad egy létező példányt, illetve létrehoz egyet, ha nincs ilyen.
- Client: hivatkozik a könnyűsúlyú objektumokra; meghatározza, tárolja a külső állapotokat.

Együttműködés:

- A könnyűsúlyú működéséhez szükséges állapotot belső és külső állapotokra kell osztani. A belső állpotot a ConcreteFlyweight objektumok tárolják, a külső állapotokért a kliens felelős. A külső állapotot a kliens adja meg a könnyűsúlyúnak a művelet hívásakor.
- A kliensek közvetlenül nem hozhatnak létre ConcreteFlyweight objektumokat, azokat csak a FlyweightFactory objektumon keresztül érhetik el. Ez garantálja a megfelelő megosztást.

26.1. Esettanulmány

A motiváció során ismertetett problémára térünk vissza. Először megadjuk a Glyph osztályt, amely lényegében egy összetétel, megfelel az összetétel mintában megismert Component osztálynak. Azt ki kell egészíteni a megjelenítéshez szükséges elemekkel. A GlyphContext objektumokból lehet kinyerni az aktuális fontot.

```
class Glyph
{
public:
    virtual ~Glyph();
    virtual void Draw(Window*, GlyphContext&);
    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);
    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);
    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

Az ebből származtatott Character osztály csak a karakter kódját tárolja.

```
class Character : public Glyph
{
public:
    Character(char);
    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

Annak érdekében, hogy elkerüljük a font attribútum tárolását minden egyes glyph esetén, felveszünk egy új GlyphContext osztályt, amelynek objektumaiban tároljuk ezt a külső információt. Ez az osztály adja meg a teljes megfeleltetést egy glyph és a különböző környezetekben hozzá tartozó font között. Ha szükséges egy művelethez a font, akkor egy ilyen objektumot adunk meg paraméterként. Ebből kapjuk meg a fontot a glyph szerkezetben elfoglalt helynek megfelelően. Ezért egy Glyph objektum gyerekeinek bejárása, illetve manupulációja során a GlyphContext karbantartására figyelni kell.

```
class GlyphContext
{
public:
    GlyphContext();
    virtual ~GlyphContext();
    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);
    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int      _index;
    BTree*   _fonts;
};
```

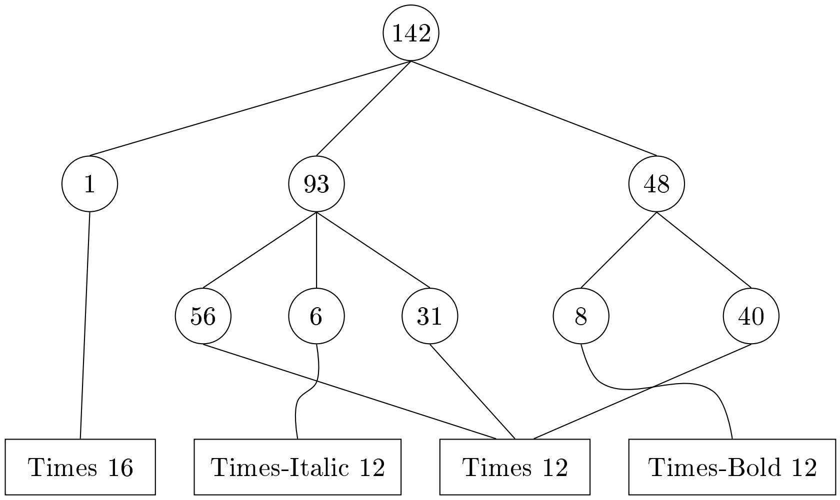
Az _index adattag adja meg, hogy hol tartunk a szerkezet bejárásában. A Next művelettel mehetünk a következő elemre a bejárás során. Minden Glyph-ből származtatott gyerekekkel rendelkező osztályban meg kell valósítanunk a bejárás Next műveletét a GlyphContext::Next() segítségével.

A GetFont művelet az index segítségével keresi ki a BTree szerkezetből a glyph-font megfeleltetést. A fa minden egyes levele tartalmazza, hogy az milyen hosszúságú sorozatra definiál fontot, és hivatkoznak egy fontra. A belső csúcsok részekre bontják a sorozatot, minden gyermek egy részsorozatnak felel meg.

Tekintsük példaként a következő szöveg részletet. (A táblazat első, szürke oszlopa a sor első jelének szövegbeli indexét adja meg.)

1	Objektumelvű ...
51	esetén *elvárt* tulajdonság ...
91	egy **iterátor** használható ...

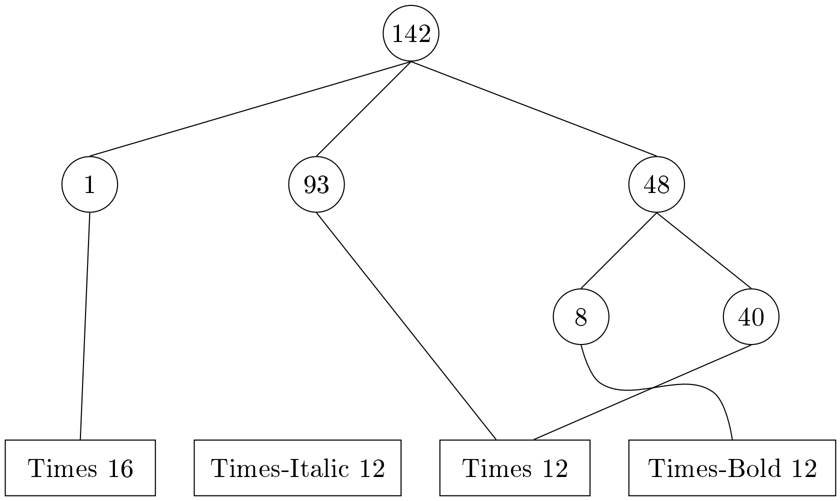
Egy ennek megfelelő BTree szerkezet lehet a következő.



Tegyük fel, hogy az aktuális pozíció az 57-es index! Hajtsuk végre a következő utasításokat!

```
GlyphContext    gc;
Font*           times12 = new Font("Times-Roman-12");
gc.SetFont(times12, 6);
```

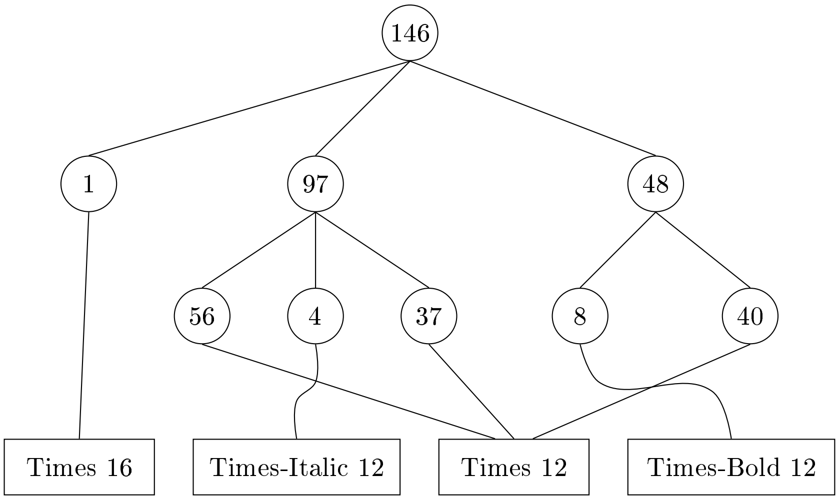
Ekkor az új fa szerkezet:



Térjünk vissza az 57. pozícióra, és szúrjuk be a *nem* szót (szóközzel) döntve!

```
Font*   timesitalic12 = new Font("Times-Italic-12");
gc.Insert(4)
gc.SetFont(timesitalic12, 4);
```

A fa:



Az aktuális font meghatározásakor a fában kell lefelé haladnunk és összegeznünk a csúcsok értékeit, amíg el nem jutunk a kívánt indexig. Miután a fontváltások száma viszonylag ritka, a fa a szöveghez képest alacsony marad. Ez biztosítja, hogy nem használunk fel túl sok memóriát, ugyanakkor kellően gyorsan elvégezhető a keresés.

Utolsóként el kell készítenünk a FlyweightFactory osztálynak megfelelő GlyphFactory osztályt. Ez példányosítja a speciális glypheket.

```
const int NCHARCODES = 128;

class GlyphFactory
{
public:
    GlyphFactory();
    virtual ~GlyphFactory();
    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    ...
private:
    Character* _character[NCHARCODES];
};

GlyphFactory::GlyphFactory()
{
    for ( int i = 0; i < NCHARCODES; ++i )  _character[i] = 0;
}

Character* GlyphFactory::CreateCharacter (char c)
{
    if ( !_character[c] )  _character[c] = new Character(c);
    return _character[c];
}
```

```
Row* GlyphFactory::CreateRow()
{
    return new Row();
}

Column* GlyphFactory::CreateColumn()
{
    return new Column();
}
```

A sorok és az oszlopok létrehozásakor vegyük figyelembe, hogy azok nem megosztottak, ezért mindig új elemet állíthatunk elő. Itt most akár el is hagyhatnánk ezeket a műveleteket, és ráhagyhatnánk a kliensekre a megfelelő osztatlan objektumok létrehozását. Azonban jobb ezt az utat követnünk, ha ezek az objektumok a későbbiekben osztottá válhatnak.

27. Helyettes

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

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

Más nevek: surrogate

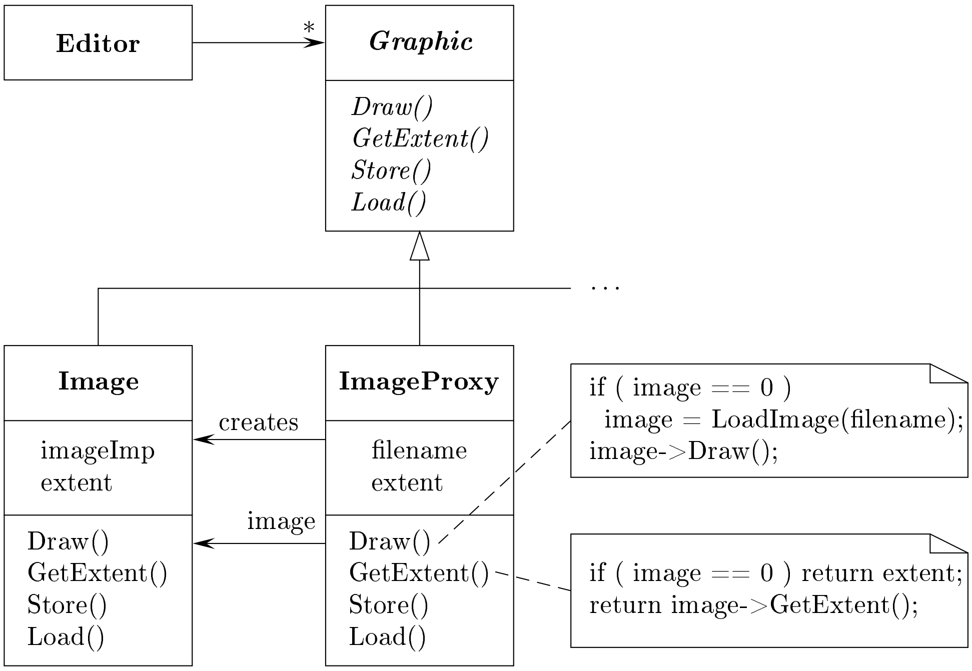
Motiváció: Egy lehetséges ok egy objektum hozzáférhetőségének szabályozására a létrehozás és inicializáció teljes költségének elhalasztása addig a pillanatig, amíg ténylegesen használni akarjuk az objektumot. Tekintsünk példaként egy dokumentum szerkesztőt, amelynek dokumentumai tartalmazhatnak grafikus elemeket is. Néhány grafikus objektum (pl.: nagy raszteres képek) létrehozása időigényes, ugyanakkor egy dokumentumot gyorsan kellene megnyitnunk. Ezt megtehetjük, ha nem hozzuk létre az összes időigényes elemet megnyitáskor. Erre amúgy sincs szükség, hiszen ezek közül legfeljebb néhány látható egyszerre.

Az eddigiek alapján a költséges (időigényes) elemeket akkor kellene létrehoznunk, amikor arra igény van, esetünkben, amikor láthatóvá válnak. De mit tegyünk egy ilyen elem helyére a dokumentumban? Hogyan tudjuk az elemet igény esetén létrehozni anélkül, hogy a szerkesztő implementációja túl bonyolulttá válna? Ez az optimalizáció ugyanis nem lehet hatással például a formázásra.

A megoldás egy másik, helyettes objektum használata, amely úgy viselkedik, mint az eredeti objektum (megadja a kiterjedést), és ha kell, létrehozza a valódi objektumot.

A helyettes objektum (ImageProxy) csak akkor hozza létre a valódi objektumot (Image), amikor a kliens (Document) ténylegesen használni akarja (megjelenítés). Ezután a helyettes a valódi objektumnak továbbítja az igényeket, ezért tárolnia kell egy hivatkozást a létrehozás után.

Tegyük fel, hogy a képeket külön fájlokban tároljuk. Ekkor a fájl nevét használhatjuk hivatkozásként. A helyettes tárolja a kép kiterjedését (szélesség, magasság). Ez biztosítja, hogy a formázó mértetre vonatkozó igényeit ki tudja elégíteni a kép példányosítása nélkül.



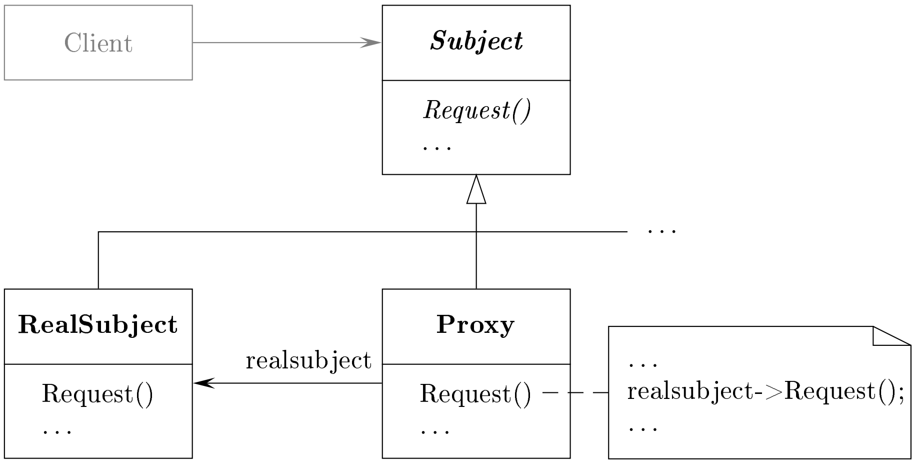
A szerkesztő az absztrakt Graphic osztály felületén keresztül kezeli a beágyazott képeket. Az ImageProxy osztály elemei olyan képek, amelyeket igény esetén hozunk létre. Ez egy fájlnévvel hivatkozik a képre, amelyet a konstruktor paraméterében kap meg.

Ezen kívül tárolja a képet befoglaló téglalapot, és egy hivatkozást a valódi képre. Ez a hivatkozás érvénytelen addig, amíg a helyettes nem példányosítja a képet. A rajzoló Draw művelet szükség esetén létrehozza a példányt. A méretet lekérdező GetExtent művelet csak akkor továbbítja az igényt, ha a kép már létrejött, ellenkező esetben a tárolt értéket adja vissza a helyettes.

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:



Elemek:

- Proxy: nyilvántart és kezel egy hivatkozást az eredeti objektumra; ezt szabályozza, létrehozza, megszünteti. A további tevékenységek a helyettes altípusától függnek.
- Subject: megadja a RealSubject és a Proxy közös felületét, ezzel biztosítva, hogy a helyettest bárhol használhassuk, ahol az eredeti objektum szerepelhet.
- RealSubject: a valódi objektum, amit a helyettes reprezentál.

Együttműködés: A helyettes alkalmas időben továbbítja az igényt a valódi objektumnak.

27.1. Esettanulmány

A motivációban leírt problémát vizsgáljuk meg, és implementáljuk virtuális helyettessel. A grafikus objektumok felületét a Graphic osztály adja meg.

```
class Graphic
{
public:
    virtual ~Graphic();
    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;
    virtual const Point& GetExtent() = 0;
    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Az Image osztály megvalósítja a Graphic osztály felületét úgy, hogy képfájlok megjelenítését teszi lehetővé. A HandleMouse művelettel a felhasználók átméretezhetik a képeket.

```
class Image : public Graphic
{
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();
    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);
    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    ...
};
```

Az ImageProxy osztály felülete megegyezik az Image osztályéval.

```
class ImageProxy : public Graphic
{
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();
    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);
    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image*   _image;
    Point    _extent;
    char*    _fileName;
};

ImageProxy::ImageProxy (const char* fileName)
{
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage()
{
    if ( _image == 0 )    _image = new Image(_fileName);
    return _image;
}

const Point& ImageProxy::GetExtent()
{
    if ( _extent == Point::Zero )
        _extent = GetImage()->GetExtent();
```

```
    return _extent;
}

void ImageProxy::Draw(const Point& at)
{
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse(Event& event)
{
    GetImage()->HandleMouse(event);
}

void ImageProxy::Save(ostream& to)
{
    to << _extent << _fileName;
}

void ImageProxy::Load(istream& from)
{
    from >> _extent >> _fileName;
}
```

Egy TextDocument osztály, amelybe beillesztehtünk Graphic objektumokat:

```
class TextDocument
{
public:
    TextDocument();
    void Insert(Graphic*);
    ...
};

TextDocument* text = new TextDocument();
    ...
text->Insert(new ImageProxy("anImageFileName"));
```

28. 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: –

Motiváció: Ha egy probléma elég gyakran fordul elő, akkor érdemes lehet a probléma példányait egy egyszerű nyelv mondataiként kifejezni. Ezután létrehozhatunk egy értelmezőt, amely a mondatok értelmezésével oldja meg a problémát.

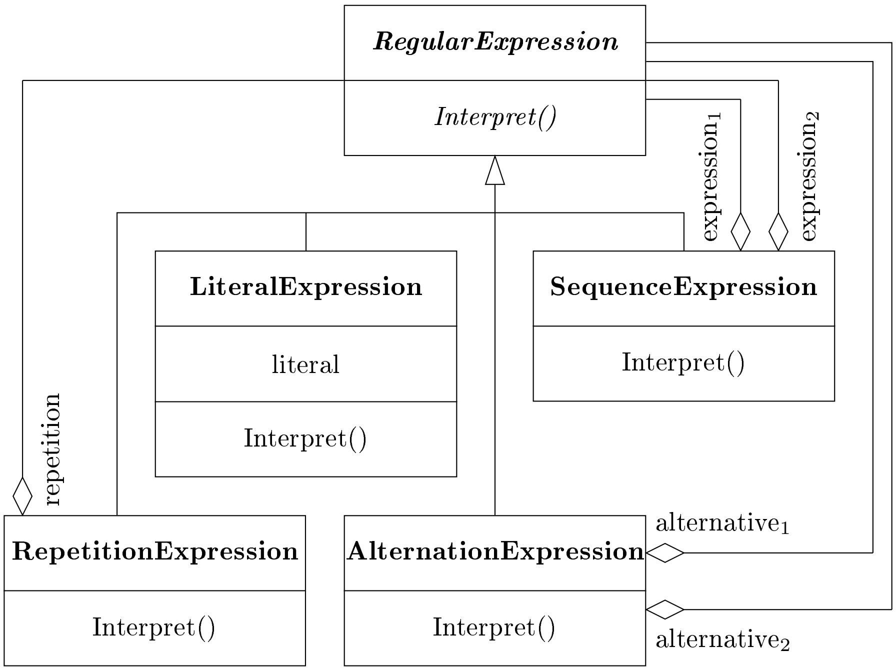
Például egy mintának megfelelő karaktersorozatok keresése gyakori probléma. A minták leírásának egy szabványos módja a reguláris kifejezések használata. Ahelyett, hogy speciális algoritmusokat készítenénk minden egyes mintára, a kereső algoritmusok értelmezhetnek egy reguláris kifejezést, ami minták halmazát írja le.

Az értelmező minta megadja, hogy miként definiáljunk nyelvtant egyszerű nyelvekhez, hogyan ábrázoljunk és értelmezzünk mondatokat a nyelvből. Reguláris kifejezések esetén a minta leírja a reguláris kifejezések nyelvtanának meghatározási módját, hogy miként ábrázoljunk egy reguláris kifejezést, és hogyan értelmezzük azt.

Tegyük fel, hogy a következő nyelvtan adja meg a reguláris kifejezéseket. (Ebben az `expression` a kezdő szimbólum, `literal` a terminális, amely egyszerű szavakat ad meg.)

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression  '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

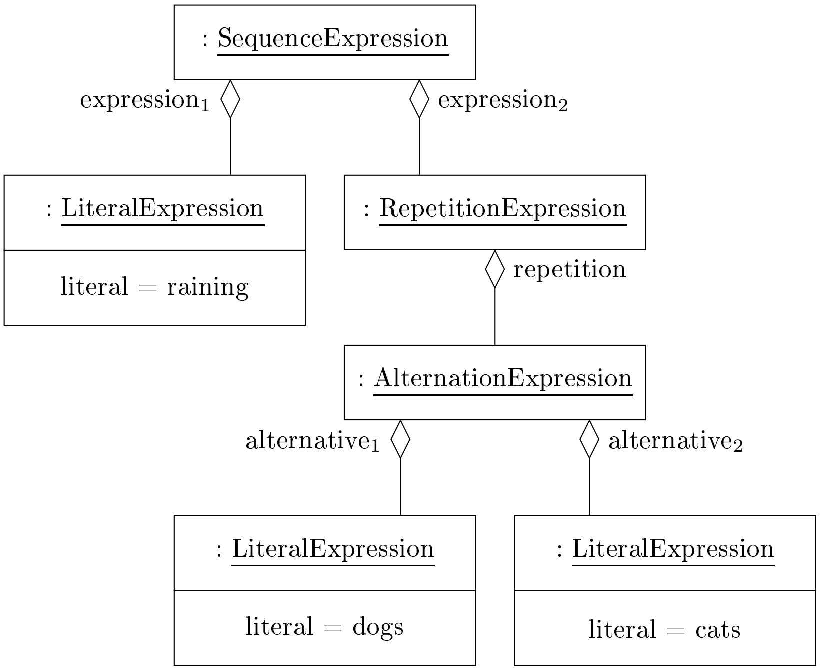
Az értelmező mintában minden egyes nyelvtani szabályt egy osztállyal reprezentálunk. A szabály jobb oldalán található szimbólumok az osztály példány változói. A fenti nyelvtant 5 osztállyal reprezentáljuk. Ezek: a RegularExpression absztrakt osztály, és négy leszármazottja a LiteralExpression, az AlternationExpression, a SequenceExpression és a RepetitionExpression. Az utóbbi három osztályban változókat definiálunk a rész-kifejezések tárolására.



A nyelvtan által definiált összes reguláris kifejezést egy absztrakt szintaxisfa reprezentálja, amelynek csúcsai az osztályok példányai. Például a

```
raining & (dogs | cats)*
```

reguláris kifejezésnek megfelelő fa a következő.

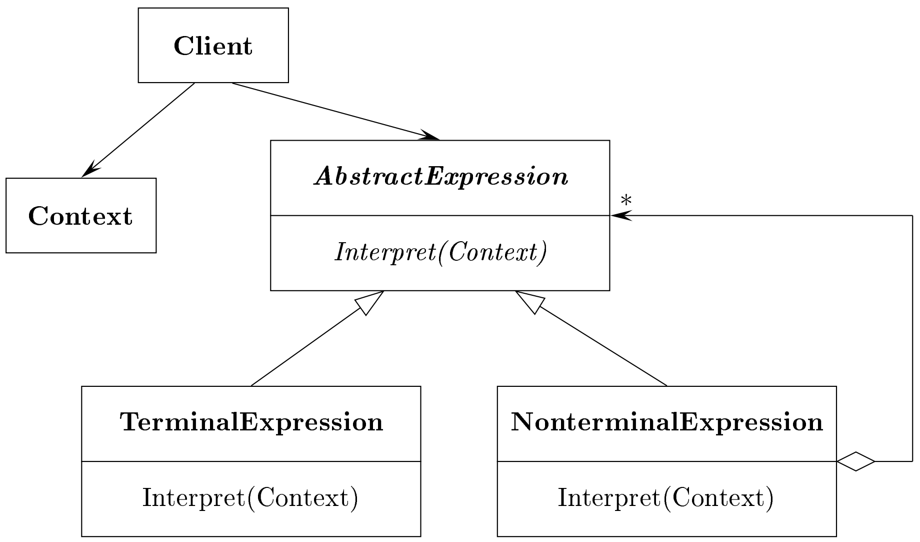


Az értelmezéshez az osztályok Interpret műveletét kell definiálnunk. A művelet paraméterként átveszi a környezetet, amelyben a kifejezést értelmezni kell. A környezet tartalmazza a bemeneti sorozatot, és az eddig illesztett rész hosszát. A művelet illeszti a bemeneti sorozat következő részét. A LiteralExpression osztály esetén ellenőrizzük, hogy a bemenet megfelel-e a tárolt sorozatnak; az AlternationExpression osztály esetén megvizsgáljuk, hogy megfeleltethető-e valamelyik esetnek; a RepetitionExpression osztálynál ellenőrizzük, hogy az ismételt kifejezés többször illeszthető-e, stb.

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:



Elemek:

- AbstractExpression: deklarálja az absztrakt Interpret műveletet, ami közös a szintaxisfa minden elemére nézve.
- TerminalExpression: megvalósítja az Interpret műveletet a terminális jeleknek megfelelően; minden terminális jelhez egy példány tartozik.
- NonterminalExpression: minden nyelvtani szabályhoz tartozik egy ilyen osztály; a szabály jobb oldalán található komponensekre hivatkozik; megvalósítja az Interpret műveletet a komponensek segítségével.
- Context: az értelmező által használt globális információ.
- Client: felépíti a konkrét mondatnak megfelelő szintaxisfát, és kiadja az Interpret műveletet.

Együttműködés:

- A kliens felépíti a NonterminalExpression és TerminalExpression osztályok példányaiból a mondatnak megfelelő szintaxisfát. Inicializálja a környezetet, és kiadja az Interpret műveletet.
- Mindegyik NonterminalExpression csúcs a komponensek segítségével definiálja az Interpret műveletet. A TerminalExpression objektumok Interpret műveletei adják meg a rekurzió legalsó szintjét.
- Minden csúcs Interpret művelete a környezetet használja az értelmezés állapotának tárolására és elérésére.

28.1. Esettanulmány

Logikai kifejezések kezelését és kiértékelését vizsgáljuk meg. A terminális jelek ekkor a logikai értékek: `true` és `false`, és logikai változók. A kifejezésekben három operátort használhatunk: `and`, `or` és `not`. Az egyszerűség érdekében tekintsük el a precedenciától! A nyelvtan a következő.

```
BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp |
               '(' BooleanExp ')'
AndExp ::= BooleanExp  'and' BooleanExp
OrExp ::= BooleanExp  'or' BooleanExp
NotExp ::= 'not' BooleanExp
Constant ::= 'true' |  'false'
VariableExp ::= ('A' | ... | 'Z' | 'a' | ... | 'z')*
```

Két műveletet adunk meg logikai kifejezésekre. Az Evaluate művelet kiértékeli a kifejezést és igaz vagy hamis értéket rendel a változókhoz. A Replace művelet egy új logikai kifejezést hoz létre úgy, hogy egy változó értékét egy kifejezéssel helyettesíti. (Ez megmutatja, hogy az értelmező mintát nem csak kifejezések kiértékelésére lehet használni.)

Az OrExp és NotExp osztályokat nem adjuk meg, ezek az AndExp osztályhoz hasonló módon elkészíthetőek. A Constant osztály reprezentálja a logikai konstansokat.

```
class BooleanExp
{
public:
    virtual ~BooleanExp();
    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
protected:
    BooleanExp();
};
```

A Context osztály megadja az egyes változók logikai értékét. Az értéket le lehet kérdezni (Value), illetve be lehet állítani (Assign).

```
class Context
{
public:
    bool Value(const char*) const;
    void Assign(VariableExp*, bool);
};
```

A VariableExp osztály egy névvel ellátott változót reprezentál.

```
class VariableExp : public BooleanExp
{
public:
    VariableExp(const char*);
    virtual ~VariableExp();
    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};

VariableExp::VariableExp(const char* name)
{
    _name = strdup(name);
}

bool VariableExp::Evaluate(Context& c)
{
    return c.Value(_name);
}

BooleanExp* VariableExp::Copy() const
{
    return new VariableExp(_name);
}

BooleanExp* VariableExp::Replace(const char* name, BooleanExp& exp)
{
    if ( strcmp(name, _name) != 0 )   return exp.Copy();
    return new VariableExp(_name);
}
```

Az AndExp osztály két logikai kifejezés 'és' kapcsolatát adja meg.

```
class AndExp : public BooleanExp
{
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~ AndExp();
    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
```

```
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* operand1;
    BooleanExp* operand2;
};

AndExp::AndExp(BooleanExp* op1, BooleanExp* op2)
{
    operand1 = op1;  operand2 = op2;
}

bool AndExp::Evaluate(Context& c)
{
    return operand1->Evaluate(c) && operand2->Evaluate(c);
}

BooleanExp* AndExp::Copy() const
{
    return new AndExp(operand1->Copy(), operand2->Copy());
}

BooleanExp* AndExp::Replace(const char* name, BooleanExp& exp)
{
    return new AndExp(operand1->Replace(name, exp),
                      operand2->Replace(name, exp));
}
```

Most adjuk meg a `(true and x) or (y and (not x))` logikai kifejezést, és értékeljük ki x = hamis és y = igaz esetén.

```
BooleanExp*  expression;
Context      context;
VariableExp* x = new VariableExp("x");
VariableExp* y = new VariableExp("y");

expression = new OrExp(new AndExp(new Constant(true), x),
                       new AndExp(y, new NotExp(x)));

context.Assign(x, false);
context.Assign(y, true);
bool result = expression->Evaluate(context);
```

A környezet (context) megváltoztatásával különböző x és y értékekre számíthatjuk ki a kifejezés értékét. Végül kicserélhetjük y változót egy új kifejezéssel, és ennek megfelelő kiértékelést is végrehajthatunk.

```
VariableExp* z = new VariableExp("z");
NotExp       not_z(z);
BooleanExp*  replacement = expression->Replace("y", not_z);

context.Assign(z, true);
result = replacement->Evaluate(context);
```

29. Konkurens rendszerek mintái

Lehetséges az osztályok bővítése új mintákkal, illetve újabb osztályok bevezetése a megoldott feladat jellege alapján. Újabb osztályok lehetnek például:

- Konkurens rendszerekhez kapcsolódó minták.
- GUI minták.
- Adatbázis kezeléssel kapcsolatos minták.

Konkurens rendszerek esetén néhány lehetséges minta:

Ütemező (Scheduler): vezérelni, hogy szálak milyen sorrendben férhetnek hozzá egy közös erőforráshoz, annak használatából egymást kölcsönösen kizárva (egyszerre csak egy használhatja). A minta megadja miként lehet az ütemezést megvalósítani, függetlenül bármilyen speciális ütemezési módszertől.

Olvasó/Író zárolás (Read/Write Lock): engedélyezni adatok egyidejű olvasását, biztosítva a kizárólagos írási hozzáférést.

Termelő-Fogyasztó (Producer-Consumer): objektumok aszinkron előállításának és felhasználásának biztosítása.

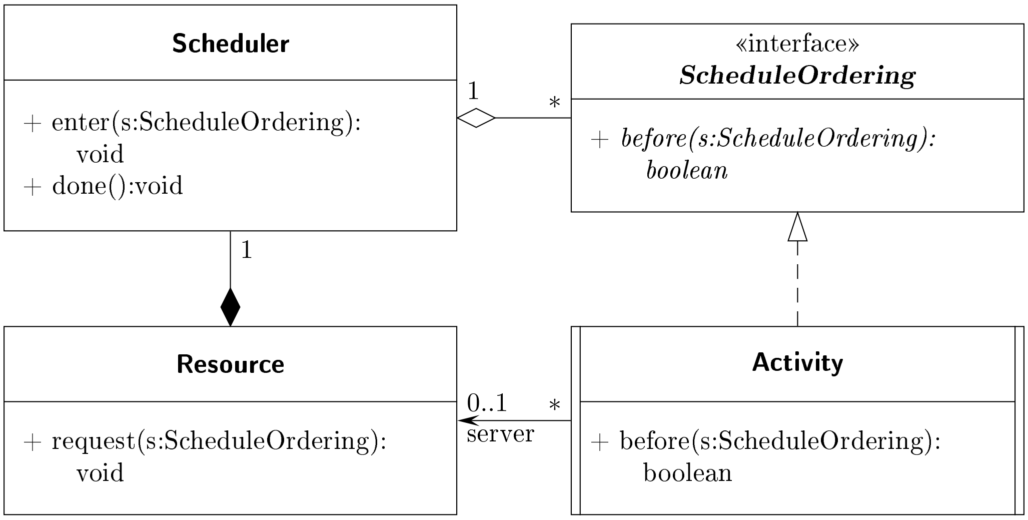
29.1. Ütemező

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

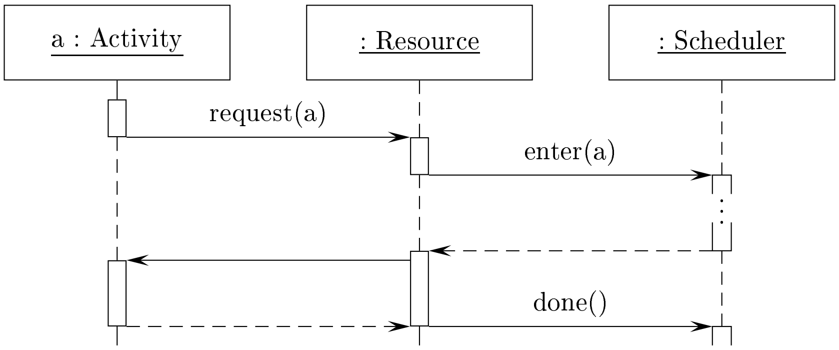
ScheduleOrdering: Az ütemezéshez használt megelőzési relációt megadó interfész.

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.

```java
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:

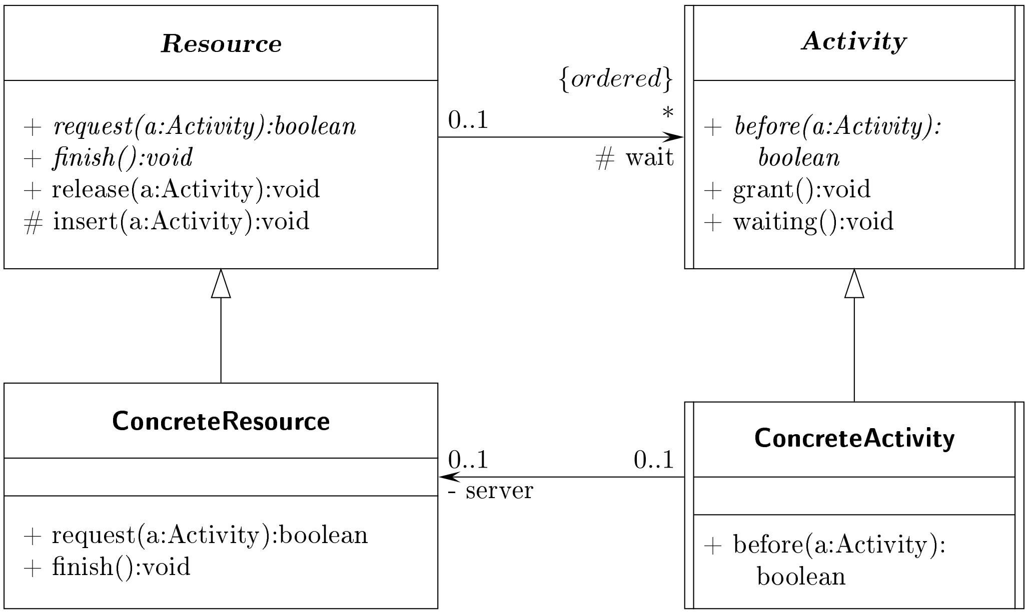
- Szinte minden vezérlés az ütemezőbe kerül, így a tevékenységek kezelése (felfüggesztése) is. Ehhez ismerni kell azok kezelési módját, azaz nem csak a ***ScheduleOrdering*** interfészt használjuk. Esetünkben ez az implementációban a *threads* tömbben tárolt szálakat *Thread*-eket, és kezelésüket jelenti.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

29.2. Menedzser

Cél

Az ütemező minta hiányosságainak javítása, illetve figyelembe venni, hogy a tevékenységek nem feltétlen várakoznak tetszőleges ideig az erőforrásra, egy adott idő lejárta után lemondhatnak arról.

Szerkezet



Elemek

Resource: A közös erőforrásokat leíró absztrakt osztály. Megadja a használat felületét (igénylés, befejezés, lemondás).

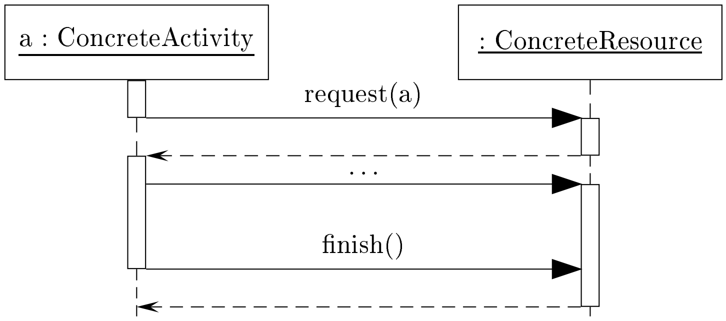
Activity: Az erőforrást használni akaró tevékenységek, szálak közös felülete. Deklarálja a sorrendbe állítás alapját adó relációt, és megadja a várakoztatás, illetve használat engedélyezés műveletét.

ConcreteResource: A konkrét erőforrás, amely megadja a használathoz szükséges műveleteket, és megvalósítja az igénylés, valamint a befejezés műveletét.

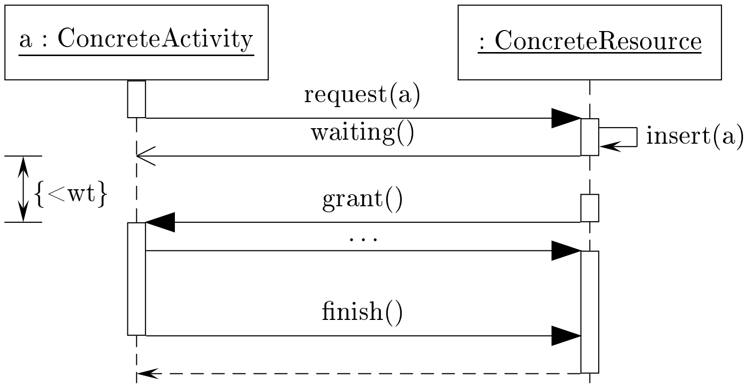
ConcreteActivity: A konkrét tevékenysége osztálya, a megelőzési reláció megvalósításával.

Együttműködés

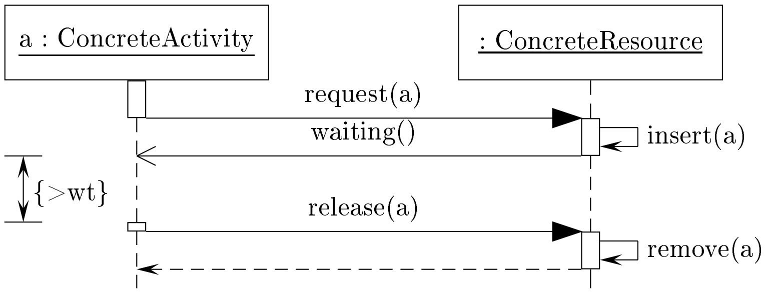
Az első szekvenciadiagram azt az esetet szemlélteti, amikor az erőforrás az igénylés pillanatában szabad.



A második esetben az erőforrás foglalt az igénylés pillanatában, és a tevékenység várakozási időn (wt) belül sorra kerül.



A harmadik esethez tartozó szekvenciadiagram azt mutatja, amikor az erőforrás foglalt, és a tevékenység nem kerül sorra a várakozási időn belül.



Példa kód

A Java megvalósításban a ConcreteActivity osztállyal még nem foglalkozunk, lehetséges eseteket később mutatunk be.

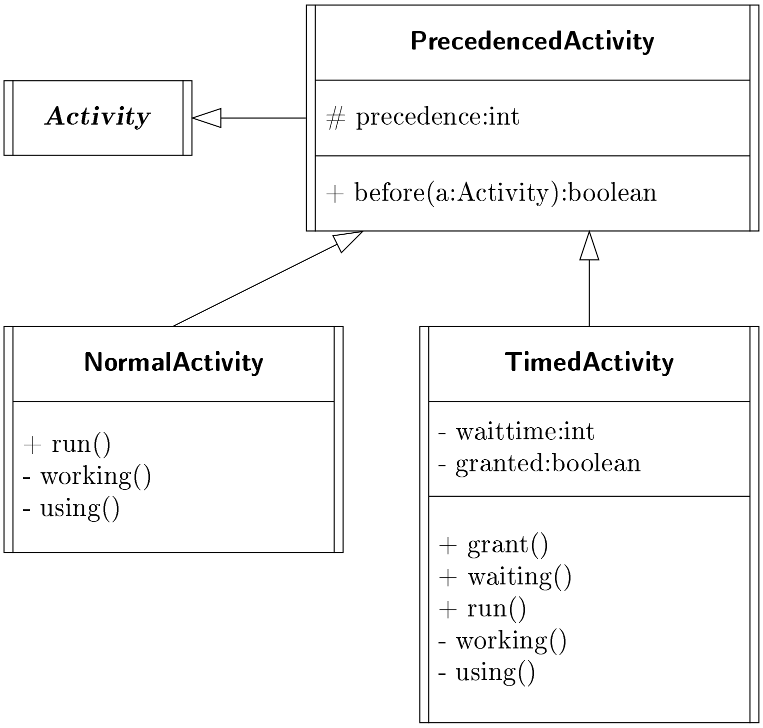
```
import java.util.Vector;

public abstract class Resource
{
    protected Vector<Activity> wait;
```

```
    public Resource() { wait = new Vector<Activity>(); }

    public abstract boolean request(Activity a);

    public abstract void finish();

    public synchronized void release(Activity a)
    {
        wait.remove(a);
    }

    protected synchronized void insert(Activity a)
    {
        int i;
        for ( i = 0; i < wait.size(); i++ )
            if ( a.before(wait.get(i)) ) break;
        wait.add(i, a);
    }
}

public abstract class Activity extends Thread
{
    public Activity() { }

    public abstract boolean before(Activity a);

    public synchronized void grant() { notify(); }

    public synchronized void waiting()
    {
        try { wait(); } catch (InterruptedException e) {}
    }
}

public final class SingleResource extends Resource
{
    private boolean free;

    public SingleResource() { free = true; }

    public synchronized void request(Activity a)
    {
        if ( !accessible() )
        {
            insert(a);
            a.waiting();
        }
    }

    public synchronized void finish()
    {
        if ( wait.isEmpty() ) free = true;
        else wait.remove(0).grant();
    }
```

```
    private synchronized boolean accessible()
    {
        boolean res = free;
        free = false;
        return res;
    }
}
```

Időzített várakozás, precedencia használata

A konkrét tevékenységek esetén használhatunk például precedenciát az ütemezés szabályozására (**ConcreteActivity** = **PrecedencedActivity**). Ezt mutatja a következő osztálydiagram, ahol bevezetjük az ütemező mintának megfelelően tetszőleges ideig várakozó (**NormalActivity**), illetve legfeljebb adott ideig várakozó (**TimedActivity**) tevékenységeket.



Példa kód

Az előzőekben megadott Java kódot kell kiegészítenünk az új osztályokkal. Az osztályokban a szimulációt az ütemező mintában megismert módon valósítjuk meg.

```
public class PrecedencedActivity extends Activity
{
    protected int precedence;

    PrecedencedActivity(int pr) { precedence = pr; }
```

```
    public boolean before(Activity a)
    {
        if ( a instanceof PrecedencedActivity )
            return precedence > ((PrecedencedActivity)a).precedence;
        return false;
    }
}

public class NormalActivity extends PrecedencedActivity
{
    private static final int UNIT = 100;
    private static boolean running = true;
    public static void stopSimulation() { running = false; }

    private String name;
    private int usetime;
    private int worktime;
    private SingleResource server;

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

    public void run()
    {
        while ( running )
        {
            working();
            System.out.println(name + ": requesting");
            server.request(this);
            using();
        }
    }

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

    private void using()
    {
        System.out.println(name + ": using");
        try { sleep(usetime); } catch (InterruptedException e) {}
        server.finish();
    }
}
```

```
public class TimedActivity extends PrecedencedActivity
{
    private static final int UNIT = 100;
    private static boolean running = true;
    public static void stopSimulation() { running = false; }

    private String name;
    private int usetime;
    private int worktime;
    private int waittime;
    private SingleResource server;
    private boolean granted;

    public TimedActivity(String name, int precedence,
                int usetime, int worktime, int waittime, SingleResource server)
    {
        super(precedence); granted = true;
        this.server = server; this.name = name; this.usetime = usetime * UNIT;
        this.worktime = worktime * UNIT; this.waittime = waittime * UNIT;
    }

    public synchronized void grant()
    {
        granted = true; notify();
    }

    public synchronized void waiting()
    {
        granted = false;
        System.out.println(name + ": waiting");
        try { wait(waittime); } catch (InterruptedException e) {};
    }

    public void run()
    {
        while ( running )
        {
            working();
            server.request(this);
            if ( granted )  using();
            else
            {
                System.out.println(name + ": time elapsed");
                server.release(this);
                granted = true;
            }
        }
    }

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

```
    private void using()
    {
        System.out.println(name + ": using");
        try { sleep(usetime); } catch (InterruptedException e) {}
        server.finish();
    }
}
```

30. 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:

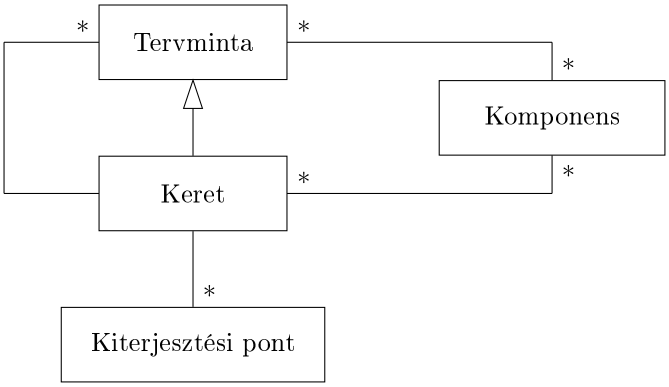
1. *A tervminták absztraktabbak, mint a keretek.* A keretekhez tartozik programkód, a tervminták esetén legfeljebb példákat lehet megadni.
2. *A tervminták kisebb szerkezeti egységek, mint a keretek.* Egy keret rendszerint több tervmintát tartalmaz, de ez fordítva nem áll fenn.

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 teszttervek 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.