

Tartalomjegyzék

1.1.	Függőség megfordításának alapelve – DIP (Dependency Inversion Principle)	2
1.1.1.	Az autós példa	3
1.1.2.	A híres Copy példa	6
1.1.3.	A híres adatbázis-kezelő dilemma	12
1.1.4.	A DIP UML osztály diagram ábrája	14
1.1.5.	A HAS-A kapcsolat 3 fajtája és a DIP	15
1.1.6.	A 3 rétegű DIP	17
1.1.7.	Dolgok szétválasztása, Kontroll megfordítása, Függőség megfordítása	18
1.1.8.	Kommunikáció a felső rétegből az alsó réteg felé	20
1.1.9.	Egy tervezési hiba: A körkörös kommunikáció	21
1.1.10.	Kommunikáció az alsó rétegből a felső réteg felé, hibajelentések	24
1.1.11.	Visszacsatolás, szabályozás, szabályozási kör	25
1.1.12.	DIP és LSP: Hogyan zárjuk egységbe az API szerződést	30
1.1.13.	DIP és az állapotgép	45
1.1.14.	DIP vizsgálata Imperatív vs. Deklaratív szemszögből	52
1.2.	Tervezési minták és a DIP	53
1.2.1.	Szétválasztás a fókuszban: Híd és Megfigyelő	54
1.2.2.	Híd	54
1.2.3.	Megfigyelő	55
1.3.	Létrehozási tervezési minták	59
1.3.1.	Absztrakt gyár	59
1.3.2.	Építő	61
1.3.3.	Prototípus	64
1.4.	Szerkezeti tervezési minták	68
1.4.1.	Díszítő és Összetétel	68
1.4.2.	Pehelysúlyú	71
1.4.3.	Helyettes	73
1.5.	Viselkedési tervezési minták	75
1.5.1.	Felelősséglánc	75
1.5.2.	Parancs	77
1.5.3.	Értelmező	79
1.5.4.	Iterátor	80

1.5.5.	Közvetítő	83
1.5.6.	Pillanatkép	88
1.5.7.	Állapot	90
1.5.8.	Stratégia	97
1.5.9.	Sablon- és Gyártó Metódus	99
1.6.	Azok a tervezési minták, amiben nincs DIP	101
1.6.1.	Egyke	101
1.6.2.	Homlokzat	102
1.6.3.	Illesztő	104
1.6.4.	Látogató	105
1.7.	Mikor ne használjuk a DIP tervezési alapelvet	109

1.1. Függőség megfordításának alapelve – DIP (Dependency Inversion Principle)

A függőség megfordításának alapelve (angolul: Dependency Inversion Principle – DIP) azt mondja ki, hogy a magas szintű modulok ne függjenek az alacsony szintűektől, se fordítva, mindkettő az absztrakciótól függjön. Azaz, az absztrakció nem függhet a részletektől, épp fordítva, a részletek függhetnek az absztrakciótól.

Eredeti angol megfogalmazásban: „The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.”

A DIP a SOLID elvek közül az utolsó. Központi szerepet játszik az Objektorientált Programozás (OOP) tervezési (angolul: Object-Oriented Programming Design – OOPD) alapelvek közt, hiszen nagyon könnyen megfogalmazható UML ábra szintjén: Minden nyíl mutasson absztrakcióra! Továbbá a segítségével könnyen több rétegű programot tudunk készíteni, ahol az egyes rétegeket egy-egy absztrakció választja el. Abban is segít, hogy hogyan érdemes olyan magas szintű alapelveket alkalmazni, mint a Dolgok Szétválasztása (angolul: Separation of Concerns). Illetve segít alacsonyabb szinten lévő tervezési mintákban meglátni a közös részt: Sok tervezési minta arra jó, hogy szétválasszunk dolgokat.

A DIP a szenior programozók, rendszertervezők, архитеktek kelléktárának alapeleme. A Dependency Injection Framework-ok alapját megfogalmazó tervezési alapelv.

Olyan, mint a kezdő programozóknak az öröklődés, imádjuk, és egy kicsit hajlamosak vagyunk túlhasználni.

A DIP-nek ugyanis vannak veszélyei is. Néha szétválasztunk olyasmit is, amit inkább jobb lenne egyben tartani. Néha nem egyértelmű, hogy mi az alacsony szintű rész, mi a magas.

Habár vannak ököl szabályok: Ami hardver közeli, az alacsonyszintű, ami felhasználó közeli, az magas szintű. Ugyanakkor a program kezdő képernyője, ami a felhasználóval kommunikál, azaz teljes mértékben felhasználó közeli, az mégsem magas szintű komponens, hanem inkább alacsony szintű, hiszen csak valamilyen frontend technológiát használó nagyon konkrét részlet.

A másik ilyen veszély, ha már 15 rétegem van és a 10. rétegben keletkező hibát a 3. rétegben kellene feldolgoznom, akkor ez már nem is olyan egyszerű.

Mivel a réteg fogalmát ilyen sokszor használtuk, nézzük meg a definícióját. Azt mondjuk, hogy a P program L része egy réteg, akkor és csak akkor, ha L annyira független P többi részétől, hogy akár másik gépen futhat, nem muszáj ugyanazon a gépen futnia, mint a program többi részének.

Pl.: 3 rétegű architektúra esetén másik gépen fut a web-browser, a BI (angolul: business intelligence, magyarul: üzleti logika) és az adatbázis-kezelő.

Sokan kritizálják a függőség megfordításának elvét, miszerint az csak az objektum-összetétel használatának, azaz a GOF2 elvnek egy következménye. Mások szerint ez egy önálló tervezési minta. Mindenesetre, mint látni fogjuk, nagyon hasznos, ha rugalmas kód fejlesztésére törekszünk.

1.1.1. Az autós példa

Tehát a függőség megfordításának alapelve (angolul: Dependency Inversion Principle – DIP) azt mondja ki, hogy a magas szintű modulok ne függjenek az alacsony szintűektől, se fordítva, mindkettő az absztrakciótól függjön. Azaz, az absztrakció nem függhet a részletektől, épp fordítva, a részletek függnek az absztrakciótól.

Miért fontos ez? Képzeliük el, hogy autót vezetünk. A kormányt és a pedálokat használva vezetjük az autót, hogy elérjük az úticélunkat. Néha ránézünk a sebességmérőre és a fordulatszámmérőre, nyomjuk a gázt, és megy az autó!

Eközben az autóban ezer apró részlet megy végbe, a robbanómotorban berobban a benzin és a levegő keveréke, a robbanás ereje felnyom egy dugattyút, a dugattyú megforgatja a hajtókart, ami megforgatja a főtengelyt, de közben a többi dugattyút lenyomja, a főtengely forgása több áttételen keresztül megforgatja a kerekeket, miközben a váltó állítja az áttételt, ezáltal a fordulatszámot, amit végsősoron gázadással, vagy fékezéssel befolyásolunk.

És ez a rengeteg apró részlet attól az absztrakt tudástól függ, amit úgy nevezünk, hogy vezetés. Miközben vezetünk, programozói fejjel gondolkozva, magas szintű metódusokat hívunk, gázadás, fékezés, kormány fordul balra, kormány fordul jobbra. Ezek a hívások alsó szinten implementálva vannak és a konkrét fizikai autó mozgását eredményezik.

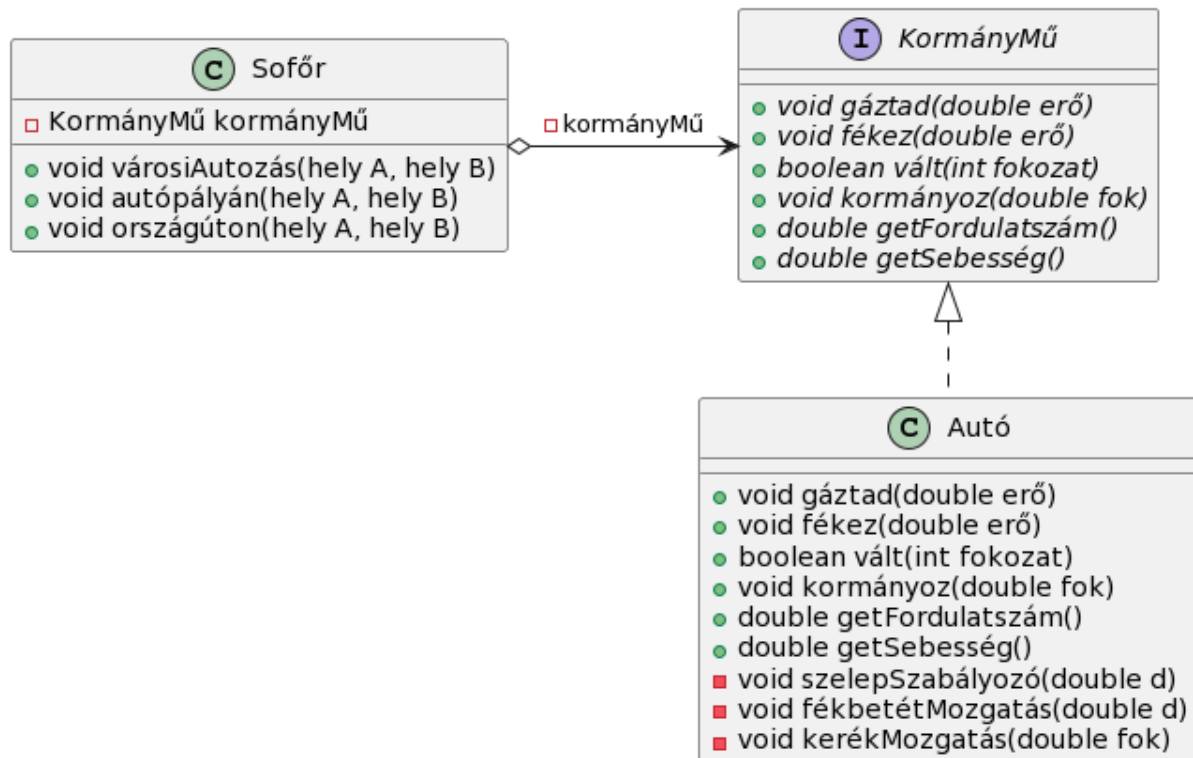
Hol van itt az absztrakció? Minden nagyon konkrétnek tűnik.

Akkor látjuk meg az absztrakciót, amikor átülünk egy másik autóba, és csodák csodája, azt is tudjuk vezetni. Hiszen a kormányművek, a gázpedál, a fékpedál, a kormánykerék, a váltó működése látszólag ugyanaz, habár egy benzines, egy dízel, vagy egy hibrid autóban teljesen más konkrét folyamatok zajlanak le. Mégis, a kormánymű lényegében ugyanaz, a visszajelző műszerek lényegében ugyanazok, azaz felület ugyanaz, és így bármelyik autót tudjuk vezetni!

Igaz ez, hogy minden autóban ugyanaz a kormánymű? Nyilván nem igaz, hiszen van 5-sebességes, 6-sebességes, 7-sebességes autó, sőt, van automataváltós autó, sőt akár a kormánykerék lehet a másik oldalon.

Mégis, még ezeket az autókat is tudjuk vezetni az absztrakt vezetési tudásunkkal, miszerint, ha gázt adunk, akkor gyorsul az autó, ha fékezünk, lassul, ha jobbra fordítjuk a kormánykereket, akkor jobbra fordul, ha balra, akkor balra. Ha túl nagy a fordulatszám, akkor felfelé kell váltani, ha nagyobb erőátvitelt szeretnénk, mert mondjuk lejtőn felfelé megyek, akkor vissza kell váltani.

Az autós példa UML ábrája:



PlantUML szkriptje:

@startuml

class Sofőr {

-KormányMű kormányMű

+void városiAutozás(hely A, hely B)

+void autópályán(hely A, hely B)

+void országúton(hely A, hely B)

}

interface KormányMű {

+{abstract} void gáztad(double erő)

+{abstract} void fékez(double erő)

+{abstract} boolean vált(int fokozat)

+{abstract} void kormányoz(double fok)

+{abstract} double getFordulatszám()

+{abstract} double getSebesség()

}

class Autó {

+void gáztad(double erő)

```

+void fékez(double erő)
+boolean vált(int fokozat)
+void kormányoz(double fok)
+double getFordulatszám()
+double getSebesség()
-void szelepSzabályozó(double d)
-void fékbetétMozgatás(double d)
-void kerékMozgatás(double fok)
}

```

Sofőr o-right-> KormányMű : -kormányMű

Autó .up.|> KormányMű

@enduml

Elemezzük ezt az UML ábrát. Első, ami feltűnik, hogy van IS-A és HAS-A kapcsolat is. Az IS-A kapcsolatból egy interfész implementációt látunk. Az Autó osztály implementálja a KormányMű interfészt. HAS-A kapcsolatból egy aggregációt látunk: A Sofőr birtokol egy KormányMű példányt a kormányMű referencián keresztül. Ez az információ kétszer is szerepel az ábrán, hiszen látjuk ezt a mezőt a Sofőr osztályon belül és a HAS-A kapcsolat nyila felett is. Ilyen értelemben hibás az ábra, hiszen felesleges egy információt kétszer is feltüntetni, de most ezen az első ábrán ez cél volt. Általában csak a nyíl felé írjuk a kapcsolatot megvalósító referencia nevét.

Nagyon fontos: UML ábrán, ha a nyíl felett van szöveg, akkor az a kapcsolatot megvalósító referencia neve, ami leggyakrabban egy mező. Még egyszer, mert ez nagyon fontos: **A nyíl felé írjuk a kapcsolatot megvalósító mező nevét!**

Vegyük észre továbbá, hogy ami absztrakt az az ábrán dőlt betűvel van írva. Ez nagyon fontos: Ami az UML ábrán dőlt, az a programban absztrakt! Persze a forráskódban egy interfészen belül nem kell kiírni minden metódushoz, hogy absztrakt, hiszen ez az interfész lényege, de a PlantUML szkriptben, ahogy a fenti példában is látjuk, minden interfész belüli metódus neve elé oda kell írni, hogy {abstract}.

Minden másban nagyon hasonlít egy C# vagy Java forráskód a fenti PlantUML szkripthez. Éppen ezért annyira könnyű megtanulni a PlantUML.com használatát. Amit még el kell sajátítani, azok a nyilak, de ezek is elég intuitívak:

Ősosztály <|-- Gyermekosztály, de fordítva is használható: Gyermekosztály --|> Ősosztály.

Interfész <|.. Megvalósító, de fordítva is használható: Megvalósító ..|> Interfész.

Birtokos o--> Birtokolt : a birtoklást megvalósító mező neve és láthatósági szintje, de fordítva is használható:

Birtoklot <--o Birtokos : a birtoklást megvalósító mező neve és láthatósági szintje.

Az intuíció pedig ez: a „<|” hasonlít egy háromszögre, az „o” hasonlít egy rombuszra. Illetve azért kel a háromszög után két mínusz jel, hogy közé lehessen írni a nyíl irányát: fel (up), le (down), jobbra (right), balra (left). A fenti PlantUML példából itt láthatunk erre példát:

Sofőr o-right-> KormányMű : -kormányMű

Autó .up.|> KormányMű

Az ábrán azt látjuk, hogy a Sofőr tud városban, országúton és az autópályán A pontból B pontba autózni. Nyilván városon belül más autózási stílust használ, mint országúton, megint mást autópályán. Ugyanakkor, bárhol és bárhogyan is autózik, az autót a KormányMű hívásaival vezérli a kormányMű referencián keresztül.

Az Autó megvalósítja a KormányMű metódusait és még néhány privát metódust, amivel a saját motorját és kerekeket vezérli alacsony szinten. Ezeket már a Sofőr nem hívhatja, nem állíthatja közvetlenül a fékberakást és a szelepek állását, de közvetve gázadással, fékezéssel befolyásolhatja azokat.

A KormányMű zárja egységbe azt az ismeretet, másnéven API-t, amit minden sofőrnek ismernie kell, milyen felületen keresztül lehet egy autót irányítani, függetlenül attól, hogy ez az autó benzines, dízeles, hibrid, tisztán elektromos, gázos, hidrogéncellás, etanolos, std Illetve, hogy hány sebességes, automata váltós, vagy bármilyen más.

A KormányMű adja az Autó definícióját: Autó az, ami megvalósítja a KormányMű interfészt. Ugyanakkor a KormányMű adja a Sofőr definícióját is: Sofőr az, aki képes használni a KormányMű interfészt. Mint látjuk, a részletek, a Sofőr és az Autó is, az absztrakciótól, a KormányMű interfésztől függ.

Azt érezzük, hogy itt még sok-sok részlet hiányzik: Hol van az ABS, a motorhőmérséklet? Ugyanakkor emlékezzünk arra, hogy a DIP alapelv szerint a programozás nem a részletek megadásának művészete. Épp ellenkezőleg, a DIP alapelv szerint a programozás az absztrakciók létrehozásának a művészete, úgy, hogy a részletek függenek az absztrakcióktól.

1.1.2. [A híres Copy példa](#)

Nézzük a szakirodalomban legelterjedtebb példákat, hogy jobban megértsük, mit jelentenek ezek a fogalmak:

- magas szintű modul,
- alacsony szintű modul,
- részletek,
- absztrakció,
- függés a részletektől,
- függés az absztrakciótól.

Kezdjük a híres Copy példával. Vegyük a következő egyszerű leíró nyelven íródott kódot:

```
public void Copy() { while( (char c = Console.ReadKey()) != EOF) Printer.PrintChar(c); }
```

Ez a szuper egyszerű metódus, addig olvassa a konzolt, amíg end of file karaktert nem olvas. A beolvasott karaktereket nyomja ki a printerre, tehát konzolról másol a printerre. Itt a Copy() metódus függ a Console.ReadKey() és a Printer.PrintChar() metódustól, azaz, ha ezeknek a metódusoknak a viselkedése változik, azaz változik egy részlet, akkor ennek a magas szintű metódusnak is változnia kell.

A Copy() metódus fontos logikát ír le, a forrásból a kimentre kell másolni karaktereket file vége jelig. Mivel itt nem egy részletet implementálunk, hanem logikát kódolunk le, ezért ez egy magas szintű metódus. Ezt a logikát sok helyen fel lehet használni, hiszen a forrás bármi lehet, bármi, ami karaktereket tud beolvasni, illetve a kimenet is bármi lehet, bármi, ami karaktert tud kiírni.

Ha most ezt a kódot újra akarom hasznosítani, akkor az első megoldás, hogy egy if – else – if szerkezet segítségével megállapítom, hogy most mi a forrás:

- ha a konzol, akkor a beolvasás kódja: `char c = Console.ReadKey();`
- ha TXT fájl, akkor `char c = txtFájl.ReadChar();`
- ha bináris fájl, akkor `char c = binFájl.Read() as char;`
- és ki tudja még hány lehetséges változat van.

Ugyanez a másik oldalon, egy if – else – if szerkezet segítségével megállapítom, hogy most mi a kimenet. Attól függően, hogy mi a kimenet, attól függően más és más konkrét hívást kell hívnom:

- ha kimenet a printer, akkor a kiírás kódja: `Printer.printChar(c);`
- ha TXT fájl, akkor `txtFájl.WriteChar(c);`
- ha bináris fájl, akkor `binFájl.Write(c as byte);`
- és ki tudja még hány lehetséges változat van.

Egy egyszerű leíró nyelven ez így fogalmazható meg:

```
public void Copy(Object in, Object out) {  
    char c;  
  
    if (in is Console) c = ((Console) in).ReadKey();  
    else if (in is Text) c = ((Text) in).ReadChar();  
    else c = 26;  
  
    while( (c != EOF) {  
        if (out is Printer) ((Printer) out).PrintChar();  
        else if (out is Text) c = ((Text) out).WriteChar();  
        if (in is Console) c = ((Console) in).ReadKey();  
        else if (in is Text) c = ((Text) in).ReadChar();  
    }  
}
```

Hú, ez nehéz volt. Több ember többször átnézte, hogy minden jó-e. Ez a módszer nagyon csúnya, könnyen elrontható, nehezen átlátható, nehezen módosítható kódot eredményez, ami nyilván megszegi az OCP elvet. Vegyük észre, hogy ez a kód a részletektől függ. Ha valamelyik részlet megváltozik, azaz, ha megjelenik egy újfajta bemenet, vagy kimenet, akkor a fenti kódot változtatnom kell.

Egy másik lehetőség, hogy minden bemenet, kimenet párhoz csinálok egy Copy metódust, valahogy így:

```

class Copy {

    public void CopyCP() { while( (char c = Console.ReadKey()) != EOF) Printer.PrintChar(c); }

    public void CopyCT(Text out) { while( (char c = Console.ReadKey()) != EOF) out.WriteChar(c); }

    public void CopyTP(Text in) { while( (char c = in.ReadChar()) != EOF) Printer.PrintChar(c); }

    public void CopyTT(File in, File out) { while( (char c = in.ReadChar()) != EOF) out.WriteChar(c);}

}

```

Mint látható, a fő üzleti logika itt nagyon sokszor ismétlődik, a bemenetről másolok a kimenetre, amíg file vége jelet nem olvasunk, csak más-más bemenettel, kimenettel. Nyilván ez a megoldás nem elég száraz, ahol a nem elég száraz (angolul: dry) szófordulat alatt azt értjük, hogy feleslegesen ismételtjük magunkat, azaz megsértjük a ne ismételd magad (angolul: Don't Repeat Yourself – DRY) elvet.

Érdekes kérdés, hogy ez a megoldás megfelel az OCP elvnek? Ez attól függ, hogy az OCP-t milyen tágan értelmezem. Az OCP két lehetséges definíciója:

- OCP bő értelmezése: A program komponensei (azaz az osztályok, és a rétegek) legyenek zártak a módosításra, de nyíltak a bővítésre, azaz új metódust és új alosztályt adhatok a kódomhoz, de meglévő, kitesztelt metódust nem írhatok át. Ezért
 - o nem használhatjuk az override kulcsszót csak absztrakt és horog (angolul: hook) metódusok átírására, mert absztrakt metódusnak nincs törzse, azaz csak bővítjük a kódunkat, nem átírjuk, illetve, mert horog metódusoknak habár van törzse, de az üres (vagy csak egy alapértelmezett értéket ad vissza), így ez megint csak bővítés, nem módosítás. Tovább:
 - o nem tanácsos if – else if szerkezetet használni, mert azt előbb utóbb egy új ággal kell bővítenem, de ez a befoglaló metódus szemszögéből módosítás. Csak akkor használjunk if – else if szerkezetet, ha biztosak vagyunk benne, hogy a jövőben nem kell új ág.
- OCP szűk értelmezése: A program komponensei (azaz az osztályok, és a rétegek) legyenek zártak a módosításra, de nyíltak a bővítésre, azaz ~~(új metódust és)~~ új alosztályt adhatok a kódomhoz, de meglévő, kitesztelt metódust nem írhatok át, meglévő, kitesztelt osztályt nem írhatok át, még új metódust sem adhatok hozzá. Azaz:
 - o nem használhatjuk az override kulcsszót csak absztrakt és horog (angolul: hook) metódusok átírására, mert absztrakt metódusnak nincs törzse, azaz csak bővítjük a kódunkat, nem átírjuk, illetve, mert horog metódusoknak habár van törzse, de az üres (vagy csak egy alapértelmezett értéket ad vissza), így ez megint csak bővítés, nem módosítás. Tovább:
 - o nem tanácsos if – else if szerkezetet használni, mert azt előbb utóbb egy új ággal kell bővítenem, de ez a befoglaló metódus szemszögéből módosítás. Csak akkor használjunk if – else if szerkezetet, ha biztosak vagyunk benne, hogy a jövőben nem kell új ág. Tovább:
 - o meglévő, kitesztelt osztályhoz, aminek a felülete már kiforrott, ahhoz nem adhatunk új metódust, főleg nem publikus metódust, mert ezzel megváltoztatjuk az osztály felületét, ráadásul az új metódus új állapotátmeneteket is eredményezhet, amit az

eddig állapotgép nem tartalmazott. Esetleg új privát metódust hozzáadhatok, ha ezt csak kódszépítés miatt teszem, azaz, mondjuk ismétlődő kódrészletet emelek ki.

Mint látható az OCP bő értelmezésébe belefér, hogy új metódussal bővítsünk egy osztályt. A szűk értelmezésben már az osztályok felületét is óvjuk, új publikus metódust nem adhatunk az osztályhoz. A szűk értelmezésnek is vannak változatai, az egyik értelmezésben az osztály felületet óvjuk. Egy másik értelmezésben pedig az osztály állapotgépét. Mivel az OCP bő értelmezése sokkal elterjedtebb, ezért a továbbiakban ezt fogjuk használni.

Tehát, igen, ez a változat megfelel az OCP-nek, mert csak új metódusokkal kell bővítenem egy meglévő osztályt, ha bejön egy újfajta bemenet vagy kimenet.

Vegyük észre, hogy ez a kód még mindig a részletektől függ. Ha kiderül, hogy van egy új kimenet, vagy bemenet, amit támogatni kell, akkor (négyzetesen emelkedő számú) új metódusokat kell írnom, csak amiatt, hogy valami piszkos fizikai részlet megváltozott.

A harmadik lehetőség, hogy feltételezem, hogy minden forrásnak ugyanaz a felülete, mondjuk: `ReadByte()`, és minden kimenetnek ugyanaz a felülete, mondjuk `WriteByte()`. Ha ez igaz, akkor csak a logikát kell leprogramoznom, miszerint:

„A `Copy()` metódus fontos logikát ír le, a forrásból a kimentre kell másolni karaktereket file vége jelig. Ezt a logikát sok helyen fel lehet használni, hiszen a forrás bármi lehet, bármi, ami karaktereket tud beolvasni, illetve a kimenet is bármi lehet, bármi, ami karaktert tud kiírni.”

Szerencsére sok nyelven van ilyen felület, pl. Javában és C#-ban a jól ismert `Stream` absztrakt osztályt használva felületnek a következő kód írható:

```
class Source2Sink
{
    private System.IO.Stream source;
    private System.IO.Stream sink;
    public Source2Sink(Stream source, Stream sink)
    {
        this.source = source;
        this.sink = sink;
    }
    public void Copy()
    {
        byte b = source.ReadByte();
        while (b != 26)
        {
            sink.WriteByte(b);
            b = source.ReadByte();
        }
    }
}
```

Vegyük észre, hogy ebben a megoldásban a részletek függenek az absztrakciótól. Ha van egy új kimeneti megoldás, akkor a fenti kódrészletet nem kell változtatni, hanem pont ellenkezőleg, az új kimeneti megoldásnak kell alkalmazkodnia a fenti megoldáshoz és implementálnia kell a `WriteByte(byte b)` metódust.

Azaz a függőség megfordult, nem az absztrakt gondolatot megvalósító kódrészlet függ a változó részletektől, hanem a részletek függenek az absztrakciótól.

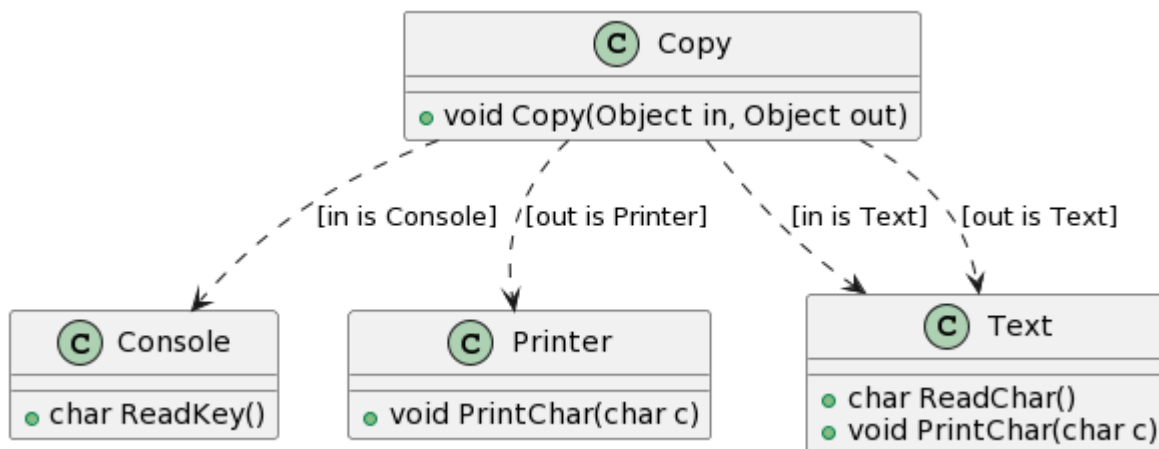
Vegyük észre továbbá, hogy ez a megoldás megfelel az OCP elvnek is, ami mutatja, hogy az OCP központi szerepet játszik az objektumorientált tervezésben.

Persze ez a kódrészlet sem állandó, hiszen a fő elvünk kimondja, hogy: „A kód állandóan változik!”. Azaz, előbb-utóbb minden megváltozik. Például, ha változik az üzleti logika és nem EOF-ig, hanem -1-ig kell olvasni, akkor hozzá kell nyúlnunk ehhez a metódushoz, de ez nem baj, mert amíg a változásnak csak egy oka van, addig az SRP szerint még jó a tervünk. Akkor van gond, ha mondjuk az üzleti logika megváltozása mellett mondjuk az operációs rendszer változása is változást okoz (az EOF kódja mondjuk 255 és nem 26). Még ekkor is mondhatjuk, hogy jók vagyunk, mert egy nehezen előrelátható változás miatt sértettük meg az SRP elvet, és csak refaktorálnunk kell, ami az agilis szoftverfejlesztésnek a sajátja.

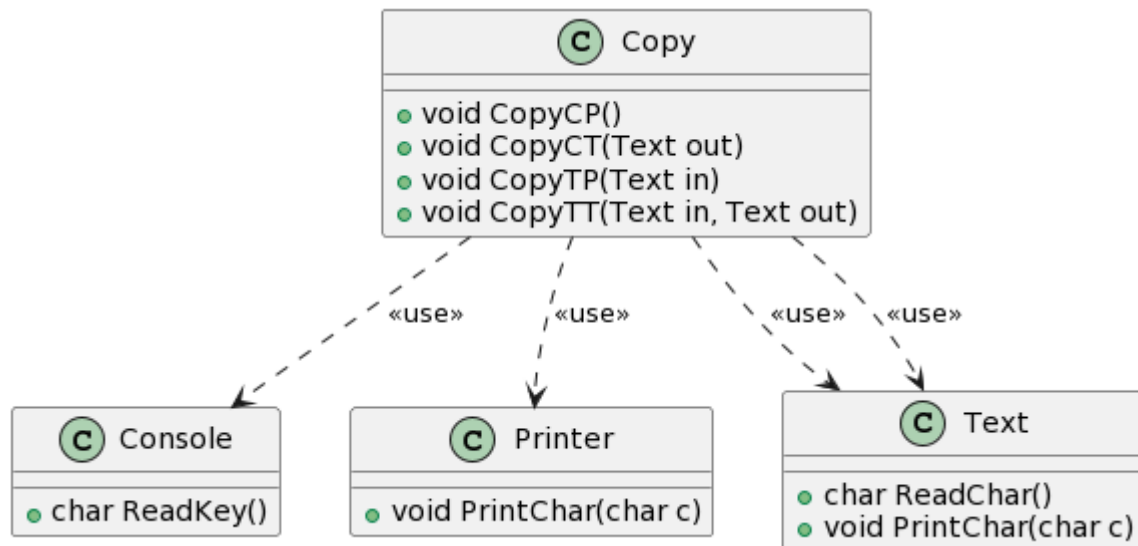
Gond akkor van, ha egy előrelátható, józan paraszti ésszel, rendszerszemlélettel előrelátható ok miatt sértjük meg az SRP. Mondhatnánk, hogy hol itt a gond, hiszen ebben az esetben is csak refaktorálni kell. Ugyanakkor a refaktorálás drága, újra kell értelmezni a kódot, ami idő, az idő pénz, a pénz a megrendelőtől jön, azaz a megrendelő pénzét pazarlom, a helyett, hogy már az elején rugalmas kódot írtam volna. És ez, hogy pazarlom az erőforrásokat, különösen, hogy a megrendelő pénzét pazarlom, ez nem fér bele az agilis szoftverfejlesztésbe.

Rajzoljuk le mindhárom fenti megoldás UML osztály diagramját.

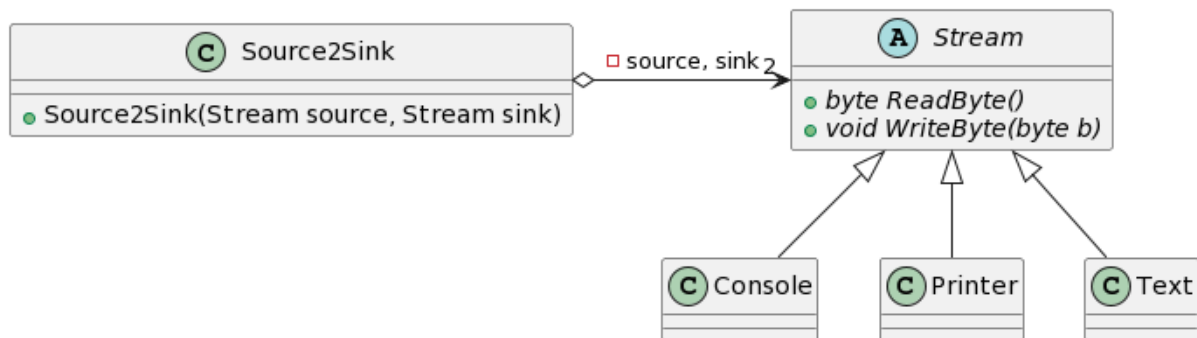
OCP-t megszegő:



DRY-t megszegő:



OCP-t, DRY-t, DIP-et betartó:



A fenti 3 UML osztálydiagram PlantUML szkriptje:

OCP-t megszegő	DRY-t megszegő	OCP-t, DRY-t, DIP-et betartó
<pre> @startuml class Copy { + void Copy(Object in, Object out) } class Console { + char ReadKey() } class Printer { + void PrintChar(char c) } class Text { + char ReadChar() + void PrintChar(char c) } Copy ..> Console : [in is Console] Copy ..> Text : [in is Text] </pre>	<pre> @startuml class Copy { + void CopyCP() + void CopyCT(Text out) + void CopyTP(Text in) + void CopyTT(Text in, Text out) } class Console { + char ReadKey() } class Printer { + void PrintChar(char c) } class Text { + char ReadChar() + void PrintChar(char c) } Copy ..> Console : <<use>> Copy ..> Text : <<use>> </pre>	<pre> @startuml class Source2Sink { + Source2Sink(Stream source, Stream sink) } abstract class Stream { +{abstract} byte ReadByte() +{abstract} void WriteByte(byte b) } class Console {} class Printer {} class Text {} Source2Sink o--> "2" Stream : source, sink Stream < -- Console Stream < -- Printer Stream < -- Text @enduml </pre>

Copy ..> Printer : [out is Printer] Copy ..> Text : [out is Text] @enduml	Copy ..> Printer : <<use>> Copy ..> Text : <<use>> @enduml	
---	--	--

Vegyük észre, hogy a DIP-nek megfelelő ábrán a magas szintű modul HAS-A kapcsolattal birtokol egy, illetve jelen esetben két, Stream példányt. A Stream absztrakt osztály pedig IS-A kapcsolatban van 3 konkrét osztállyal. Nyilván a 3 példa valamennyire kizárja egymást, mert egyszer azt feltételeztük, hogy a Console osztályban a ReadKey() metódus statikus. Egy másik példában meg azt, hogy a Console megvalósítja a Stream felületet. Ugyanakkor a képzeletbeli programozási nyelvben, amit használunk tegyük fel, hogy így van.

A harmadik, DIP-nek megfelelő ábra egy nagyon fontos tulajdonsággal bír: Minden nyíl, a HAS-A kapcsolatok is és az IS-A kapcsolatok is, az absztrakt osztályra mutatnak. Ez nem véletlen, ez a fő tulajdonsága a DIP-nek. Úgy szoktuk mondani, hogy akkor felelünk meg a DIP alapelvnek, ha minden nyíl absztrakcióra (azaz absztrakt osztályra, vagy interfészre mutat). Másszóval, ha a magas és az alacsony szintű modulokat absztrakció (egy absztrakt osztály, vagy interfész) választja el egymástól.

Ez azt jelenti, hogy a DIP a nagyon magas szintű Dolgok Szétválasztása (angolul: Separation of Concerns) alapelv alacsonyabb szintű alakja? Azaz, a nagy véres kard kistervére? Egy kis véres szike? Vagy, maga a nagy véres kard? Erre a kérdésre még visszatérünk.

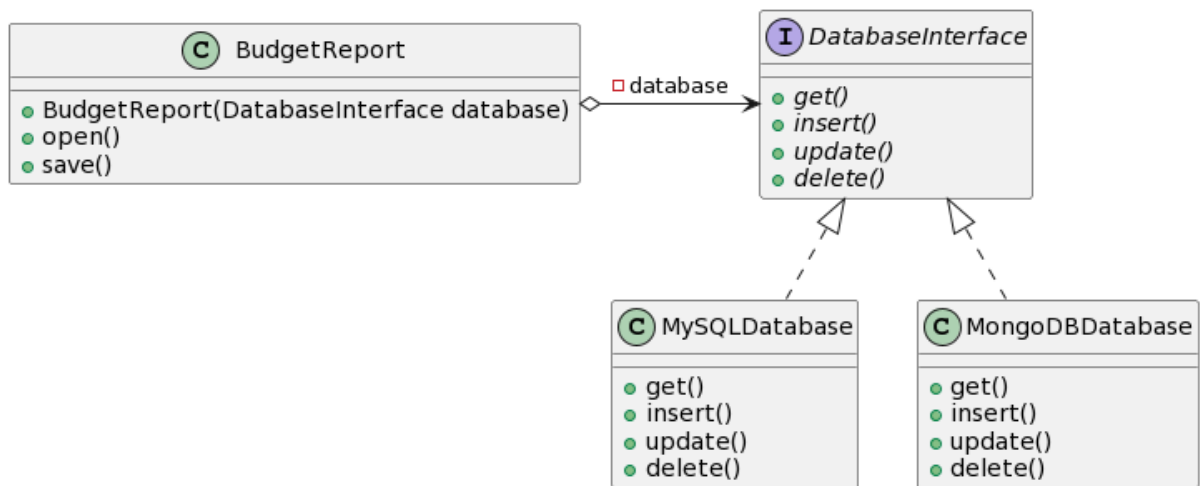
1.1.3. A híres adatbázis-kezelő dilemma

Ebben a részben azt tárgyaljuk, hogy az adatbázis-kezelő választás, habár fontos dolog, de csak egy részlet.

Ha már volt szerencsénk éles projekten dolgozni, akkor biztos volt olyan tapasztalatunk, hogy a vezető fejlesztő, a projekt menedzser, a csoport tapasztaltabb emberei több hétig vakarták a fejüket, hogy milyen adatbázis kezelőt használjunk a projekten. Legyen a jól bevált Oracle, vagy a szintén robusztus Microsoft SQL Server, vagy inkább egy kisebb, de rugalmasabb, mint a MySQL vagy az SQLite, vagy inkább NoSQL és MongoDB? És akkor még nem is beszéltünk az összes nagyobb választási lehetőségről.

Nekünk, akik egyre inkább értjük a DIP elvet, ez egy kicsit feleslegesnek tűnik, hiszen ez csak egy piszkos részlet. Akkor járunk a legjobban, ha nem kötjük magunkat egyik adatbázis-kezelőhöz sem, hanem egy absztrakció mögé bújtatjuk ezt a döntést. Persze, még mindig húsbavágó kérdés, mi legyen az adatbázis-kezelő motor, mert lehet, hogy SQL-ben, azon belül is PL/SQL nagyobb a tapasztalata a csapatnak, mint bármi másban. Ugyanakkor, ha megértjük, hogy ez a választás csak egy piszkos részlet, amit egy absztrakció mögé kell dugni, akkor a programunk többi része nem fog függeni ettől a döntéstől, és sokkal könnyebben levezényelhetünk egy adatbázis-kezelő cserét, ha erre valami miatt esetleg szükség lenne.

Nézzünk egy egyszerű példát egy jó tervre, ami az adatbázis-kezelő választást egy absztrakció mögé rejti:



PlantUML script:

@startuml

interface DatabaseInterface {

+ {abstract} get()

+ {abstract} insert()

+ {abstract} update()

+ {abstract} delete()

}

class MySQLDatabase {

+ get()

+ insert()

+ update()

+ delete()

}

class MongoDBDatabase {

+ get()

+ insert()

+ update()

+ delete()

}

class BudgetReport {

+BudgetReport(DatabaseInterface database)

```
+ open()  
+ save()  
}
```

BudgetReport o-right-> DatabaseInterface : -database

DatabaseInterface <|.. MySQLDatabase

DatabaseInterface <|.. MongoDBDatabase

@enduml

Ebben a példában a BudgetReport egy DatabaseInterface példányt birtokol a database referencián keresztül, ami végső soron vagy egy MySQLDatabase vagy egy MongoDBDatabase példányra mutat. Hogy melyikre, az mindegy, hiszen a kettőnek közös a felülete, és a BudgetReport csak ezt a közös felületet ismeri. Így könnyen válthatunk a két megoldás között, vagy esetleg könnyen hozzáadhatunk egy harmadikat, anélkül, hogy a BudgetReport osztályt változtatni kellene.

Természetesen az alsó szinten ugyanahhoz a híváshoz más-más implementáció tartozik, amit akár nagyon nehéz is lehet lefejleszteni. Végső soron, ezért szoktak annyit rágódni a vezető fejlesztők az adatbázis-kezelő választáson. Ugyanakkor a felső szintet ez a választás már nem kell, hogy érdekelje.

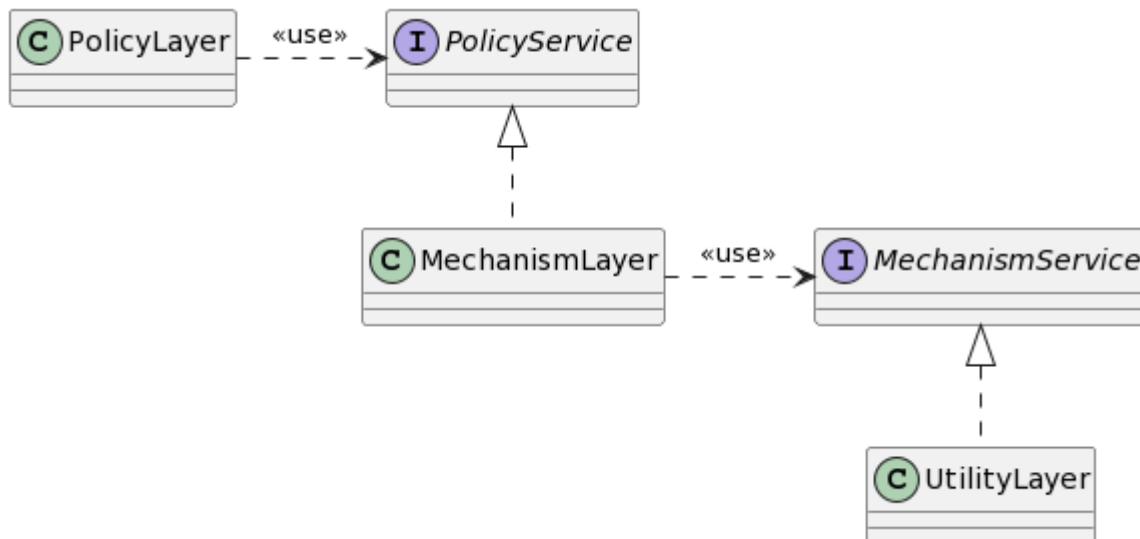
Ebben a példában is azt látjuk, hogy a felső és az alsó szint is az absztrakciótól függ. UML szinten ez azt jelenti, hogy minden nyíl absztrakcióra mutat.

Nagyon gyakori hasonló dilemma, ha mondjuk nagy számokkal kell dolgozunk, akkor melyik nagy szám osztálykönyvtárt (angolul: library) használjuk. Itt is ugyanaz a válasz, ez csak egy piszkos részlet, amit absztrakció mögé kell rejteni.

Itt érdemes még egy kérdést végiggondolni: az újrafelhasználhatóságot. Az alacsony szintű komponensek újrafelhasználása jól megoldott az úgynevezett osztálykönyvtárak (angolul: library) segítségével. Ezekbe gyűjtjük össze azokat a metódusokat, amikre gyakran szükségünk van. A magas szintű komponensek, amik a rendszer logikáját írják le, általában nehezen újrafelhasználhatók. Ezen segít a függőség megfordítása. Ahogy a híres copy példánál is láthattuk.

1.1.4. [A DIP UML osztály diagram ábrája](#)

Nézzük meg a DIP ábráját, azt, amit a Wikipédián találunk. Ez az ábra egy 3 rétegű DIP ábrája.



PlantUML szkriptje:

```

@startuml
package DIP {
    class PolicyLayer {}
    interface PolicyService {}
    class MechanismLayer {}
    interface MechanismService {}
    class UtilityLayer {}
}
PolicyLayer .right.> PolicyService : <<use>>
MechanismLayer .up.|> PolicyService
MechanismLayer .right.> MechanismService : <<use>>
UtilityLayer .up.|> MechanismService
@enduml
  
```

Először is vegyük észre a legfontosabbat: Minden nyíl absztrakcióra (jelen esetben interfészekre) mutat. Ez az a tulajdonság, amiről a legkönnyebb felismerni a DIP-et.

Mit tudunk még megállapítani az ábráról? Három réteget látunk:

- PolicyLayer, ez egy magas szintű rész, ami az üzleti logikát, a poliszit zárja egységbe.
- MechanismLayer, ez a középső szint, ahol a mechanizmusok vannak egységbe zárva.
- UtilityLayer, alacsony, hardver közeli réteg, ahol a hardvert kezelő szerszámoszláda funkcionalitás van egységbe zárva.

A három réteget interfészek választják el egymástól:

- A PolicyLayer és a MechanismLayer közt van a PolicyService.
- MechanismLayer és a UtilityLayer közt van a MechanismService.

A két szervíz interfész azoknak a metódusoknak a fejét tartalmazza, ami a poliszi megvalósításához, illetve a közepes szinten lévő mechanizmusok megvalósításához kell. Ez azért érdekes, mert ez azt jelenti, hogy időben hamarabb találom ki ezeket az interfészeket, mint ahogy a megvalósító osztályokat elkezdem implementálni.

Gyakori hiba, hogy először a kézzel fogható, alacsony szintű osztályokat csináljuk meg, aztán e felé húzunk egy absztrakciót, amibe kiemeljük a publikus metódusok fejeit. Ez a megoldás teljesen életszerű, de sajnos nem használja ki a DIP által ajánlott fordított gondolkozást: Ne a konkrét részletekből induljunk ki, az absztrakcióból induljunk ki.

1.1.5. A HAS-A kapcsolat 3 fajtája és a DIP

Ha újra megnézzük a fenti ábrát, akkor egy érdekességet veszünk észre. Érdekes, hogy nem a jól megszokott HAS-A kapcsolatot, azaz egy rombuszból kiinduló nyilat, hanem egy szaggatott nyilat látunk a magasszintű modul és az interfész közt. Ez nem véletlen, és nem is hiba. Ideje, hogy kibővítsük a HAS-A kapcsolatról eddig felgyűlt tudásunkat.

Eddig úgy tudtuk, hogy a HAS-A kapcsolatnak két fajtája van a kapcsolat erőssége szerint. Ezt a két fajtát egy kérdéssel különböztetünk meg egymástól: „Ha meghal a gitáros, vele temetik a gitárját?” Habár ez a kérdés egy kissé morbid, de könnyen megjegyezhető, ráadásul jól szemlélteti, hogy mit csinál a szemét gyűjtő algoritmus (angolul: Garbage Collector - GC).

A HAS-A kapcsolat, vagy másnéven az objektum összetételnek, eddig két változatát tanultuk a birtoklás erőssége szerint:

- Kompozíció, PlantUML-ben: Gitáros *--> Gitár, „ha meghal a gitáros, akkor vele temetik a gitárját”, azaz a gitár csak a gitárosé, senki másnak nincs referenciája erre a gitárra; ha a gitárost törli a GC, akkor a gitárját is.
- Aggregáció, PlantUML-ben: Gitáros o--> Gitár, „ha meghal a gitáros, akkor nem temetik vele a gitárját, mert a gitár nem csak az övé”, azaz a gitár nem csak a gitárosé, van másnak is referenciája erre a gitárra; ha a gitárost törli a GC, akkor a gitárját még nem törli, mert a referencia számláló még nem nulla, hiszen másnak is van rá referenciája.

Ezen túl van egy harmadik, nagyon gyenge HAS-A kapcsolat, a barátság (angolul: friendship), amit néha asszociációnak is nevezünk. A barátságnak két nyíl is megfelel UML-ben, a sima nyíl (PlantUML-ben: -->) és a szaggatott nyíl (PlantUML-ben: ..>). A sima nyílra nem szoktunk írni semmit, legfeljebb a kapcsolatot megvalósító mező nevét. A szaggatott nyílra viszont ráírjuk, hogy milyen fajta barátságról van szó:

- A Gitáros híja a Gitár valamely metódusát: <<use>>
- A Gitáros hozza létre a Gitárt és vagy használja, vagy nem: <<create>>
- A fenti két gyakori eseten kívül mindenféle elképzelhető: <<call>>, <<build>>, <<notify>>, ..., ugyanakkor ezeket már inkább szekvencia diagrammon jelöljük, nem osztály diagrammon.

A barátság minden esetben átmeneti kapcsolatot jelöl. A Gitáros nem Gitárral a kezében születik, de élete során szerezhethet egyet, eladhatja, vehet egy másikat, akár össze is törheti. Aggregáció és kompozíció esetén mindig van egy mező, ami megvalósítja a HAS-A kapcsolatot (a Gitáros osztályban van például egy „Gitár kedvencGitár;” nevű mező), és ez a mező nem lehet null értékű. Barátság esetén nem muszáj, hogy legyen mező, de ha van is, akkor az lehet null értékű. A barátság legtipikusabb megjelenése, hogy egy metódus paramétereként kapom meg a Gitárt, azt használom, aztán el is felejttem. Ugyanakkor az is barátság, ha egy setGitár(Gitár g) metóduson keresztül megkapom a gitárt, megjegyzem egy mezőben, de ez a mező eddig null értékű volt, és akár újra lehet null értékű, azaz a gitárt nem a konstruktorban kapom meg.

Tehát a HAS-A kapcsolat 3 fajtája a birtoklás erőssége szerint:

- Kompozíció: Állandó tulajdonviszony (persze lecserélhető), valószínűleg születésemtől fogva megvan, és csak az enyém. Egy mező valósítja meg a kapcsolatot. Gyakran nem lehet null értékű ez a mező.
- Aggregáció: Állandó tulajdonviszony (persze lecserélhető), valószínűleg születésemtől fogva megvan, de nem csak az enyém. Egy mező valósítja meg a kapcsolatot. Gyakran nem lehet null értékű ez a mező. Itt érdemes megemlíteni, hogy az aggregáció szó informatikán belül más jelentéssel is bír. Adatbázis kezelés esetén, ha egy egység több részegységből áll össze, akkor ezt kifejezhetjük az aggregáció szóval is: egy egység több részegység aggregátuma. Tehát a rész-egész típusú kapcsolatok leírására is használható az aggregáció szó.
- Barátság vagy asszociáció: Átmeneti tulajdonviszony, valószínűleg nem kapom meg születésemnél. Lehet, hogy egy mező valósítja meg a kapcsolatot, de akkor ez a mező null értékű is lehet. Gyakrabban csak egy metódus egy paramétere valósítja meg a kapcsolatot.

Hogy melyik HAS-A kapcsolatot használjuk a DIP UML ábráján, az a DIP szemszögéből mindegy. Ha egy nagyon általános DIP ábrát akarunk rajzolni, akkor a barátságot használjuk, hiszen, az adja a legkevesebb kötöttséget. Viszont, ha egy konkrét feladat UML ábráját akarjuk elkészíteni, akkor gyakran aggregációt, vagy kompozíciót kell használnunk.

Ugyanakkor DIP nincs HAS-A kapcsolat nélkül. Azaz a DIP megvalósításához kell valamilyen fajta HAS-A kapcsolat. A HAS-A kapcsolatot megvalósító referencián keresztül érjük el azokat a szolgáltatásokat, amik a magas szintű logika megvalósításához kellenek. Ez a referencia lehet egy mező, vagy egy sima változó. Ugyanakkor a referencia mögött álló objektumot nem ismerjük, csak a felületét, azaz egy API-n keresztül hívjuk a szolgáltatásokat. Ezt az API-t zárja egységbe az az absztrakció, amire a HAS-A kapcsolat mutat.

Szűkebb értelemben az API csak a hívható metódusok fejét és leírását tartalmazza. Tágabb értelemben a metódusok szerződését is tartalmazza. A szűkebb értelmez a GOF1 betartását segíti. A tágabb értelmezés a GOF1 és az LSP betartását is segíti.

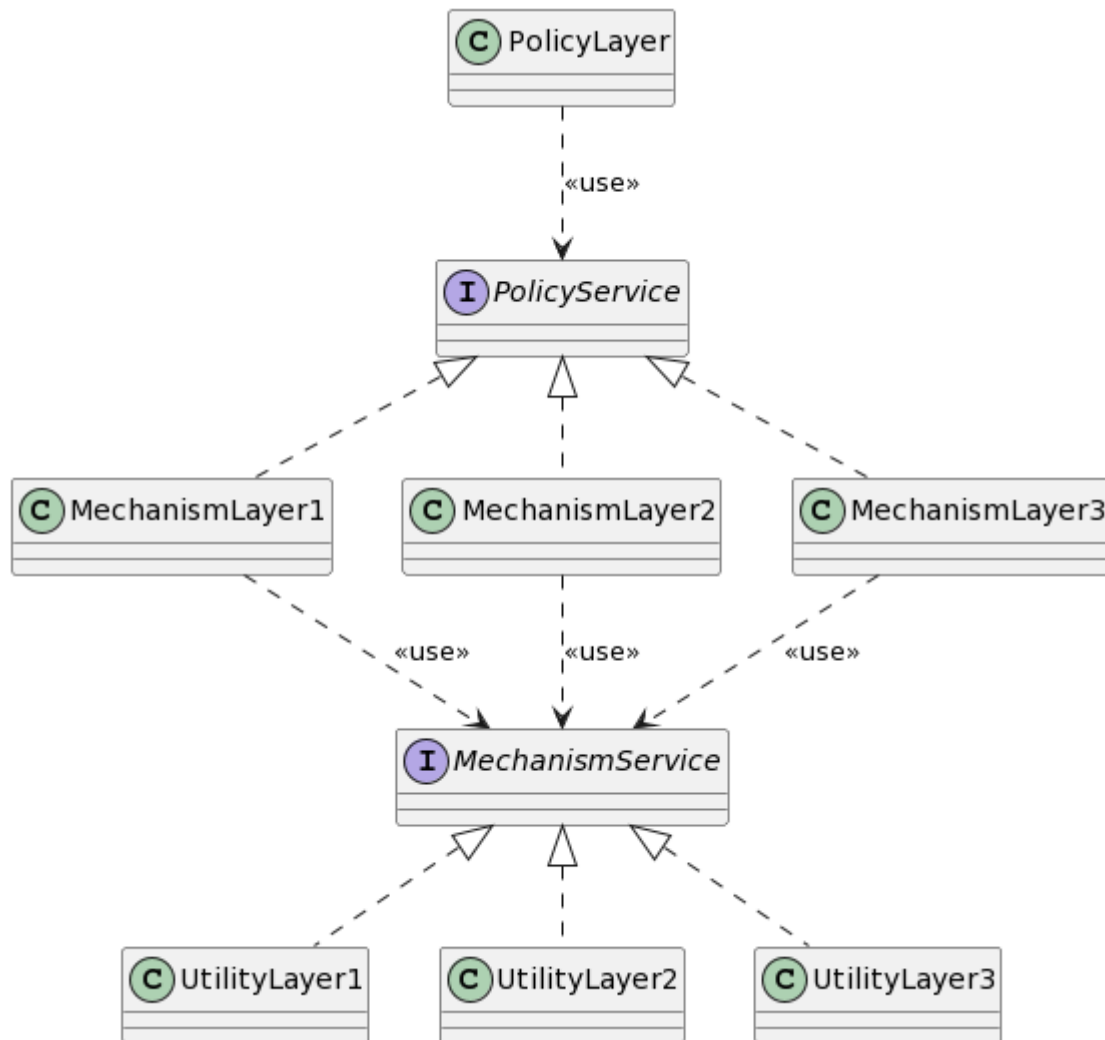
HAS-A kapcsolat esetén, hacsak nem a példányon belül hozzuk létre a kapcsolatot megvalósító referencia értékét, ezt az értéket valahogy be kell juttatni példányba. Ezt nevezzük felelősség beinjektálásának, vagy röviden felelősség injektálásnak, vagy még rövidebben injektálásnak. Az injektálás az a kódrészlet, ahol a kapcsolatot megvalósító referencia értéket kap kívülről. A felelősség injektálásának több típusa is létezik:

- Felelősség injektálása konstruktorral: Ebben az esetben az osztály a konstruktorán keresztül kapja meg azokat a referenciákat, amiken keresztül a neki hasznos szolgáltatásokat meg tudja hívni. Ezt más néven objektum-összetételnek is nevezzük és a leggyakrabban épp így programozzuk le.
- Felelősség injektálása szetter metódusokkal: Ebben az esetben az osztály szetter metódusokon keresztül kapja meg azokat a referenciákat, amikre szüksége van a működéséhez. Általában ezt csak akkor használjuk, ha opcionális működés megvalósításához kell objektum-összetételt alkalmaznunk.
- Felelősség injektálása interfész megvalósításával. Ha a példányt a magas szintű komponens is elkészítheti, akkor elegendő megadni a példány interfészét, amit általában maga a magas szintű komponens valósít meg, de paraméterosztály paramétereként is jöhet az interfész.
- Felelősség injektálása elnevezési konvenció, konfigurációs állomány, vagy annotáció alapján. Ez általában keretrendszerekre jellemző. Ezeket csak tapasztalt programozóknak ajánljuk,

mert nyomkövetéssel nem lehet megtalálni, hogy honnan jön a példány és ez nagyon zavaró lehet.

1.1.6. A 3 rétegű DIP

Hogy még jobban megértsük a DIP-et, nézzünk egy olyan 3 rétegű megvalósítást, ahol az egyes rétegekben több osztály is van:



PlantUML szkript:

```
@startuml
```

```
PolicyLayer ..> PolicyService : <<use>>
```

```
MechanismLayer1 .up.|> PolicyService
```

```
MechanismLayer2 .up.|> PolicyService
```

```
MechanismLayer3 .up.|> PolicyService
```

```
MechanismLayer1 ..> MechanismService : <<use>>
```

```
MechanismLayer2 ..> MechanismService : <<use>>
```

```
MechanismLayer3 ..> MechanismService : <<use>>
```

UtilityLayer1 .up.|> MechanismService

UtilityLayer2 .up.|> MechanismService

UtilityLayer3 .up.|> MechanismService

@enduml

Mint látjuk, ebben a példában is minden nyíl az absztrakcióra mutat. Ez nagyon jellemző a DIP-re.

Mint látjuk, a középső és alsó rétegben több konkrét megvalósítás közül is választhatunk. Az Autós példánál a középső rétegben lehet a sima 5 sebességes megvalósítás, illetve az automataváltós megvalósítás. Az alsó rétegben a konkrét autók lehetnek. De miért van a legfelső Poliszí rétegben csak egy? Ha az a főprogram, akkor ez így jó. Ha a Poliszí réteg felett van még a főprogram, akkor ebben a rétegben is lehet több konkrét megvalósítás.

Ez az ábra nagyon jól mutatja a DIP értelmét. Ha nem lenne DIP, akkor mindhárom középső réteg béli mechanizmusnak mindhárom alsó szintű szerszámoszládaival képesnek kellene lennie együttműködni. Ez vagy nagy if - else if szerkezeteket feltételez, mint ahogy az egyik Copy példában láttuk, vagy legrosszabb esetben 3*3 osztályt, vagy ugyanennyi ismétlődő logikát tartalmazó metódust, ahogy egy másik Copy példában láttuk. Ugyanakkor, így, hogy a rétegeket szépen elválasztottuk egy-egy absztrakcióval, így büntetlenül adhatunk hozzá újabb és újabb osztályokat az egyes rétegekhez.

1.1.7. Dolgok szétválasztása, Kontroll megfordítása, Függőség megfordítása

A Dolgok szétválasztásának elve (angolul: Separation of Concerns – SoC) egy nagyon magas szintű elv, amely kimondja, hogy ha valami szét lehet választani, akkor azt érdemes szétválasztani. Ettől alacsonyabb szinten van a Kontroll megfordításának elve (angolul: Inversion of Control – IoC), amely kimondja, hogy „ne a program várjon eseményre, az esemény hívja a programot”. Illetve így is megfogalmazható: „Ha egy döntés elhalasztható, azt halasszuk el.” Programozás szintjén ez lényegében azt jelenti statikus kódkötés helyett (ami fordítási időben feloldható) érdemes dinamikus kódkötést használni (ami csak futási időben oldható fel). Ezt szoktuk az IoC tág értelmezésének nevezni. Ettől is alacsonyabb szinten van a Függőség megfordításának alapelve (angolul: Dependency Inversion Principle – DIP), amely kimondja, hogy az absztrakció nem függhet a részletektől, épp fordítva, a részletek függenek az absztrakciótól.

A könnyebb olvashatóság kedvéért a fenti bekezdést pontokba szedve is megadjuk:

- A Dolgok szétválasztásának elve (angolul: Separation of Concerns – SoC): „Amit szét lehet választani, azt érdemes szétválasztani.”
- A Kontroll megfordításának elve (angolul: Inversion of Control – IoC), tág értelmezése: „Ne a program várjon eseményre, az esemény hívja a programot”, azaz statikus kódkötés helyett érdemes dinamikus kódkötést használni.
- A Függőség megfordításának alapelve (angolul: Dependency Inversion Principle – DIP): „A konkrét kódrészletek az absztrakciótól függenek, az absztrakció nem függjön konkrét részletektől.”

Ez a három elv szorosan összefügg:

- a Dolgok szétválasztása (SoC) teljesen általános, nem programozás specifikus,
- a Kontroll megfordítása (IoC) már programozás specifikus, de nem OOP specifikus,
- a Függőség megfordítása (DIP) már OOP specifikus tervezési alapelv.

Mindhárom elv arra vonatkozik, hogy válasszunk szét dolgokat, ugyanakkor azok maradjanak együttműködésre képesek. Maradjanak együttműködésre képesek, de hogyan? Lássuk a lehetséges válaszokat:

- a Dolgok szétválasztása (SoC) erre a kérdésre semmilyen választ nem ad, hiszen teljesen programozás független elv,
- a Kontroll megfordítása (IoC) erre több lehetőséget is kínál:
 1. OOP esetén: a szülőosztály absztrakt vagy horog (angolul: hook) metódust hív, amelyet a gyermekosztály fejt ki, mint például a sablon metódus tervezési minta esetén, így fordítási időben dől el késői kötés (angolul: late binding) révén, hogy melyik kód fut le;
 2. OOP esetén: függőség beszúrása (angolul: dependency injection) révén kapunk egy referenciát, amelyen keresztül meghívjuk a megfelelő polimorfikus metódust, így fordítási időben dől el késői kötés (angolul: late binding) révén, hogy melyik kód fut le;
 3. funkcionális programozás esetén: ugyanannak a metódusnak megírjuk más-más paraméterlistával is, így a metódus hívásakor minta illesztéssel (angolul: pattern matching) dől el futási időben, hogy melyik metódus fut le (megjegyzés: ez a módszer használható OOP esetén is, pl. a Látogató (angolul: Visitor) tervezési minta is ezen alapszik, de ennek a módszernek a használata kevésbé elterjedt OOP esetén);
 4. teljesen általános esetben: egy esemény hatására meghívjuk az eseményre feliratkozott metódusokat, mivel futási időben lehet fel- és leiratkozni, ezért szükségszerűen futási időben dől el, mely metódusokat kell futtatni;
 5. teljesen általános esetben: egy állományt, gyakran egy osztályt, beolvasva hozunk létre futtatható kódot és az elindítjuk, a futó kód attól függ, mi volt az állományban;
 6. és még sok más megoldás is elképzelhető, amikor nem egy if -else if szerkezettel választjuk ki, hogy melyik függvény fusson, hanem valahogy máshogy.
- a Függőség megfordítása (DIP) az IoC-nél sokkal konkrétabb választ kínál, használjuk a IoC esetén felsorolt 2. megoldást, de a referencia legyen interfész (vagy absztrakt osztály) típusú, hiszen ez esetben mind a konkrét hívó, mint a konkrét hívott kódrészlet absztrakciótól függ, amit az interfész (vagy az absztrakt osztály) zár egységbe.

Mint látható, itt van egy hierarchia: SoC > IoC > DIP.

Összefoglalva:

- A SoC egy nagyon általános elv, amit nem csak programozás esetén használhatunk sikeresen. Azon az általános megfigyelésen alapszik, hogy egy nehéz problémát könnyebb megoldani, ha kisebb, egyszerűbb részproblémákra bontjuk. Lásd, a mesterséges intelligencia probléma redukciós módszerét.
- Az IoC már programozás specifikus, de csak annyit ír elő, hogy törekedjünk a dinamikus (azaz nem fordítási, hanem futás idejű) kötés használatára. Ezzel tegyük lehetővé a keretrendszer specifikus és a nem keretrendszer specifikus kódrészletek szétválasztását.
- Amíg az IoC egyáltalán nem OOP specifikus, addig a DIP már OOP specifikus és elég konkrét megoldás kínál: egy UML osztálydiagrammon minden nyíl mutasson absztrakcióra. Ezzel érjük el, hogy a magasszintű (keretrendszer specifikus) kódrészleteket és az alacsonyszintű (nem keretrendszer specifikus) kódrészleteket szétválasszuk, és arra kényszerítsük mindkét oldalt,

hogy egy interfészekben / absztrakt metódusokban megfogalmazott API leírásoknak feleljenek meg.

A fenti hierarchia megismerése után tudunk válaszolni egy előző alfejezetben feltett kérdésre: „A DIP a nagyon magas szintű Dolgok Szétválasztása elv alacsonyabb szintű alakja? Azaz, a nagy véres kard kistervére? Egy kis véres szike?”. A válasz: Igen, a DIP a Dolgok szétválasztásának egy konkrét esete. Igen: A DIP a kis véres szike. Egy nagyon hasznos aleset, amit OOP területén szoktunk használni!

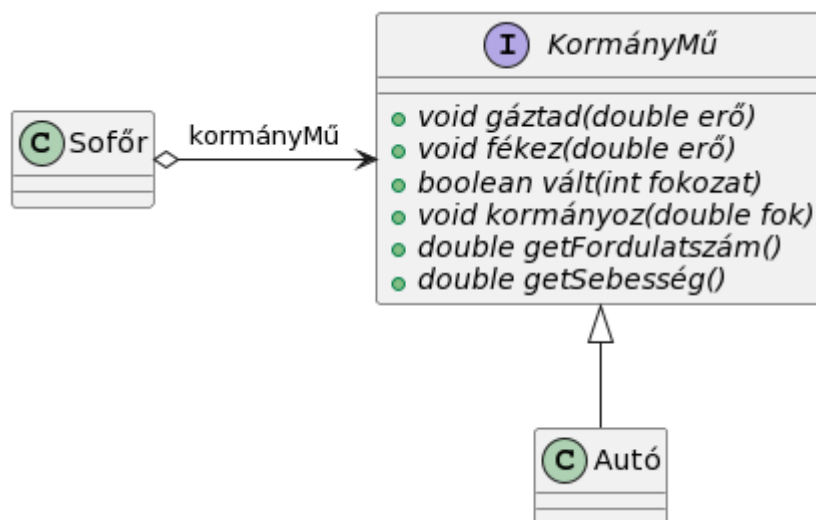
Még itt lehet érvelni, hogy a DIP nem feltétlenül OOP specifikus, ami annyiban igaz is, hogy csak absztrakció középpontba helyezését írja elő. Ugyanakkor OOP területén nagyon széles körben elterjedt, elismert alapelv, ami valószínűleg annak köszönhető, hogy az egyik fő OOP alapelv épp az absztrakció, így a két módszer természetesen illik egymáshoz.

Fontos megjegyzés: A fenti leírásban az IoC tág értelmezését használtuk. A jegyzet többi részében, hacsak ezt nem hangsúlyozzuk ki, az IoC szűk értelmezését használjuk, azaz az IoC 1. megoldását, azaz amikor egy nem absztrakt metódus absztrakt vagy horog metódust hív.

1.1.8. Kommunikáció a felső rétegből az alsó réteg felé

A DIP felépítéséből adódik, hogy mindig a felső szint kezdeményezi és irányítja a kommunikációt, az alsó szint csak végrehajt. Az autós példánál maradva, persze fontos a motor fordulatszáma, de csak akkor, ha ránéz a sofőr a megfelelő kijelzőre, azaz meghívja a `getFordulatszám()` metódust. Ez is elősegíti azt, hogy az alsó szint könnyen változtatható, cserélhető legyen. Ellenkező esetben például egy hardver megváltozásakor a felső szinten is változásokat kellene eszközölni.

A felső szintnek el kell küldenie azokat az adatokat az alsó szintnek, amik az alsó szint működéséhez szükségesek. Az autós példánál maradva, ilyen lehet, hogy hol áll a kormánykerék, a gázpedál, a fék, stb... Ebben az egyszerű esetben csak kell egy referencia, amin keresztül hívjuk az alsó szintet, persze az alsó és a felső szintet elválasztjuk egy absztrakcióval. Ezt az egyszerű esetet szemlélteti az alábbi példa:



PlantUML szkriptje:

```
@startuml
class Sofőr {}
interface KormányMű {
```

```

+{abstract} void gáztad(double erő)
+{abstract} void fékez(double erő)
+{abstract} boolean vált(int fokozat)
+{abstract} void kormányoz(double fok)
+{abstract} double getFordulatszám()
+{abstract} double getSebesség()
}

class Autó {}

```

Sofőr o-right-> KormányMű : kormányMű

Autó -up-|> KormányMű

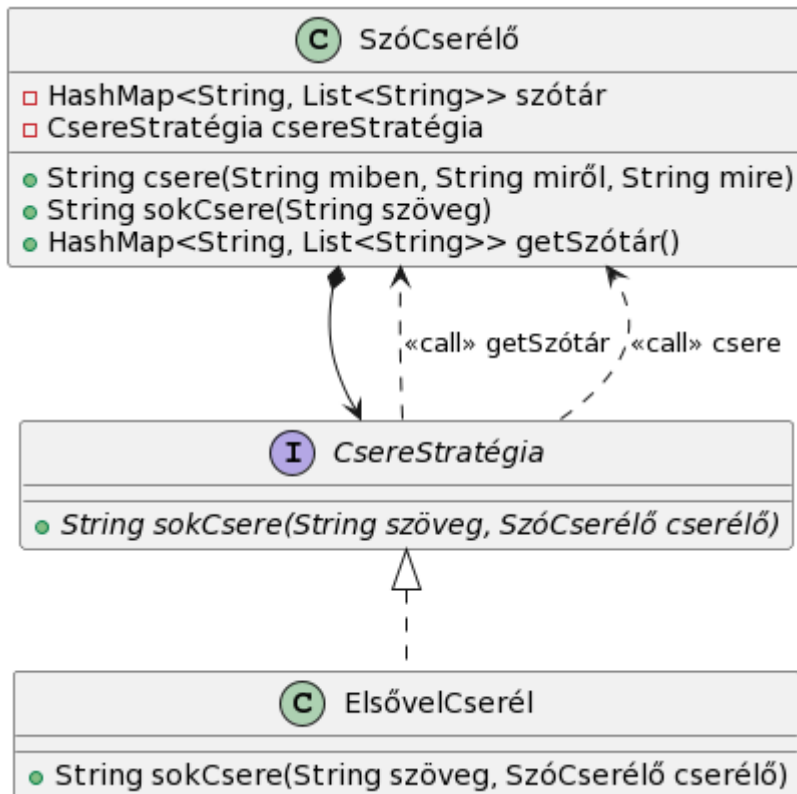
@enduml

Ebben az egyszerű példában a Sofőr a kormányMű referencián keresztül tudja hívni a KormányMű interfészben felsorolt szolgáltatásokat, amiket az Autó osztály valósít meg. Általában a felső szint ilyen egyszerű módon ad parancsokat az alsó szintnek. Ha gázt ad, akkor az autó gyorsul. Mint látjuk, van néhány visszajelzésre szolgáló metódus, pl. a getSebesség metódus, de hogy ezt meghívja-e a Sofőr vagy sem, az már a Sofőr implementációján múlik. Az alsó szint persze adhat riasztást, amit a felső szint már nem hagyhat figyelmen kívül, de erről egy kicsi később írunk.

1.1.9. Egy tervezési hiba: A körkörös kommunikáció

Nézzünk egy másik példát, amikor első megközelítés szerint körkörös kommunikációra lenne szükség a felső és az alsó szint közt. A felső szintnek is hívni kell az alsó szinten megvalósított metódust és az alsó szintnek is a felső szinten megvalósított metódust. Ez a körkörös kommunikáció. Ez egy tipikus tervesi hiba. Erre mutatunk ebben az elfejezetben példát. Egyben megmutatjuk, hogyan kell a körkörös kommunikációt feloldani DIP segítségével.

Tegyük fel, hogy különböző stratégiák mentén helyettesítünk egy szövegben szavakat más szavakra. Ekkor az alsó szinten, ahol megvalósítjuk ezeket a stratégiákat, ismerni kell a szócsere-adatbázist. Általában ez egyszerű, csak paraméterként átadjuk a szükséges adatokat a hívott metódusnak. Néha persze nem is olyan egyszerű, mert a logika előírhatja, hogy a szócsere a felső szinten legyen, ugyanakkor a szócsere stratégiáját az alsó szinten kell meghatározni. Néha ehhez elég egy egyszerű függvényvisszatérési érték, de néha az alsó szintnek ismernie kell a felső szint referenciáját. Ez körkörös tervet eredményez, ami tervezési hibára utal. Nézzük egy ilyen körkörös tervet:



PlantUML szkriptje:

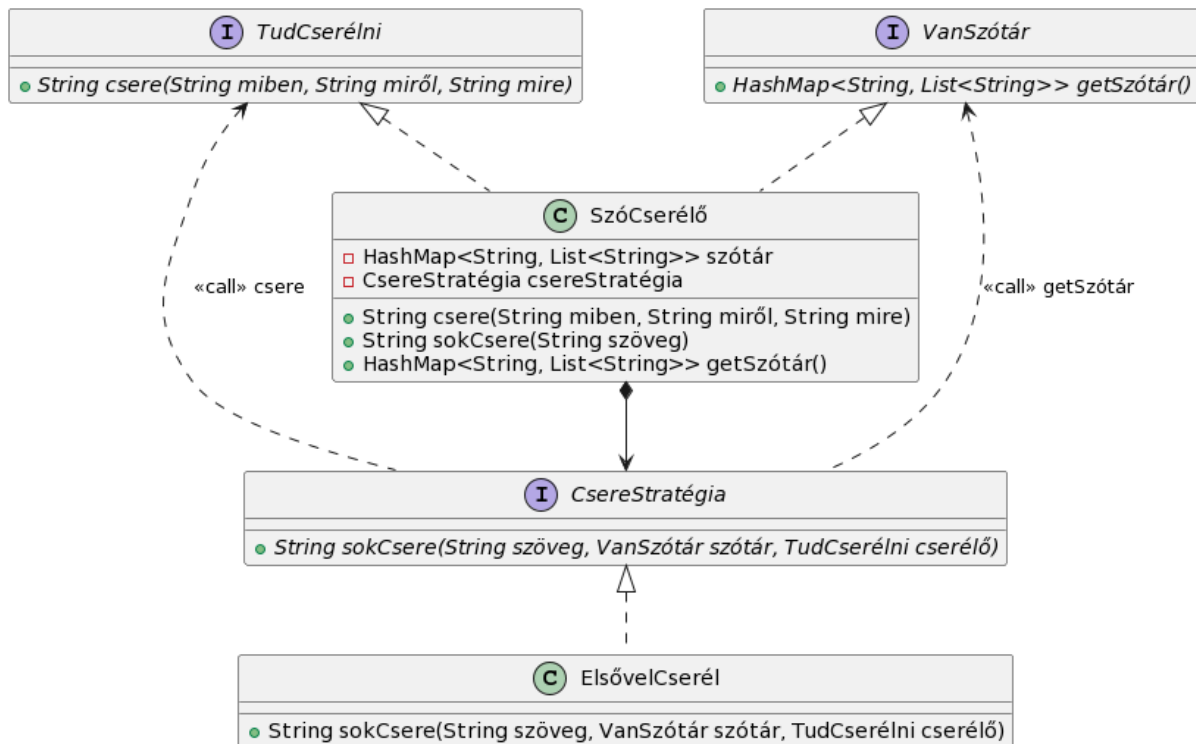
```

@startuml
class SzóCserélő {
- HashMap<String, List<String>> szótár
- CsereStratégia csereStratégia
+ String csere(String miben, String miről, String mire)
+ String sokCsere(String szöveg)
+ HashMap<String, List<String>> getSzótár()
}
interface CsereStratégia {
+ {abstract} String sokCsere(String szöveg, SzóCserélő cserélő)
}
class ElsővelCserél {
+ String sokCsere(String szöveg, SzóCserélő cserélő)
}
SzóCserélő *--> CsereStratégia
CsereStratégia <|.. ElsővelCserél
CsereStratégia ..> SzóCserélő : <<call>> getSzótár
CsereStratégia ..> SzóCserélő : <<call>> csere
@enduml
  
```

Már első ránézésre is látszik, hogy itt nincs minden rendben, hiszen nem minden nyíl mutat absztrakcióra. A CsereStratégia interfészből induló két <<call>> nyíl is konkrét osztályba mutat. Itt külön érdemes lenne megvitatni, hogy egy interfészből hogyan indulhat <<call>> nyíl, de fogadjuk el, hogy ez a szerződés része: A sokSzóCsere metódust a getSzótár és a csere metódusok hívásával kell megvalósítani.

Ugyanakkor a baj az ábrával nem csak az, hogy nem minden nyíl absztrakcióra mutat. Sokkal nagyobb baj, hogy körkörös hivatkozás van: A SzóCserélő osztály mutat a CsereStratégia-ra, és fordítva, a CsereStratégia interfész mutat a SzóCserélő osztályra. A körkörös hivatkozás általában tervezési hibára utal. Nehezen átlátható, szükségtelen komplexitást hordoz, felesleges fordítási függőséget okoz.

Mindig, ha látunk egy körkörös hivatkozást, próbáljuk azt feloldani a DIP alkalmazásával, azaz absztrakt osztályok, vagy interfészek bevezetésével. A fenti példa így javítható ki:



PlantUML szkriptje:

```

@startuml
interface VanSzótár{
+ {abstract} HashMap<String, List<String>> getSzótár()
}
interface TudCserélni{
+ {abstract} String csere(String miben, String miről, String mire)
}
class SzóCserélő {
- HashMap<String, List<String>> szótár
- CsereStratégia csereStratégia
+ String csere(String miben, String miről, String mire)
+ String sokCsere(String szöveg)
+ HashMap<String, List<String>> getSzótár()
}
interface CsereStratégia {
+ {abstract} String sokCsere(String szöveg, VanSzótár szótár, TudCserélni cserélő)
}
class ElsővelCserél {
+ String sokCsere(String szöveg, VanSzótár szótár, TudCserélni cserélő)
}

```



```

VanSzótár <|.. SzóCserélő
TudCserélni <|.. SzóCserélő
SzóCserélő *--> CsereStratégia
CsereStratégia <|.. ElsővelCserél
CsereStratégia .up.> VanSzótár : <<call>> getSzótár
CsereStratégia .up.> TudCserélni : <<call>> csere
@enduml

```

Mint látható, minden nyíl absztrakcióra mutat, azaz sikerült betartanunk a DIP-et. Nincs körkörös hivatkozás sem. Ez a terv már jó! Ráadásul most már nem lenne muszáj kiírni a <<call>> sztereotípiát sem, hiszen, amire mutatunk azok interfészek és mindegyik csak egy-egy metódust tartalmaz. Mindenesetre a jobb értehetőség miatt meghagytuk a <<call>> sztereotípiát.

Most gondoljuk meg, hogy fog kinézni a sokCsere metódus törzse:

```

public String sokCsere(String szöveg) {
    return csereStratégia.sokCsere(szöveg,this,this);
}

```

Mint látható, a második és a harmadik paraméter is this. Egyszer VanSzótár típusként, egyszer pedig TudCserélni típusként.

Mivel ezek az interfészek csak egy-egy metódust tartalmaznak, ezért akár használhatjuk a négypont (::) operátort is, hogy teljesen világos legyen, melyik metódus meghívására adunk itt lehetőséget a stratégián belül:

```

public String sokCsere(String szöveg) {
    return csereStratégia.sokCsere(szöveg,this::getSzótár,this::csere);
}

```

A négypont operátor segítségével egy a szignatúrának megfelelő metódust választhatunk ki az objektumból. Java esetén általában stream alapú kollekció kezelésnél szoktuk a négypont operátort használni.

Ha egy kicsit közelebbről megnézzük a SzóCserélő osztály deklarációját:

```

class SzóCserélő implements VanSzótár, TudCserélni

```

akkor látjuk, hogy ez egy szép példa az interfész szegregációs alapelvre. minden kliens csak egy szűk felületen keresztül látja a példányt, olyan szűk felületen keresztül, amely csak azokat a metódusokat tartalmazza, amire a kliensnek szüksége van. Lényegében ezzel oldjuk fel a fordítási függőséget, ami a körkörös példában volt.

1.1.10. Kommunikáció az alsó rétegből a felső réteg felé, hibajelentések

A DIP felépítéséből adódik, hogy mindig a felső szint kezdeményezi és irányítja a kommunikációt, az alsó szint csak végrehajt. Ugyanakkor az alsó szintnek is kommunikálnia kell a felső szinttel, hiszen ha ez nem így lenne, akkor gyakran olvashatnánk az újságban, hogy újra felrobbant egy pálinkafőzde a túl magas hőmérséglett és nyomás miatt. Ugyanakkor ilyen nagyon-nagyon ritkán olvasunk. Nyilván azért, mert a pálinkafőző mester állandóan ellenőrzi a nyomást, a hőmérségletet, illetve biztonsági szabályozási köröket tartalmaz a rendszer. Ez utóbbi témával foglalkozik az irányítástechnika témaköre, ami sok tankönyv mint az alacsony szintű programozás példáját tárgyalja, ugyanakkor mi az OOP szemszögből vizsgáljuk majd.

Hogyan jöhetünk arra rá, melyik a felső szintű része a programnak? A felső szinten történik az inputkezelés, míg az alsó szint majdnem minden esetben az adatbázis-kezelését végzi (ez alól csak az adatbázis-kezelő programok a kivételek).

Mindkét szint egyaránt végez hibakezelést. A hardverhibákért az alsó, az inputhibákért a felső szint felelős. De mi történjen akkor, hogyha az alsó szinten hiba történik, amiről a felső szintnek tudnia kellene? Ha mindig a felső szint irányítja a kommunikációt, akkor az alsó szint nem küldhet üzenetet a felső szintnek? A felső szint hogyan értesül róla, hogy az alsó szinten kezelendő hiba, vagy rendellenes működés történt? Ez a DIP egyik nagy hátulütője, a felső szint az alsó szintnek csak parancsokat osztogat, de ha nincsennek beépített szabályozási körök, akkor nem feltétlenül figyel az alsó szintre.

Nézzük milyen lehetséges megoldások vannak arra, hogy a felső szintnek muszáj legyen az alsó szint üzeneteivel foglalkozni:

1. A felső szint egy callback metódust ad az alsónak, amin keresztül az hiba esetén értesítheti.
2. Megfigyelő (angolul: Observer) tervezési mintát használva a felső szint feliratkozik az alsó szint megfigyelői közé, ami hiba esetén az értesíti a felső szintet.
3. Az felső szinten egy try-catch blokkban hívjuk meg az alsó szintet. Az alsó szint kivételek kiváltásával értesíti a felső szintet a hibáról.
4. A felső szint önmagára mutató referenciát ad az alsó szintnek, amin keresztül az alsó szint jelzi a hibát a felső szintnek.
5. Az alsó szint egy hibakódot ad vissza. ?????

Mit csinál a felső szint, ha az alsó szinten hiba történik?

- Újra kiadja a parancsot.
- Tranzakció kezelést használ, sikertelen tranzakció esetén rollbacket végzünk.
 - o A tranzakció egy parancssorozat, mely visszavonható parancsokból áll.
 - o A Rollback azt jelenti, hogy a kiadott parancsokat a kiadásukkal ellentétes sorrendben visszavonjuk.
- A felső szinten megvalósítjuk az alsó szint egy leegyszerűsített változatát, és hiba esetén azt futtatjuk.
- A hiba továbbadható a GUI-ra.

Mindenesetre, bármelyik megoldást is választjuk, vagy azok valamilyen kombinációját, a hibát logolni kell.

Habár ez az alfejezet elég rövid, mégis az elméleti óra mellett haladó gyakorlati órán ez egy központi rész, hogy minden lehetőségre konkrét kódot lássunk, hogyan jelzi a hibát az alsó réteg a felső réteg felé.

1.1.11. [Visszacsatolás, szabályozás, szabályozási kör](#)

Amikor egy informatikai rendszert tervezünk, nagyon gyakran van olyan helyzet, hogy a felső szinten hozott döntések az alsó szinten mért értékektől függenek. Az autós példánál maradva, ha túllépjük a sebességhatárt, akkor lassítani kell, amihez lehet, hogy csak le kell vennünk a gázzal a lábunk, de az is lehet, hogy fékezni kell, vagy esetleg a tempómat beállításán kell változtatni.

Egyre gyakrabban kell robotot terveznünk. A robottal az a baj, hogy a fizikai térben létezik. Ha kiadjuk a kereket forgató léptető motornak, hogy két fordulatot tegyen, aminek hatására 5cm-et kellene előrehaladnia, a valóságban nagyon könnyen előfordulhat, hogy csak 4.9cm-et tesz meg, mert csúszik a kerék, vagy akár 5.1cm-et, mert nem pontosan ismerjük a kerék átmérőjét. Akármilyen is okozza az eltérést, a robot valós fizikai jellemzőit vissza kell mérni, és ha nem ott állunk, nem úgy állunk, ahogy elvileg kellene állnunk, akkor korrekcióra van szükség. Megint ugyanaz a helyzet, a felső szintnek az alsó szinten mért adatok alapján kell megvalósítania a működését.

Az ezt tárgyaló területet irányítástechnikának hívjuk, ami szorosan kötődik a mérnöki tudományokhoz, azon belül is a villamosmérnökök munkaköréhez. Az irányítástechnika feltételezi, hogy amit irányítunk az egy fizikai eszköz. Mi ennél kevésbé vagyunk szigorúak, csak annyit feltételezünk, hogy van felső és alsó szint, az alsó szintnek nem muszáj egy fizikai objektummal kommunikálnia. Ugyanakkor a példákban gyakran lesz autó, robot, vagy akár pálinkafőző. Illetve használjuk az irányítástechnika néhány bevett fogalmát.

Az irányítástechnika a kibernetika része. A kibernetika az informatikának az az ága, amely egy rendszert a rendszer részeiként ír le, a részek közti kommunikációra, interakcióra, egymásra gyakorolt hatásukra koncentrál, ugyanakkor az egyes részek belső felépítését elhanyagolja. A kibernetika általunk jól ismert része az információelmélet, és az öntanuló rendszerek. Kevésbé jól ismert része a szabályozás elmélet, amelyet irányítástechnikának is hívunk, ha a mérnöki megközelítést akarjuk hangsúlyozni.

Az elmélet szerint rendelkezésünkre állnak:

- tevékenységek, programozás esetén: metódusok,
- mérhető értékek, programozás esetén: mezők, a mezők által alkotott belső állapot,
- zavar, amik a rendszert zavarják, általában valamilyen külső hatás, és
- zaj, amik a mérés pontosságát befolyásolják.

Az irányítástechnikának két nagy területe van:

- a szabályozás (angolul: closed-loop control, vagy: feedback control), és
- a vezérlés (angolul: open-loop control, vagy: feedforward control).

A szabályozás esetén van egy szabályozott érték, aminek van előírt értéke. Az értéket mérjük. Ez az érték gyakran eltér az előírttól a zavarok és a zaj miatt. A mért eredmény a visszacsatolás (angolul: feedback), ami alapján szabályozunk, hogy a mért és az előírt érték közti eltérés csökkenjen. Ilyen lehet az autós példa esetén a sebesség, amit gázadásával, vagy fékezéssel szabályozunk, hogy ne lépjük túl a megengedett sebességhatárt. Ekkor:

- a szabályozott érték: az autó sebessége,
- az előírt érték: a megengedett sebességhatár,
- szabályozó tevékenység: gáz adás, vagy fékezés,
- zavar: például a vizes aszfalt miatt a feltételezettől eltérő súrlódás,
- zaj: például a sebességmérő rendszer pontatlansága.

A vezérlés esetén nem a szabályozott értéket, hanem a zavart mérjük, esetleg becsüljük, és az alapján végzünk előreecsatolást (angolul: feedforward), hogy elérjük a kívánt hatást. Tehát vezérlés esetén nem a rendszer belső állapota, hanem a környezet állapota, az úgynevezett „rendszer külső állapota”, alapján hozunk döntést.

Összegezve:

- a szabályozás alapja a rendszer belső állapotának mérése, a visszacsatolás;
- a vezérlés alapja a rendszer külső állapotának mérése, az előreecsatolás.

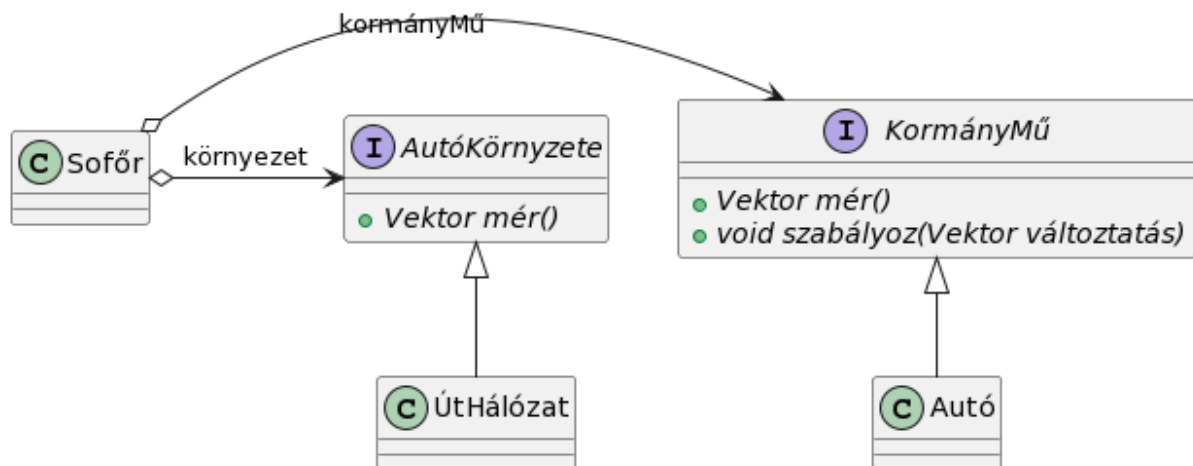
Autós példánál maradva: A sofőr látja, hogy 80-as tábla közeledik (a 80-as tábla, a környezet, a külső állapot része), ezért előreecsatolás módszerével beállítja a tempomat-ot 80 km/óra-ra. Majd, a szokásos rutin keretében nézi a tank töltöttségi szintjét. Látja, hogy a mutató már a piros zónában van. Ez egy

visszacsatolás, aminek hatására úgy dönt, hogy a legközelebbi töltőállomásnál megáll tankolni, ami által az üzemanyag szintje visszaáll az előírt értéktartományba.

Mint látjuk, a felső rétegnek gyakran két alsó rétege van:

- a rendszer belső állapota, amit mérhet és szabályozhat,
- a rendszer külső állapota, amit mérhet, de nem befolyásolhat.

Nézzük meg ezt a megfigyelést UML ábrán az autós példánál maradva, betartva a DIP előírásait:



PlantUML szkriptje:

```
@startuml
class Sofőr {}
interface KormányMű {
    +{abstract} Vektor mér()
    +{abstract} void szabályoz(Vektor változtatás)
}
class Autó {}
interface AutóKörnyezete {
    +{abstract} Vektor mér()
}
class ÚtHálózat {}
Sofőr o--> KormányMű : kormányMű
Sofőr o--> AutóKörnyezete : környezet
ÚtHálózat --|> AutóKörnyezete
Autó --|> KormányMű
@enduml
```

A fenti leírásban nagyon fontos a „szokásos rutin”. A szokásos rutin keretében ellenőrzi a sofőr, hogy minden rendben van-e az autóval, nem megyünk-e túl lassan, túl gyorsan, van-e elég üzemanyag, nem világít-e valamelyik hibajelző. Irányítástechnikai szemszögből nézve, ez egy szabályozási kör.

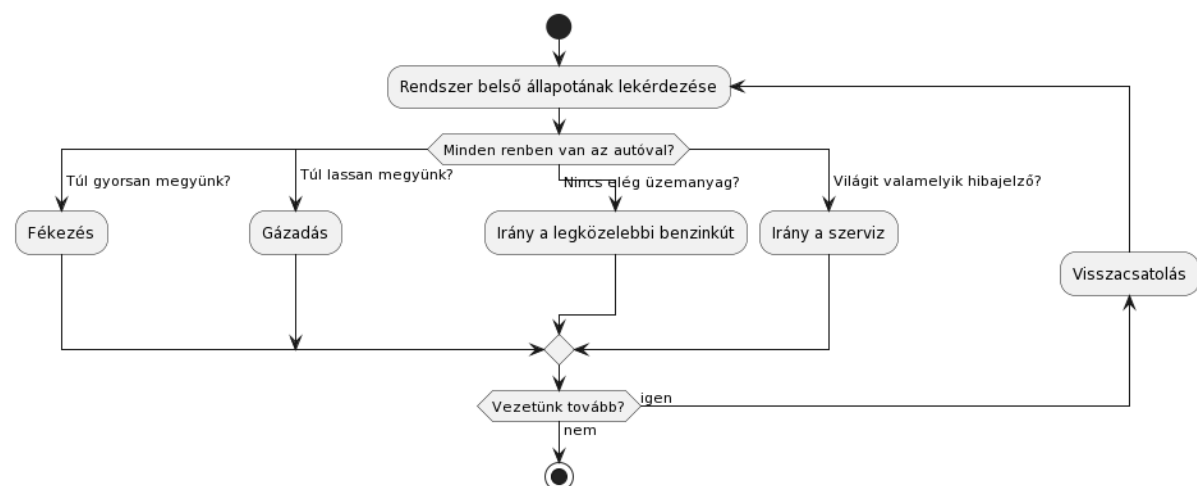
A szabályozási kör (angolul: control loop) részei:

- érzékelők, mérőeszközök, szenzorok, folyamat szenzorok (angolul: process sensor),
- szabályozó tevékenységek, folyamatok (angolul: controller function),
- döntéshozás, döntéstámogatás, végső szabályozó alrendszer (angol: final control element – FCE),
- szabályozott érték (angolul: process variable – PV)
- előírt érték (angolul: desired set-point – SP), még gyakrabban, előírt értéktartomány.

Mit látható, a szabályozási kör egyetlen eddig nem tárgyalt elemet tartalmaz, a döntéshozást. A döntéshozás nyilván az üzleti logika része. Ha csak néhány szabályról van szó, akkor érdemes egy egyszerű IF – ELSE IF szerkezetet használni. Ha már egy kicsit bonyolultabb a helyzet, akkor érdemes a döntéshozást egy startégiába kiszervezni. Végső esetben akár egy döntéstámogató rendszert is használhatunk, amibe betöltjük a döntéshozatal szabályait. Java esetén egy kiforrott döntéstámogató keretrendszer a Drools – Business Rules Management System nevű csomag.

Egy nagyon érdekes kérdés, hogy a szabályozási kör, mitől lesz kör? Induljunk ki a szabályozási kör angol nevéből: control loop. A loop angol szó fordítható körnek, de informatika területén leginkább ciklusnak fordítandó. Tehát a szabályozási kör az egy ciklus.

Nyilván ez lehet egy végtelen ciklus (while(true)), amiben időről-időre meghívjuk a rendszer belső állapotát figyelő metódusokat, és ha valami eltérést észlelünk, azaz a szabályozott értékek nem az előírt értéktartományban vannak, akkor meghívjuk a döntéshozót. A döntéshozó meghívja valamely szabályozó függvényt. Habár a Hollywood alapelv alapján, tudjuk, hogy a végtelen ciklusból való hívogatás, azaz a tevékeny várakozás (angolul: busy waiting), rossz megoldás, mégis felvázoljuk ezt egy UML aktivitási diagram segítségével:



PlantUML szkript:

@startuml

start

repeat :Rendszer belső állapotának lekérdezése;

switch (Minden renben van az autóval?)

case (Túl gyorsan megyünk?)

:Fékezés;

case (Túl lassan megyünk?)

:Gázadás;

case (Nincs elég üzemanyag?)

:Irány a legközelebbi benzinkút;

case (Világít valamelyik hibajelző?)

:Irány a szerviz;

endswitch

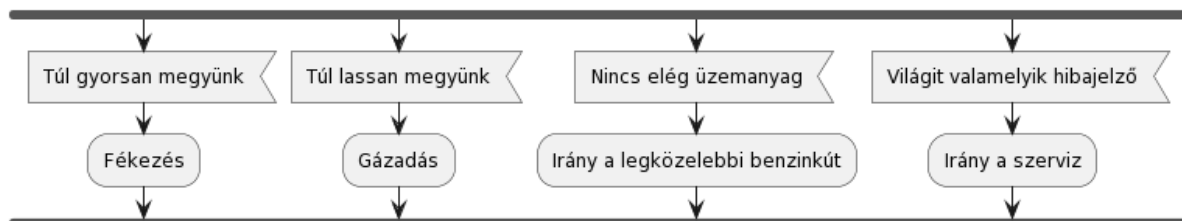
backward:Visszacsatolás;

repeat while (Vezetünk tovább?) is (igen) not (nem)

stop

@enduml

Ha nem szeretnénk megsérteni a Hollywood alapelvet, akkor a fenti megoldás helyett alkalmazhatunk esemény vezérlést is:



PlantUML szkript:

@startuml

fork

:Túl gyorsan megyünk<

:Fékezés;

fork again

:Túl lassan megyünk<

:Gázadás;

fork again

:Nincs elég üzemanyag<

:Irány a legközelebbi benzinkút;

fork again

:Világít valamelyik hibajelző<

:Irány a szerviz;

end fork

@enduml

Ez az esemény vezérelt változat. Ha jön egy Túl gyorsan megyünk esemény, akkor fékezünk. Ha kapunk egy Túl lassan megyünk eseményt, akkor gázt adunk, stb.... Ez nem végtelen ciklus, hanem esemény vezérelt programozás, csak akkor csinálunk valamit, ha a kapunk egy eseményt. Tehát ez megfelel a Hollywood alapelvnek.

1.1.12. DIP és LSP: Hogyan zárjuk egységbe az API szerződést

Egy kicsit feljebb, láttunk egy nagyon érdekes részt: „A CsereStratégia interfészből induló két <<call>> nyíl is konkrét osztályba mutat. Itt külön érdemes lenne megvitatni, hogy egy interfészből hogyan indulhat <<call>> nyíl, de fogadjuk el, hogy ez a szerződés része: A sokSzóCsere metódust a getSzótár és a csere metódusok hívásával kell megvalósítani.”

Amit egy kicsit feljebb elhanyagoltunk, azt most itt kifejtjük, hogyan indulhat ki egy interfészből <<call>> nyíl? Forráskód oldalról nézve ez első ránézésre hülyeség, hiszen egy interfészből nem hívhatunk metódust. Második ránézésre gondolhatunk arra, hogy Java programozási nyelven a default kulcsszó segítségével adhatok meg alapértelmezett implementációt egy metódusnak, ahogy a lenti példa is mutatja:

```
interface CsereStratégia {  
    default String sokCsere(String szöveg, VanSzótár szótár, TudCserélni  
cserélő) {  
        szótár.getSzótár();  
        return cserélő.csere();  
    }  
}
```

Ez a megoldás, csak akkor működik, ha a leggyengébb HAS-A kapcsolatot, az asszociációt használjuk, hiszen akkor paraméterben kapom meg a referenciát. Ha erősebb HAS-A kapcsolatot szeretnénk, akkor már mezőre van szükségünk, de Javában egy interfésznek nem lehet példány szintű mezője, csak osztály szintű. Javában minden interfészben deklarált mező public static final lesz. Ez azért van így, mert egy interfésznek nem lehet belsőállapota.

Ha mégis szükségünk van barátságnál erősebb HAS-A kapcsolatra, akkor meg kell értenünk a következő mondatot: Az interfész egy API-t zár egységbe, ahol az API alatt a publikus metódusok fejét és azok szerződéseit értjük. Másképen megfogalmazva, a sokat emlegetett absztrakció, az programozói szemszögből, egy API, azaz egységbe (interfészbe, vagy absztrakt osztályba) zárt metódus fejek és azok szerződései.

A szerződésekről már tanultunk az LSP elvnél. Azt tanultuk, hogy minden metódusnak van szerződése. Egy metódus szerződése azt mondja ki, hogy ha a hívó betartja a hívott metódus előfeltételét, akkor a hívott garantálja, hogy igaz lesz az utófeltétele is. Az előfeltétel függhet a metódus paramétereitől és a hívott objektum belső állapotától. Az utófeltétel függ a visszatérési értéktől és szintén a belső állapottól. Itt gyakran megkülönböztetjük a hívás előtti régi és a hívás utáni új belső állapotot. Ritka esetben a metódusnak lehet mellékhatása is, azaz megváltoztathatja a külső állapotot. Ilyenkor a szerződés rögzíti az előidézett mellékhatást is.

Az LSP elvénél azt tanultuk, hogy a szerződés megadható assert utasításokkal és az öndokumentáló megjegyzés használatával. A default kulcsszó használatával adhatunk meg assert-eket, de azt a többi programozó aligha fogja nézni, mert egy metódus törzse nem része az API-nak. Ezzel szemben az öndokumentáló megjegyzés nagyon is a része.

Nézzük meg az előbbi példát, hogyan oldjuk meg, hogy az implementációt elvégző programozó tudja, hogy neki meg kell hívnia egy-egy függvényt.

```
/**
 * Ez a metódus a bemeneti szövegben a szótár alapján a csere
 * segítségével kicseréli az összes előforduló szót és
 * ezt az új szöveget visszaadja.
 * @param szöveg Bemeneti szöveg, ebben cserél.
 * @param szótár Innen kell lekérni a szótárt.
 * @param cserélő Ennek a csere metódusa végzi a cserét.
 * @return A szócsere eredménye.
 */
String sokCsere(String szöveg, VanSzótár szótár, TudCserélni cserélő);
```

Mivel az öndokumentáló megjegyzés az API része, ezért ezt mindenki látni fogja, aki ezt az interfészt használja.

Már csak egy kérdés van: Van ennek a leírásnak bármilyen kényszerítő ereje? A válasz egyszerű, nincs! A fenti dokumentáció ellenére a megvalósító bármilyen implementációt írhat, nem fog fordítási hibát kapni. Javában, legjobb tudomásunk szerint, nincs olyan lehetőség, ami kikényszeríti a szerződés betartását.

Absztrakt osztályt és IoC-t alkalmazva már ki lehet kényszeríteni a szerződés betartását. Az LSP-ből sőt ismert BankAccount példán keresztül mutatjuk meg, hogyan lehet ezt elérni:

```
abstract class BankAccount {
    protected double balance;
    public final double Withdraw(double amount) {
        assert(amount >= 0 && amount <= balance);
        double newBalance = WithdrawImp(amount);
        assert(newBalance == balance - amount);
        balance = newBalance;
        return balance;
    }
    protected abstract double WithdrawImp(double amount);
}
```

Ebben a kis példában a Withdraw metódus része az API-nak, hiszen publikus. Nem lehet felülírni, hiszen final. Ugyanakkor IoC módszerével meghívja az absztrakt WithdrawImp metódust, amit már felül tud írni a gyerekosztály, hiszen protected abstract. Ugyanakkor a WithdrawImp nem része a felületnek, hiszen nem publikus. Csak a Withdraw metódust lehet hívni, viszont az lefuttatja az két assert utasítást, ami már kikényszeríti az elő- és utófeltétel használatát. Igaz, hogy csak futási időben, és csak akkor, ha nem használjuk a -disableassert (röviden: -da) kapcsolót. Szóval ezzel a trükkel valamennyi kényszerítő erőt tehetünk a gyermekosztályokra.

Ugyanakkor a legjobb, ha megbízunk a gyerekosztály írójában, hogy betartja az LSP elvet. Az LSP elv a következőket mondja ki:

Azt mondjuk, hogy A osztálynak B osztály altípusa, akkor és csak akkor, ha A-nak gyerekosztálya B, továbbá bármely T tulajdonságra igaz, hogy ha A minden x példányára igaz T(x), akkor B minden y példányára szintén igaz T(y).

Első dolog, amit meg kell értenünk, hogy nem minden gyerekosztály altípus, csak az, amire az LSP elv igaz. Erre néztük példaként a híres Téglalap – Négyzet példát, amit itt most nem ismételünk meg.

Továbbá azt is látnunk kell, hogy az altípus jó dolog, hiszen, ha Ős példányt használva az elvárásoknak megfelelően működött a programunk, akkor a gyermek példánnyal is jól fog. Itt van az LSP-nek egy gyenge értelmezése: Ha Ős példányt használva nem kapunk futási hibát, akkor gyermek példányt használva sem szabad futási hibát kapnunk. Habár ez az értelmezése eléggé elterjedt az LSP-nek, mi nem ezt használjuk, hiszen ez az az elv, ami matematikailag precízen megfogalmazott, nem kell homályos mondatokat értelmeznünk, azon gondolkozva, vajon mire gondolt az elv megalkotója.

Továbbá látnunk kell, hogy konkrét osztályból örököltetni, és abban reménykedni, hogy a gyerekosztály altípus lesz, elég botor dolog, mert minden bementre, amire az Ős kimenetet ad, ugyanazokra a bemenetekre a gyerekosztálynak is ugyanazt kell visszaadnia, mint az Ősnek. Na de akkor mi értelme van a gyerekosztálynak? Legfeljebb több paramétert fogadhat, csinálhat valami plusz dolgot, mondjuk logolást, esetleg használhat gyorsító tárt, de az alapvető viselkedése meg kell, hogy egyezzen az Ősével.

Örököltetni absztrakt osztályból, interfészből van értelme, hiszen ebben az esetben az Ősnek nincs konkrét működése, lehetnek elvárásaink, de azok szükségszerűen nem túl részletesek, van értelme a gyerekosztálynak, a gyerekosztályoknak, hiszen a hiányzó részleteket úgy töltik ki, ahogy akarják.

Az LSP elv vonatkozik a szerződés alapú programozásra is. A szerződés alapú programozás kimondja, minden metódusnak van előfeltétele és utófeltétele, és ha a hívó betartja a hívott előfeltételét, akkor igaz lesz az utófeltétele is. Az előfeltétel abból következik, hogy a programunk konzisztensen működik. Az utófeltételből pedig az következik, hogy a programunk továbbra is konzisztens.

Próbáljuk meg ezt a híres vasalós példával szemléltetni. A programunk konzisztensen működik, ha a vasaló, sosem melegebb, mint 250 fok, egyébként kigyulladhat a ruha, amit vasalunk. Továbbá feltételezzük, hogy mindig van maximum 28 fokos csapvizünk. Ebben a környezetben üzemeltetjük a vasalót. Tegyük fel, hogy van egy bekapcsol metódusunk, aminek előfeltétele, hogy van hűtővizünk és az maximum 50 fokos, illetve utófeltétele, hogy a vasaló sosem melegszik 200 fok felé. Ez előfeltétel teljesül, hiszen van csapvizünk, ami max 28 fokos. Ez formálisan is leírható: KONZISZTENS_KÖRNYEZET => ELŐFELTÉTEL. Ebben a konkrét esetben: (Víz-Hőfok <= 28) => (Víz-Hőfok <= 50).

Vizsgáljuk meg az utófeltételt, a vasaló maximum 200 fokra melegedhet egy belső szabályozási kör miatt. Ez jó, mert a konzisztens működéshez az kell, hogy a vasaló sose legyen melegebb, mint 250 fok. Ez formálisan is leírható: UTÓFELTÉTEL => KONZISZTENS-KÖRNYEZET. Ebben a konkrét esetben: (Vasaló-Hőfok <= 200) => (Vasaló-Hőfok <= 250).

Ha most a Vasalónak lesz egy gyerekosztálya, mondjuk HatékonyVasaló, akkor ennek a gyerekosztálynak ebben a környezetben kell működnie, ahol a csapvíz max 28 fokos, és a vasaló nem lehet 250 foknál melegebb. Ezt legkönnyebben úgy érhetjük el, hogy használjuk LSP elvet, hogy a gyerekosztály egyben altípus is legyen. Az LSP elvbe behelyettesítve a fenti formulákat, ezt kapjuk:

$(KK \Rightarrow \text{ŐS-ELŐFELTÉTEL}) \Rightarrow (KK \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$, illetve

$(\text{ŐS-UTÓFELTÉTEL} \Rightarrow KK) \Rightarrow (\text{GYEREK-UTÓFELTÉTEL} \Rightarrow KK)$, ahol KK a Konzisztens-Környezet.

Ahhoz, hogy ezek teljesüljenek, ahhoz a jól ismert következtetési lánc módszert fogjuk használni: Ha A-ból következik B, és B-ből következik C, akkor A-ból következik C. Ezt alkalmazva ezt kapjuk:

$((KK \Rightarrow \text{ŐS-ELŐFELTÉTEL}) \text{ és } (\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})) \Rightarrow (KK \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$, illetve

$((\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{ŐS-UTÓFELTÉTEL}) \text{ és } (\text{ŐS-UTÓFELTÉTEL} \Rightarrow \text{KK})) \Rightarrow (\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{KK}).$

Azaz, ezt a két megszorítást kell alkalmaznunk, hogy az LSP elv igaz legyen:

- $(\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$
- $(\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{ŐS-UTÓFELTÉTEL})$

Ezt a két plusz feltételt szavakkal is ki lehet fejezni. Ehhez azt kell tudnunk, hogy az erősebb feltételből következik a gyengébb. Persze, ha a két állítás ekvivalens, azaz ugyanolyan erős, akkor egyikből következik a másik. Tehát $A \Rightarrow B$ jelentése:

- A erősebb, mint B, vagy ekvivalensek; illetve
- A nem gyengébb, mint B; illetve
- B gyengébb, mint A, vagy ekvivalensek; illetve
- B nem erősebb, mint A.

Tehát ezt a két plusz feltételt így adhatjuk meg szavakkal:

- $(\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$
- $(\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{ŐS-UTÓFELTÉTEL})$
- A gyermekosztály előfeltétel nem erősebb, mint az ősz előfeltétele: Minden a **gyerekosztályban** felüldefiniált metódus **előfeltétele** vagy ekvivalens, vagy **gyengébb (azaz nem erősebb)**, mint az őszosztályban lévő.
- A gyermek utófeltétel nem gyengébb, mint az ősz utófeltétele: Minden a **gyerekosztályban** felüldefiniált metódus **utófeltétele** vagy ekvivalens, vagy **erősebb (azaz nem gyengébb)**, mint az őszosztályban lévő.

Nézzük meg, hogy mit jelent mindez a Vasaló – HatékonyVasaló példa esetén. Tegyük fel, hogy felülírjuk a vasaló bekapcsoló metódust. Emlékezzünk vissza, hogy a Vasaló osztályban, azaz az Őszosztályban ez volt az előfeltétel (Víz-Hőfok ≤ 50). Vajon az alábbi két lehetséges előfeltétel közül melyik felel meg az LSP elvnek?

1. (Víz-Hőfok ≤ 40)
2. (Víz-Hőfok ≤ 60)

Emlékezzünk vissza, hogy a csapvizünk max 28 fokos, ez mindkét előfeltételt kielégíti. Csakhogy se a Vasaló, se a HatékonyVasaló osztály nem tudja, hogy hány fokos a csapvizünk. Viszont a HatékonyVasaló tudja, hogy az Ősében mi az előfeltétel: (Víz-Hőfok ≤ 50). E szerint a legrosszabb esetben a környezetből jövő víz 50 fokos, és ez az 1. előfeltételt nem elégíti ki, viszont a 2.-at igen. Nézzük meg, hogy a formális vizsgálat is ezt mondja-e, azaz igaz-e $(\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$ állítás: $(\text{Víz-Hőfok} \leq 50) \Rightarrow (\text{Víz-Hőfok} \leq 60)$. Ha a víz hőfok kisebb egyenlő, mint 50, akkor bármilyen is a víz hőfoka, az kisebb, mint 60, és így a kisebb egyenlő is igaz. A 2. előfeltétel felel meg az LSP elvnek a HatékonyVasaló osztályban. Tehát a gyermekben az előfeltételt lehet gyengíteni, de erősíteni nem.

Hasonló gondolatmenet mentén megmutatható, hogy a HatékonyVasaló bekapcsol metódusának az utófeltétele lehet például az, hogy (Vasaló-Hőfok ≤ 180), de az nem lehet, hogy (Vasaló-Hőfok ≤ 300), de még az se, hogy (Vasaló-Hőfok ≤ 220), legalábbis, ha szeretnénk betartani az LSP elvet.

A szerződés alapú programozás esetén az osztályoknak van invariánsa is, ami minden metódus hívás előtt és után igaz. Nyilván az invariánsból kell, hogy következzen a program konzisztensen működése. Ha erre az állításra alkalmazzuk a LSP elvet, akkor ezt a formulát kapjuk:

$(\text{ŐS-INVARIÁNS} \Rightarrow \text{KK}) \Rightarrow (\text{GYERMEK-INVARIÁNS} \Rightarrow \text{KK})$, ahol KK a Konzisztens-Környezet.

Ahhoz, hogy ez a feltétel igazzá váljon, ahhoz így kell kiegészíteni:

$((\text{GYERMEK-INVARIÁNS} \Rightarrow \text{ŐS-INVARIÁNS}) \text{ és } (\text{ŐS-INVARIÁNS} \Rightarrow \text{KK})) \Rightarrow (\text{GYERMEK-INVARIÁNS} \Rightarrow \text{KK})$.

Azaz, ezt megszorítást kell alkalmaznunk, hogy az LSP elv igaz legyen:

- $(\text{GYERMEK-INVARIÁNS} \Rightarrow \text{ŐS-INVARIÁNS})$, amit a következő szavakkal fejezhetünk ki:
- A gyermek invariánsa vagy ekvivalens, vagy **erősebb (azaz nem gyengébb)**, mint az Ősosztály invariánsa.

A fentiekén túl az úgynevezett történeti megszorítás (angolul: history constraint) is szükséges, amely azt mondja ki, hogy az állapotátmenetekre vonatkozó megszorításokat is tiszteletben kell tartania a gyerekosztálynak, ahhoz, hogy altípus legyen. Szerencsére ez könnyen teljesül, ha az elő- és utófeltételekben megadjuk a belsőállapotokra vonatkozó feltételeket is. Ez egész pontosan ez az jelenti, hogy az utófeltételnél megadjuk azt is, hogy milyen állapotátmenetet okoz a metódus. Ha ezt betartjuk, akkor már csak arra kell vigyáznunk, hogy ne legyenek se publikus, se protected láthatósági szintű mezőink. Azt, hogy ne legyen publikus mezőnk, már jól megtanultuk, hiszen, ha van publikus mezőnk, akkor bárki kontrolálatlanul megváltoztathatja kívülről a példányaink belső állapotát, és ez baj. Ugyanez a baj a protected láthatósági szinttel is, bármelyik gyermek példány megváltoztathatja az ősből örökölt protected mező értékét, ami változásra az ősből nem tudunk felkészülni. Tehát, ha nincs se public, se protected mező az osztályunkban, és a metódusok utófeltételében megadjuk, hogy milyen állapotátmenet történik a metódus hívásakor, akkor a történeti megszorítás igaz lesz, nem kell vele külön foglalkozni. Minden más esetben annyi lehetséges állapotátmenet van, hogy úgyis elfelejtjük valamelyiket leírni, és így előfordulhat, hogy a gyerekosztály megsérti a történeti megszorítást. Éppen ezért, ebben a jegyzetben azt feltételezzük, hogy nem használunk se public, se protected mezőket, és minden metódus utófeltétele leírja a metódus által okozott állapotátmenetet is.

Már csak néhány tipikus példa van hátra, mit szabad és mit nem szabad egy gyerekosztály, vagy implementáló osztály készítése esetén. A példákhoz használni fogjuk a Páros(x) és Osztható(x, y) függvényt. Legyen Páros(x) jelentése: x páros szám; Osztható(x, y) jelentése: x osztható y-nal. Ekkor:

- $\text{Páros}(x) \Rightarrow \text{Osztható}(x, 2)$, hiszen a kettő ugyanazt jelenti.
- $\text{Osztható}(x, 2) \Rightarrow \text{Páros}(x)$, hiszen a kettő ugyanazt jelenti.
- $\text{Osztható}(x, 6) \Rightarrow \text{Páros}(x)$, hiszen minden 6-tal osztható szám páros is.
- Tudjuk, hogy $(\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$, azaz
- a gyerekosztályban az előfeltétel lehet gyengébb, de nem erősebb, például:
 - o $\text{ŐS-ELŐFELTÉTEL} = \text{Osztható}(x, 6)$,
 - o $\text{GYEREK-ELŐFELTÉTEL} = \text{Páros}(x)$.
- A gyerekosztályban az előfeltétel bővíthető VAGY operátorral, például:
 - o $\text{ŐS-ELŐFELTÉTEL} = \text{Osztható}(x, 2)$,
 - o $\text{GYEREK-ELŐFELTÉTEL} = \text{Osztható}(x, 2) \text{ VAGY } \text{Osztható}(x, 3)$.
- A gyerekosztályban az előfeltételéből elhagyható ÉS operátor, például:

- o $\text{ŐS-ELŐFELTÉTEL} = \text{Osztható}(x, 2) \text{ ÉS } \text{Osztható}(x, 3)$
- o $\text{GYEREK-ELŐFELTÉTEL} = \text{Osztható}(x, 2)$.
- Tudjuk, hogy $(\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{ŐS-UTÓFELTÉTEL})$, azaz
- a gyerekosztályban az utófeltétel lehet erősebb, de nem gyengébb, például:
 - o $\text{ŐS-UTÓFELTÉTEL} = \text{Páros}(x)$,
 - o $\text{GYEREK-UTÓFELTÉTEL} = \text{Osztható}(x, 6)$.
- A gyerekosztályban az utófeltétel bővíthető ÉS operátorral, például:
 - o $\text{ŐS-UTÓFELTÉTEL} = \text{Osztható}(x, 2)$,
 - o $\text{GYEREK-UTÓFELTÉTEL} = \text{Osztható}(x, 2) \text{ ÉS } \text{Osztható}(x, 3)$.
- A gyerekosztályban az utófeltételéből elhagyható VAGY operátor, például:
 - o $\text{ŐS-UTÓFELTÉTEL} = \text{Osztható}(x, 2) \text{ VAGY } \text{Osztható}(x, 3)$
 - o $\text{GYEREK-UTÓFELTÉTEL} = \text{Osztható}(x, 2)$.

Összegezve, ahhoz, hogy a gyermek-, illetve az implementáló osztály altípus legyen, ezt a 3 megszorítást kell alkalmazni:

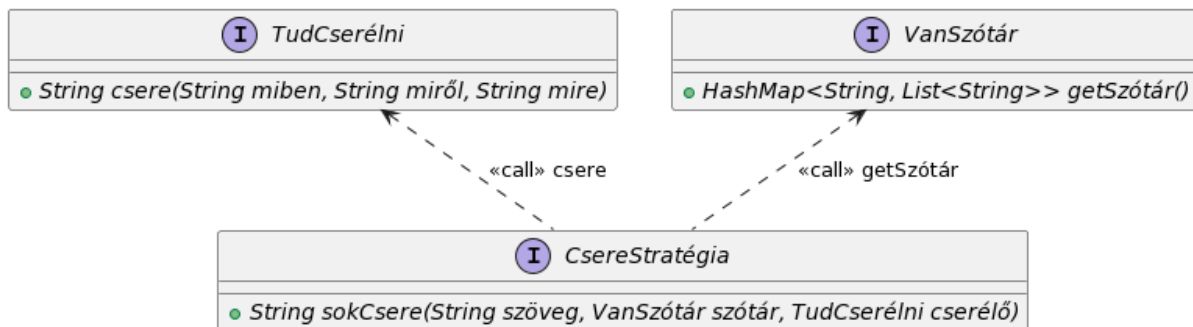
- $(\text{ŐS-ELŐFELTÉTEL} \Rightarrow \text{GYEREK-ELŐFELTÉTEL})$,
- $(\text{GYEREK-UTÓFELTÉTEL} \Rightarrow \text{ŐS-UTÓFELTÉTEL})$,
- $(\text{GYERMEK-INVARIÁNS} \Rightarrow \text{ŐS-INVARIÁNS})$.

Azután, hogy ilyen részletesen tárgyaltuk az LSP elvet, nézzük meg az LSP és a DIP kapcsolatát. A DIP elv esetén mindig van egy absztrakt osztály vagy interfész. Erre az absztrakcióra mindig mutat legalább egy HAS-A kapcsolat és legalább egy IS-A kapcsolat. A két elv úgy kapcsolódik, hogy az absztrakcióban megfogalmazott szerződések (az elő- és utófeltételeket, az invariánst) az IS-A kapcsolatban lévő osztályokban a fenti 3 megszorítást alkalmazva meg kell megvalósítani.

Forráskód szintén az interfész, vagy absztrakt osztály öndokumentáló megjegyzésébe kell beleírni a metódusok szerződéseit, illetve az esetleges invariánst. Ez utóbbit nehéz elképzelni, hiszen az invariáns a belsőállapothoz köthető fogalom. Az osztály invariánsa minden publikus metódus hívás előtt és után igaz kell, hogy legyen. Mivel egy interfésznek nincs belső állapota, ezért invariánsa sincs. Ezután az implementációt megvalósító programozóban bízunk, hogy betartja az LSP elvet. Ettől többet az interfész megírásának az idejében, ami a DIP szerint megelőzi az osztályok megírását, nem nagyon tehetünk, vagy csak nagyon nyakatekert módon.

A DIP és az LSP közt feszültség van, mert a DIP elv szerint jobb, ha interfészeket használunk, de az interfészeknek nincs belső állapota, az LSP elv szerint pedig le kell írunk a szerződéseket, de azt meg a belső állapot nélkül nem lehet, csak akkor, ha leírt dolognak nincs állapota, vagy csak egy állapota van. A megoldás az absztrakt osztályok használata. A második megoldás, hogy az interfészben getter metódusokat definiálunk, és ha egy mezőre kellene hivatkozni, akkor a getter metódusra hivatkozunk.

Már csak egy nyitott kérdés maradt, hogyan lehet UML ábrán megadni a szerződéseket. Erre már láttunk egy lehetőséget, a szaggatott nyíl használatát mindenféle sztereotípa segítségével, például a `<<call>>` segítségével. Ez egy nagyon hasznos és jól olvasható eszköz. Pl.:



PlantUML szkriptje:

```

@startuml
interface VanSzótár{
+ {abstract} HashMap<String, List<String>> getSzótár()
}
interface TudCserélni{
+ {abstract} String csere(String miben, String miről, String mire)
}
interface CsereStratégia {
+ {abstract} String sokCsere(String szöveg, VanSzótár szótár, TudCserélni cserélő)
}
CsereStratégia .up.> VanSzótár : <<call>> getSzótár
CsereStratégia .up.> TudCserélni : <<call>> csere
@enduml
  
```

A másik megoldás, hogy az UML logikai nyelvét használjuk, az úgynevezett Object Constraint Language – OCL nyelvet. Ezt magyarra talán Objektum Megszorítási Nyelvnek lehetne fordítani, de inkább maradunk az OCL rövidítés használatánál.

Szerencsére az OCL támogatja az invariánsok, az elő- és utófeltételek megadását, rendre az inv, pre, és post kulcsszavak segítségével. Ezek szintaxisa (a kulcsszavak félkövérek):

context <osztály neve> **inv** [<megszorítás neve>]: <Boolean OCL kifejezés>

context <osztály neve>::<metódus neve>(<paraméter lista>) : <visszatérési típus> **pre** [<megszorítás neve>]: <Boolean OCL kifejezés>

context <osztály neve>::<metódus neve>(<paraméter lista>) : <visszatérési típus> **post** [<megszorítás neve>]: <Boolean OCL kifejezés>

Ha környezet rész ugyanaz, akkor a pre és a post összevonható:

context <osztály neve>::<metódus neve>(<paraméter lista>) : <visszatérési típus>

pre [<megszorítás neve>]: <Boolean OCL kifejezés>

post [<megszorítás neve>]: <Boolean OCL kifejezés>

A paraméter listában a paramétereket paraméter név: típus szintaxissal lehet megadni. A Boolean OCL kifejezés tartalmazhat metódushívásokat, és logikai operátorokat. Mezőkre a self referencia megadásával lehet hivatkozni. Az utófeltételben a mező hívás előtti értékére is lehet hivatkozni a mezőnév@pre szintaxis segítségével. A visszatérési értékre a result kulcsóval lehet hivatkozni. Kivétel

dobás a throw kulcsszóval lehetséges a body részben. A body részben lehet if then else endif szerkezetet használni. Egyszerű típusok: Boolean, Integer, Real, String. Összetett típusok: Collection, Set, Bag, Sequence. Ezeknek sok hasznos metódusuk is van. Ezeket itt nem soroljuk fel, de egy-egy példában felhasználunk egy párat, ha azok könnyen érthetőek. Nézzünk pár példát:

context BankAccount **inv:** self.balance >= 0

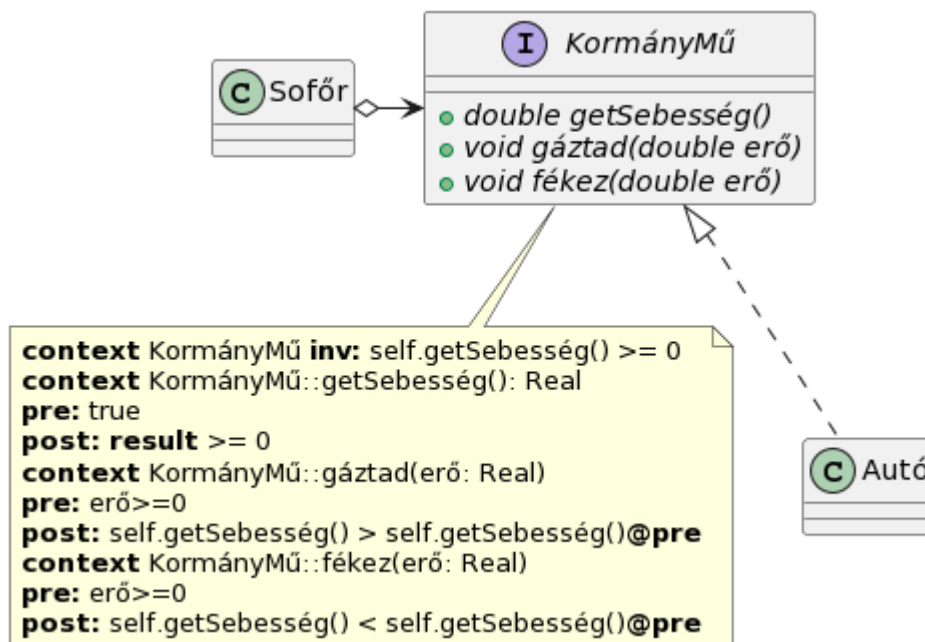
context BankAccount::Withdraw(amount: Real): Real

pre: amount >= 0 **and** self.balance >= amount

post: self.balance = self.balance@**pre** – amount **and** **result** = self.balance

Sajnos a PlantUML nem támogatja az OCL nyelvet, de elég sok szabadságot nyújt, hogy egyszerű szöveges információként felvigyük a metódusok szerződését. Erre látunk itt 3 példát.

Az első példában kiemeljük megjegyzésbe az OCL megszorításokat:



Az első példa PlantUML szkriptje:

@startuml

class Sofőr {}

interface KormányMű {

+{abstract} double getSebesség()

+{abstract} void gáztad(double erő)

+{abstract} void fékez(double erő)

}

note as SZERZŐDÉS

****context**** KormányMű ****inv:**** self.getSebesség() >= 0

```

**context** KormányMű::getSebesség(): Real
**pre:** true
**post:** **result** >= 0

**context** KormányMű::gáztad(erő: Real)
**pre:** erő>=0
**post:** self.getSebesség() > self.getSebesség()**@pre**

**context** KormányMű::fékez(erő: Real)
**pre:** erő>=0
**post:** self.getSebesség() < self.getSebesség()**@pre**

```

end note

```
class Autó {}
```

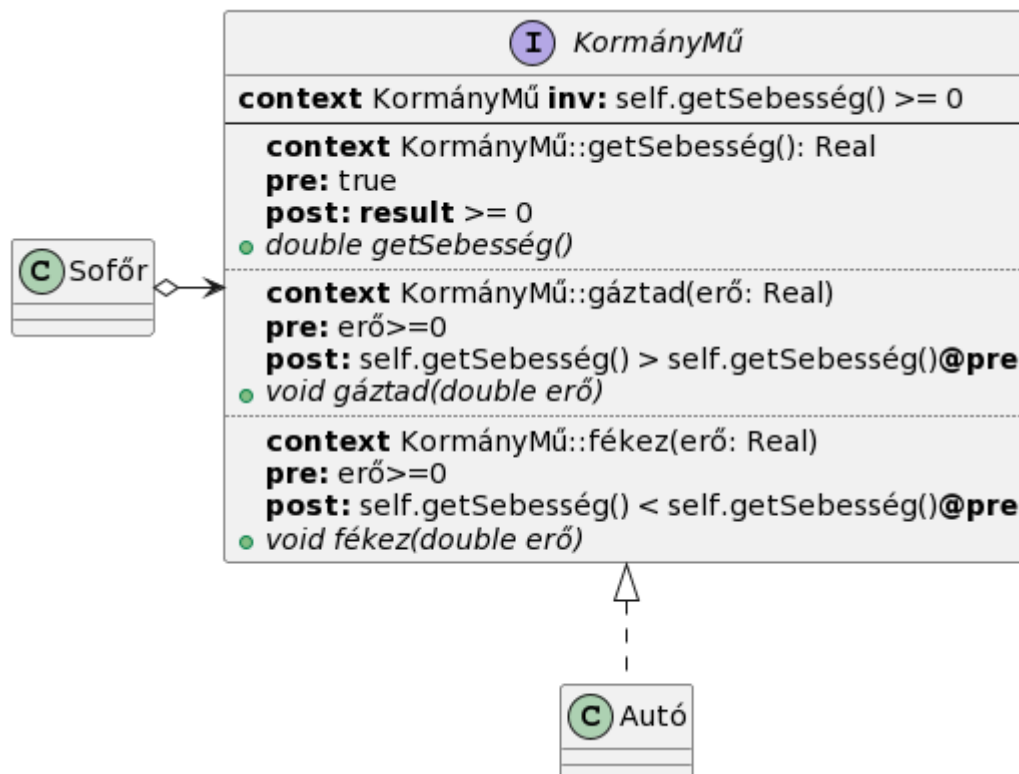
Sofőr o-right-> KormányMű

Autó .up.|> KormányMű

KormányMű .. SZERZŐDÉS

@enduml

A második példában az osztály szövegének része az OCL leírás, és kitesszük a **context** részt:



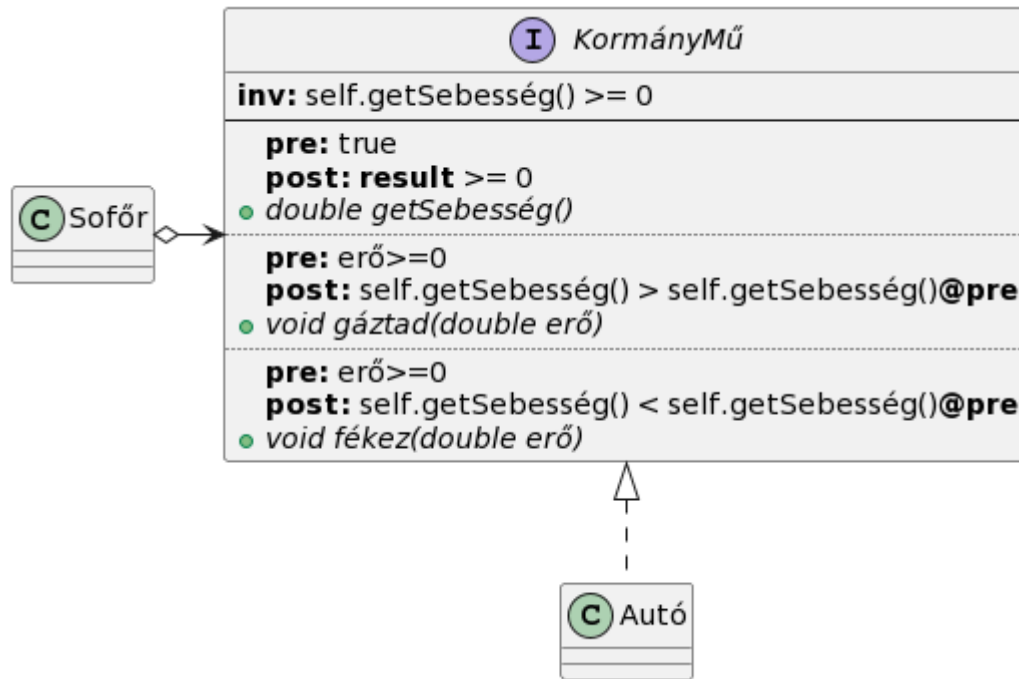
A második példa PlantUML szkriptje:

```

@startuml
class Sofőr {}
interface KormányMű {
    **context** KormányMű **inv:** self.getSebesség() >= 0
    --
    **context** KormányMű::getSebesség(): Real
    **pre:** true
    **post:** **result** >= 0
    +{abstract} double getSebesség()
    ..
    **context** KormányMű::gáztad(erő: Real)
    **pre:** erő>=0
    **post:** self.getSebesség() > self.getSebesség() **@pre**
    +{abstract} void gáztad(double erő)
    ..
    **context** KormányMű::fékez(erő: Real)
    **pre:** erő>=0
    **post:** self.getSebesség() < self.getSebesség() **@pre**
    +{abstract} void fékez(double erő)
}
class Autó {}
Sofőr o-right-> KormányMű
Autó .up. |> KormányMű
@enduml

```

A harmadik példában az osztály szövegének része az OCL leírás, a felesleges **context** részt nem írjuk ki:



A harmadik példa PlantUML szkriptje:

```

@startuml
class Sofőr {}
interface KormányMű {
  **inv:** self.getSebesség() >= 0
  --
  **pre:** true
  **post:** **result** >= 0
  +{abstract} double getSebesség()
  ..
  **pre:** erő>=0
  **post:** self.getSebesség() > self.getSebesség()@pre
  +{abstract} void gáztad(double erő)
  ..
  **pre:** erő>=0
  **post:** self.getSebesség() < self.getSebesség()@pre
  +{abstract} void fékez(double erő)
}
class Autó {}
  
```

Sofőr o-right-> KormányMű

Autó .up.|> KormányMű

@enduml

A három példa teljesen ugyanazt mutatja, az autós példának azt a részét, amikor gázadással és fékezéssel tudjuk szabályozni a sebességet. Mivel interfészt használunk, nem írhatjuk elő, hogy legyen sebesség mezőnk, de azt előírhatjuk, hogy legyen getSebesség getter metódus. Mind a három példa hibás abból a szempontból, hogy a **@pre** módosítót elvileg csak mező neve után lehet írni, ami a mező előző értékét jelöli még a hívás előtt. Ugyanakkor, amiatt, hogy interfészt használtunk, nem lehetnek mezőink, ezért választottuk ezt a hibás, de könnyen érthető megoldást.

Kritikaként lehetne megfogalmazni, hogy középen van egy szépen kidolgozott interfész a példában, de se a Sofőr, se az Autó osztályról nincs semmi részlet. Ugyanakkor ez szándékos volt, hogy kiemeljük, hogy a DIP szempontjából az absztrakció a lényeg, minden az absztrakciótól függ.

A konkrét példával szemben lehetne még megfogalmazni, hogy az autó sebessége lehet negatív is, hiszen mehetünk hátrafelé is. Ugyanakkor a sebességmérő órán nincs negatív tartomány. Az első és második példával szemben pedig azt lehet kifogásolni, hogy az OCL szerint a getSebesség metódus egy Real-t ad vissza az UML szerint meg double értéket. Sajnos OCL-ben nincs double, Javában meg nincs Real. Ez tényleg zavaró, de azért még elfogadható különbség.

További kritika lehet a konkrét példával kapcsolatban, hogy a szerződés nem fogalmazza meg azt az evidenciát, hogy minél nagyobb erővel nyomom a gázpedált, annál nagyobb a gyorsulás. Csak annyit fogalmaz meg, ha gázt adok, akkor nő a sebesség. Ez direkt van így, hogy lássuk, hogy a tervezés során nem szabad túl sok részletet rögzíteni. A részletek kidolgozást rá kell hagyni az implementáló osztályokra.

Azt, hogy milyen részletességgel kell megadni a szerződést, az lényegében azon az API-n múlik, amit ezen az interfészen keresztül akarunk elérni. Más szóhasználattal, egységbe zárni. Legyen a példánk a Verem API. Ez az API nagyon egyszerű, beállítható a verem maximális mérete, ami nem lehet kisebb, mint az eddigi maximális verem méret, a verembe lehet betenni elemet a push metódussal, kivenni pedig a pop metódussal. Ha a teli verembe akarunk még tenni elemet, az lehetetlen, ezért a TeleVan kivételt dobja. Ha az üres veremből akarunk kivenni elemet, az lehetetlen, ezért ekkor az Üres kivételt dobja. Nézzük ennek az API-nak az UML ábráját:



PlantUML szkript:

@startuml

interface Veren<T> {

 inv: $0 \leq \text{self.getMéret}()$

 and $\text{self.getMéret}() \leq \text{self.getMaxMéret}()$

--

 pre: $\text{méret} \geq \text{self.getMaxMéret}()$

 post: $\text{self.getMaxMéret}() = \text{méret}$

 +{abstract} void setMaxMéret(int méret)

..

 pre: True

 post: **result** ≥ 0

 +{abstract} int getMaxMéret()

..

 pre: True

 post: **result** ≥ 0 **and** **result** $\leq \text{self.getMaxMéret}()$

```

+{abstract} int getMéret()
..
**pre:** self.getMéret() < self.getMaxMéret()
**body:** **if** self.getMéret() = self.getMaxMéret()
    **then** **throw** new TeleVan()
    **endif**
**post:** self.getMéret() = self.getMéret()**@pre** + 1
+{abstract} void push(T e) throws TeleVan
..
**pre:** self.getMéret() > 0
**body:** **if** self.getMéret() = 0
    **then** **throw** new Üres()
    **endif**
**post:** self.getMéret() = self.getMéret()**@pre** - 1
+{abstract} T pop() throws Üres
}
@enduml

```

Mint látjuk, amiatt, hogy interfészt használtunk, nagyon sok kompromisszumot kellett hoznunk. Például a GetMaxMéret metódus utófeltétele nagyon megengedő. Lássuk, mennyivel konkrétan fogalmazhatjuk meg a Verem API-t, ha absztrakt osztályt használhatunk:



PlantUML szkript:

@startuml

abstract class Veren<T> {

 inv: $0 \leq \text{self.méret}$ **and** $\text{self.méret} \leq \text{maxMéret}$

 int méret = 0, maxMéret = 100

--

 pre: $\text{méret} \geq \text{self.maxMéret}$

 post: $\text{self.maxMéret} = \text{méret}$

 +{abstract} void setMaxMéret(int méret)

..

 pre: True

 post: **result** = self.maxMéret

 +{abstract} int getMaxMéret()

..

 pre: True

```

**post:** result** = self.méret
+{abstract} int getMéret()
..
**pre:** self.méret < self.maxMéret
**body:** **if** self.méret = self.maxMéret
    **then** **throw** new TeleVan()
    **endif**
**post:** self.méret = self.méret**@pre** + 1
+{abstract} void push(T e) throws TeleVan
..
**pre:** self.méret > 0
**body:** **if** self.méret = 0
    **then** **throw** new Üres()
    **endif**
**post:** self.méret = self.méret**@pre** - 1
+{abstract} T pop() throws Üres
}
@enduml

```

Absztrakt osztályt használva sokkal több részletet megadhatunk, sokkal konkrétabb szerződéseket írhatunk, ugyanakkor megadtunk egy olyan részletet is, ami már piszkos részletnek számít. Megadtuk a Verem kezdő maximális méretét, 100-ra állítottuk ezt a részletet, amit igazán semmi sem indokol. Nem volt benne az API leírásban. Ezt a piszkos részletet jobb lett volna a gyerekosztályokra hagyni. A másik gond: Milyen láthatósági szintje legyen a két mezőnek? Publikusak nyilván nem lehetnek, de protected-ek sem, mert akkor a történeti megszorítást is figyelembe kellene venni. Akkor marad a private láthatósági szint, ami meg szintén nem lehetséges, mert nincs setMéret metódusunk. A megoldás nyilván private mezők és egy protected setMéret metódus, de erre kezdő tervezőként nehéz rájönni.

Ezzel a kis példával csak azt akartuk aláhúzni, hogy sok szempontból jobb, ha a DIP által előírt absztrakció interfész.

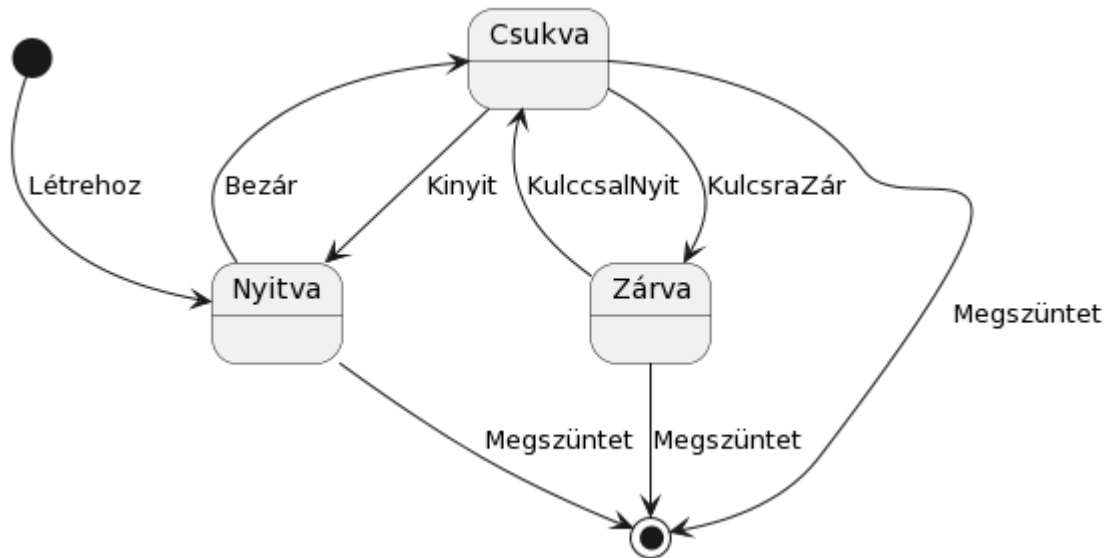
1.1.13. DIP és az állapotgép

Gyakran halljuk, hogy a programozó legjobb barátja az állapotgép (angolul: state machine). Az állapotgép világosan leírja, hogy egy objektum, milyen esemény hatására milyen állapotból milyen állapotba kerül, azaz az objektum viselkedését írja le részletesen. Ez a fajta részletes leírás csak ritkán áll rendelkezésünkre. Általában csak egy víziót kapunk, jobb esetben egy követelmény listát, még jobb esetben egy funkcionális specifikációt. Ennél jobb leírásunk általában csak akkor van, ha a csapatban van tervező. A tervezőnek a feladata, hogy előre átgondolja, hogyan kell kielégíteni a követelményeket, megoldani a feladatot, rájönni, hogy vannak-e félreértések.

Ha a tervező eljut az állapotgép készítésig, azzal nagyon sokat segít a programozónak.

Első lépésben nézzük meg, hogyan lehet UML állapotgépet diagramot készíteni. Majd nézzük néhány példát. Végül megvizsgáljuk a DIP és az állapotgép kapcsolatát.

Az UML állapotgép diagram mindig egy kezdőállapotból (angolul: initial state) indul, és minden út egy végállapotba vezet (angolul: final state). Ugyanakkor nem kötelező, hogy legyen kezdő- és végállapot. A köztük állapothoz vannak, amit lekerekített sarkú téglalappal jelölünk. Az állapotok közt nyilak vannak, amik eseményeket, metódushívásokat jelölnek. Ezek a nyilak állapotátmenetek (angolul: state transition). Nézzünk egy nagyon egyszerű példát:



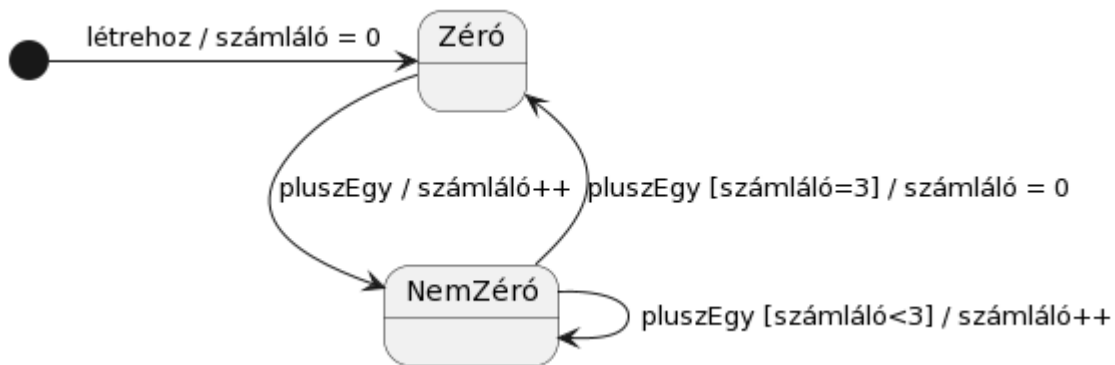
PlantUML szkript:

```
@startuml
[*] -down-> Nyitva : Létrehoz
Nyitva -down-> Csukva : Bezár
Csukva --> Nyitva : Kinyit
Csukva -right-> Zárva : KulcsraZár
Zárva --> Csukva : KulccsalNyit
Nyitva --> [*] : Megszüntet
Csukva --> [*] : Megszüntet
Zárva --> [*] : Megszüntet
@enduml
```

A fenti állapotgép egy ajtó lehetséges állapotait írja le. Először is létre kell hozni az ajtót, ami kezdetben nyitott lesz (nyitva állapot). Ha bezárjuk, csukva állapotba kerül. Majd, ha kulccsal bezárjuk, akkor zárva állapotba kerül. Az ajtót példányt, bármelyik állapotban is van, megszüntethetjük. Csak a csukott ajtót lehet kinyitni, a kulcsra zártat nem. A kulcsra zárt ajtót először kulccsal nyitni kell, csak

utána lehet kinyitni. Ez mind-mind szépen leolvasható az ábráról. Ezt leprogramozni könnyű, nem kell azon rágódni, mit is szeretne a megrendelő.

Egy állapot átmenetre írhatunk kiváltó eseményt (angolul: trigger), index zárójelben OCL feltételt, amit őrfeltételnek (angolul: guard) szoktunk hívni, és per (/) jel után pedig programkódot írhatunk, például, hogy egy mező hogyan változik. Ez utóbbi szoktuk hatásnak (angolul: effect) is hívni. Nézzünk egy egyszerű példát, a 2 bites számlálót:



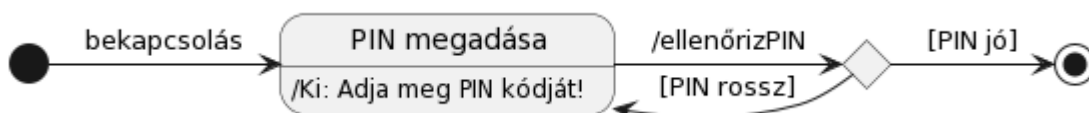
PlantUML szkript:

```

@startuml
[*] -right-> Zéró : létrehoz / számláló = 0
Zéró --> NemZéró : pluszEgy / számláló++
NemZéró --> NemZéró : pluszEgy [számláló<3] / számláló++
NemZéró --> Zéró : pluszEgy [számláló=3] / számláló = 0
@enduml
  
```

Ez nem a 2 bites számláló szokásos ábrázolása. A szokásos leírásban 4 állapot van: 00 -> 01 -> 10 -> 11 -> 00, ugyanakkor ebben a példában nincs lehetőség bemutatni az állapot átmenetre írható „kiváltó esemény [őrfeltétel] / hatás” hármast. Ezért a fenti példában csak két állapot van: Zéró és NemZéró. A 2 bites számláló értékét a számláló mezőben tároljuk. A kiváltó esemény a pluszEgy. A NemZéró állapotból pluszEgy hatására vagy maradunk a NemZéró állapotban, ha még nem értük el számlálóval a 3-mat, ha már elértük, akkor visszakörülünk a Zéró állapotba, hiszen 2 biten maximum 3-ig tudunk számolni.

Fontos megjegyezni, hogy a fenti hármastól bármelyik elhagyható. Erre nézzük meg a PIN kód példát, ami egyben példa az elágazásra is:



PlantUML szkript:

```

@startuml
state elágazás <<choice>>
  
```



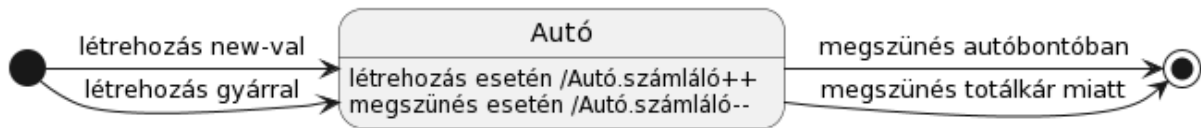
```
state "PIN megadása" as PIN
PIN: /Ki: Adja meg PIN kódját!
[*] -right-> PIN : bekapcsolás
PIN -right-> elágazás : /ellenőrizPIN
elágazás -left-> PIN : [PIN rossz]
elágazás -right-> [*] : [PIN jó]
@enduml
```

ITT járok

```
@startuml
state elágazás <<choice>>
state elágazás2 <<choice>>
state "PIN megadása" as PIN
PIN: /Ki: Adja meg PIN kódját!
[*] -right-> PIN : bekapcsolás / counter = 0
PIN -right-> elágazás : /ellenőrizPIN
elágazás -left-> elágazás2 : [PIN rossz] / counter++
elágazás2 --> PIN : [counter<3]
elágazás2 --> [*] : [counter==3]
elágazás -right-> [*] : [PIN jó]
@enduml
```

Itt lázhatunk olyan állapotátmenetet, ahol csak a kiváltó eseményt adtuk meg (bekapcsolás), olyat, ahol csak OCL őrfeltételt ([PIN rossz], illetve [PIN jó]), és olyat is, ahol csak hatást (/ellenőrizPIN).

Illetve látunk egy olyat is, hogy „/Ki: Adja meg PIN kódját!”. Ez a következő szint, amikor az állapoton belül tevékenységeket (angolul: state action) adunk meg. Ez lehet egy egyszerű üzenet a felhasználónak, de akár programkód is kerülhet ide. Ennek akkor van jelentősége, ha több ugyanoda mutató állapot átmenetnek is ugyanaz a hatása, vagy a kimenő éleknek ugyanaz a hatása, de bármi mást is kifejezhetünk. A szokásos szintaxis ugyanaz, mint az állapot átmenet esetén: <kiváltó esemény> <[OCL őrfeltétel]> / hatás. E hármasból a hatást mindenképp meg kell adni, ami általában valamilyen pszeudokód. Ha egyértelmű, hogy hatásról van szó, akkor a per (/) jel elhagyható. Ha nem írunk semmit, csak a hatást, akkor ezt úgy értjük, hogy ez a hatás, ha belépünk az állapotba. Nézzünk példát állapoton belül tevékenységre:



PlantUML kód:

@startuml

Autó: létrehozás esetén /Autó.számláló++

Autó: megszűnés esetén /Autó.számláló--

[*] -right-> Autó : létrehozás new-val

[*] -right-> Autó : létrehozás gyárral

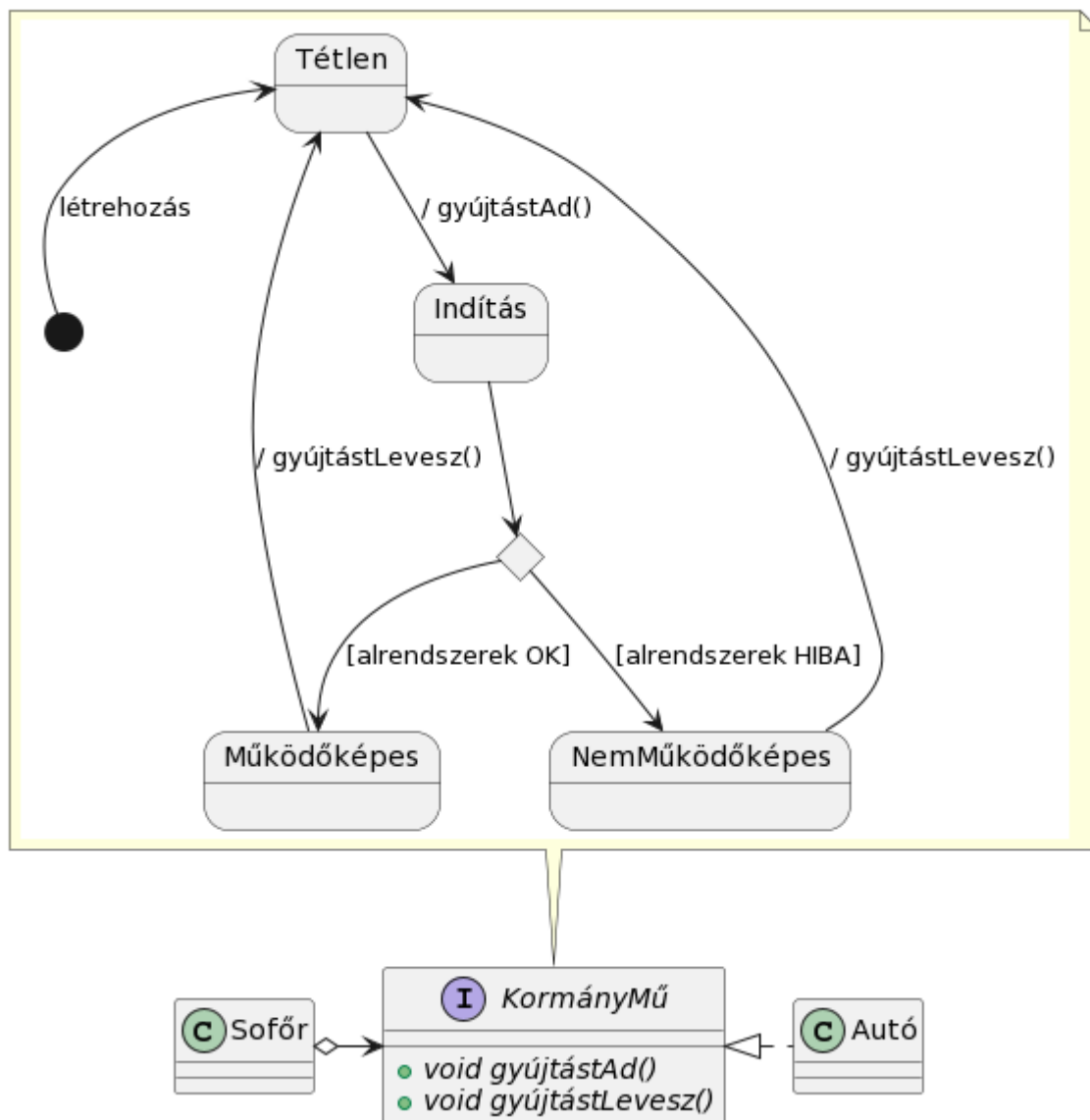
Autó -right-> [*] : megszűnés totálkár miatt

Autó -right-> [*] : megszűnés autóbontóban

@enduml

Hogyan kapcsolódik a DIP és az állapotgép? Könnyen belátható, hogy az állapotgép inkább az üzleti logikához kapcsolódik, feltéve, hogy elég magas szinten írja le a rendszer működését, sok részletet nyitva hagy, az alsóbb rétegek kidolgozhatják ezeket a részleteket.

Ha ez így van, akkor a DIP szerint az állapotgép az absztrakció része. Egy olyan szerződés, amit állapotokkal fogalmazunk meg. Ahhoz, hogy ezt ábrázolni tudjunk, ahhoz az osztály diagramot kell vegyíteniünk állapotgép diagrammal. Ez többféleképp is megoldható. Az egyik legegyszerűbb, ha az állapotgépet betesszük egy megjegyzésbe, és a megjegyzés az absztrakcióra mutat. Nézzünk erre egy példát:



PlantUML kód:

```
@startuml
```

```
allow_mixing
```

```
class Sofőr {}
```

```
interface KormányMű {
```

```
    +{abstract} void gyújtástAd()
```

```
    +{abstract} void gyújtástLevesz()
```

```
}
```

```
class Autó {}
```

```
note as ÁllapotGép
```

```
{{
```

state Elágazás <<choice>>

[*] -up-> Tétlen : létrehozás

Tétlen --> Indítás : / gyújtástAd()

Indítás --> Elágazás

Elágazás --> Működőképes : [alrendszerek OK]

Elágazás --> NemMűködőképes : [alrendszerek HIBA]

Működőképes --> Tétlen : / gyújtástLevesz()

NemMűködőképes --> Tétlen : / gyújtástLevesz()

}}

end note

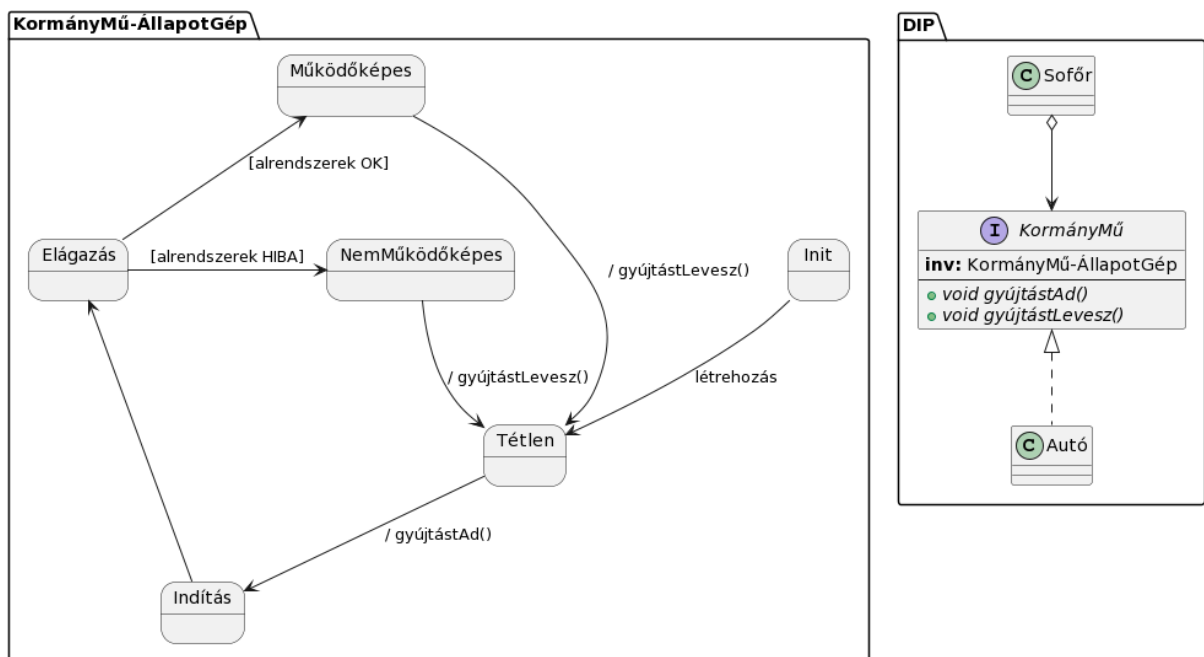
ÁllapotGép -- KormányMű

Sofőr o-right-> KormányMű

Autó .left. |> KormányMű

@enduml

A másik lehetőség, hogy közvetlenül az absztrakt osztályon vagy interfészen belül adjuk meg az állapotgépet. Még ezt is támogatja a PlantUML úgy hogy az állapotgépet {{ }} jelek közé írjuk, mint az előző példában a megjegyzésen belül. Még jobban kombinálhatjuk a különböző diagramokat, ha PlantUML-ben kiadjuk az allow_mixing parancsot. Sajnos ez elég csúnya végeredményt ad. Jobb a harmadik megoldás, amikor csak jelzem, hogy ehhez az absztrakcióhoz van egy állapotgép, és azt egy másik ábrán adom meg. Erre is nézzünk egy példát. Ebben a példában egyszerű invariánsként adom meg, hogy van állapotgép, amit be kell tartani:



PlantUML kód:

```

@startuml
allow_mixing
package DIP {
    class Sofőr {}
    interface KormányMű {
        **inv:** KormányMű-ÁllapotGép
        --
        +{abstract} void gyújtástAd()
        +{abstract} void gyújtástLevesz()
    }
    class Autó {}
    Sofőr o-down-> KormányMű
    Autó .up.|> KormányMű
}
package KormányMű-ÁllapotGép {
    state Init
    state Tétlen
    state Indítás
    state Elágazás <<choice>>
    state Működőképes
    state NemMűködőképes
    Init --> Tétlen : létrehozás
    Tétlen --> Indítás : / gyújtástAd()
    Indítás --> Elágazás
    Elágazás -up-> Működőképes : [alrendszerek OK]
    Elágazás -right-> NemMűködőképes : [alrendszerek HIBA]
    Működőképes --> Tétlen : / gyújtástLevesz()
    NemMűködőképes --> Tétlen : / gyújtástLevesz()
}
@enduml

```

Ebben a példában már használni kellett az `allow_mixing` parancsot. Habár az előző példában is benne volt, ott kihagyható lett volna. Sajnos a kezdő állapot, ami PlantUML-ben a `[*]` karaktersorozat jelöl, mindig hibát okoz. Ezért ezt az `Init` állapottal váltottuk ki. Mint látható, a két példa lényegében ugyanaz. Talán még szebb lenne különálló diagramokat használni. Itt csak a jegyzet kedvéért tettük be a két diagramot egy ábrába. Minden tervező döntse el, hogy neki melyik megoldás a szimpatikusabb.

A lényeg, az állapotgépet, mint a szerződés részét kell használni, invariánsként lehet felvenni az absztrakció megtervezésekor.

1.1.14. DIP vizsgálata Imperatív vs. Deklaratív szemszögből

Az előző részekben a DIP és a szerződések kapcsolatát elemeztük. Azt láttuk, hogy a szerződés rögzíti, hogy MIT csinál a metódus, de nem rögzíti, hogy HOGYAN oldja meg ezt a feladatot. A HOGYAN kérdésre a metódus implementációja adja meg a választ, de azt majd csak az implementáló osztály adja meg. A MIT és HOGYAN szavakat azért kell kiemelni, mert egy nagyon fontos dologra tudunk a segítségükkel rávilágítani.

Tudjuk, hogy a programozási nyelveknek 2 nagy családja van, az imperatív és a deklaratív programozási nyelvek. Minden programozási nyelven algoritmusokat írunk és egy algoritmusnak két kérdésre kell válaszolnia: MIT old meg? és HOGYAN oldja meg?

Jól tudjuk, hogy az imperatív nyelvek a HOGYAN kérdésre teszik a hangsúlyt, a programozó lényegében szabad kezét kap, HOGYAN implementálja az adott feladatot. Azt szoktuk mondani, hogy az imperatív oldalon a programozó szabadon lehet hülye.

Ez egyre kevésbé igaz, mert az igazán veszélyes, bonyolult, könnyen elrontható programozó eszközöket sorra csavarják ki a kezünk közül. C-ben még volt pointer aritmetika, Javában már nincs. C-ben még a programozó feladata volt a dinamikus memóriából a szemét felszabadítása, Javában ezt már a GC végzi. Lassan ott tartunk, hogy már ciki ciklust írni. A menő C# programozók LINQ-t, a menő Java programozók stream-eket használnak és az ott elérhető függvénnyel paraméterezhető függvényeket, és már alig írnak ciklusokat.

A másik oldalon, a deklaratív programozási nyelvek a MIT kérdésre teszik a hangsúlyt. Csak deklarálni kell a megoldandó problémát, azaz MIT kell megoldani, és a keretrendszer kitalálja, hogy hogyan, melyik algoritmus segítségével. Ez könnyűnek hangzik és tényleg, könnyebb is deklaratív oldalon programozni, tisztább, szárazabb érzés. Ugyanakkor ezen az oldalon is vannak gondok, nincs mellékhatás, ami első hallásra jó hír, de ha belegondolunk, a képernyőre írás is mellékhatás. Nincs ciklus, csak rekurzió, és absztrakció helyett magasabb rendű függvények vannak. Ráadásul, az imperatív nyelvek sokkal jobban elterjedtek.

Tehát az imperatív nyelvek elterjedtebbek, de deklaratív módon könnyebb programozni, mindkét oldalnak van előnye és hátránya, ezért a két oldal közeledik egymáshoz. Például: deklaratív oldalon egyre több nyelvben jelenik meg az objektum és az osztály fogalma; imperatív oldalon használhatunk lambda kifejezést.

A DIP elvet betartva egy újabb deklaratív elem jelenik meg az imperatív oldalon: Mivel az absztrakció a MIT kérdésre ad választ, ezért rákényszerülünk, hogy a MIT kérdés hangsúlyosabb legyen az imperatív oldalon is.

1.2. Tervezési minták és a DIP

A tervezési alapelvek magasabb absztrakciós szinten vannak, mint a tervezési minták. Ez azt jelenti, hogy a tervezési mintákban gyakran fülön csíphetünk egy-egy tervezési alapelvet. Először nézzük meg az egyes absztrakciós szinteket a nagyon elvonttól az igazán konkrét felé:

- „A program kódja állandóan változik.”: Ez a fő alapelvünk, programozás technológiában minden más elvet ebből vezethetünk le.
- Dolgok szétválasztásának elve: „Separation of Concerns”, „Amit szét lehet választani, azt érdemes szétválasztani”.
- Kontrol megfordításának elve, Inversion of Control (IoC), ne a program várjon eseményre, az esemény hívja a programot.
- OOP alapelvei, mint az öröklődés, az egységbezárás, a többalakúság és az absztrakció használata.
- Tervezési alapelvek, ide tartoznak a SOLID elvek, a GOF1-2, de a Hollywood elv is.
- Tervezési minták, ide tartoznak a GOF könyv mintái.
- Osztálykönyvtárak (angolul: library), amit egy régi szóhasználat, napjainkban inkább a névtér, vagy a csomag név használata elterjedt egy-egy feladat megoldásának eszközkészlete.
- Újrahasznosítható forráskód, amit mondjuk könnyen használhatunk Android-os, és mondjuk webes környezetben is.
- Megrendelőre szabott specifikus forráskód, amely legvégül, a sok réteg legalján lefut.

Persze ez a felsorolás csak a szerzők meggyőződését tükrözi. Más források más-más absztrakciós szinteket tartalmazhatnak, de abban elég nagy egyetértés van, hogy a tervezési alapelvek magasabb absztrakciós szinten állnak, mint a tervezési minták.

Mindenki egyetért abban, hogy Megfigyelő tervezési mintában szépen tetten érhető a Hollywood tervezési alapelv, „ne hívj, majd mi hívunk”. Sok-sok hasonló példát hozhatnánk itt fel. Ebben a részben azt tárgyaljuk, hogy a DIP hogyan jelenik meg az egyes tervezési mintákban.

Majd minden tervezési mintáról elmondható, hogy szétválaszt valamit valamitől. Mivel a DIP-nek is a feladata, ezért nagyon gyanús, hogy DIP több tervezési mintában is tetten érhető.

Nézzük meg milyen kérdésekre kell feltennünk, ezen vizsgálat során?

- Minden nyíl absztrakcióra mutat? Ha igen, akkor valószínűleg egy jó tervet látunk, ahol a tervező erőskezd, ugyanakkor a programozók nem tudnak nemet mondani a tervezői túlkapásokra (lásd a „Mikor ne használjuk a DIP tervezési alapelvet” részt). Mindenesetre az ilyen tervekben könnyen találhatunk DIP-et. Megjegyzés: Ha nem minden nyíl mutat absztrakcióra, attól még lehet a tervben DIP, sőt valahol egészségesebb is, ha erre nem a válasz, mert az mutatja, hogy a programozóknak is volt beleszólásuk a tervbe.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Ha igen, ez már majdnem biztos DIP. Ugyanakkor lehet, hogy ugyanaz az osztály mutat HAS-A és IS-A kapcsolattal az absztrakcióra. Ez az eset nem DIP, mivel a DIP lényege, hogy szétválassza a HAS-A és az IS-A oldalt. Ha ez a két oldal egy és ugyanaz, vagy erős egyéb kapcsolat van köztük, akkor ez nem DIP.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat, és ez a két oldal szétválík? Ha igen, akkor ez DIP! Ha a két oldal csak nagyból válik szét, azaz nem csak a HAS-A oldalról jön hívás az absztrakción keresztül az IS-A oldal felé, hanem visszafelé is, mondjuk egy callback-en

keresztül, akkor ez vagy DIP vagy nem DIP. Ha a visszafelé kommunikáció több, mint valamilyen visszajelzés, akkor ez nem DIP. Ha visszafelé kommunikáció csak valamiféle visszajelzés, akkor ez DIP. Nyilván itt van értelmezési szabadság, hogy ki mit gondol „csak valamiféle visszajelzésnek”, de ez egy absztrakt alapelvénél teljesen természetes. Ebben a jegyzetben akkor mondjuk azt, hogy ez „csak valamiféle visszajelzés”, ha az IS-A oldalról egy egyszerű asszociáció mutat vissza a HAS-A oldalra.

- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat, és ez a két oldal szétválik, és a HAS-A oldal a kliens kód? Ha igen, akkor a legszigorúbb értelmezés szerint is ez egy DIP. Már csak az a kérdés, hogy mit jelent a kliens kód? A kliens kód alatt olyan kódot értünk, amiről nem tudunk semmit, csak azt, hogy használni akarja a tervezés alatt álló kódrészlet szolgáltatásait.

Ebben a fejezetben megvizsgálunk két tervezési mintát, amiről tudván tudjuk, hogy a legfontosabb tulajdonságuk két dolog szétválasztása, ez a Híd és a Megfigyelő. Ezek után a GOF könyvben felsorolt tervezési mintákat vesszük szép sorba.

1.2.1. Szétválasztás a fókuszban: Híd és Megfigyelő

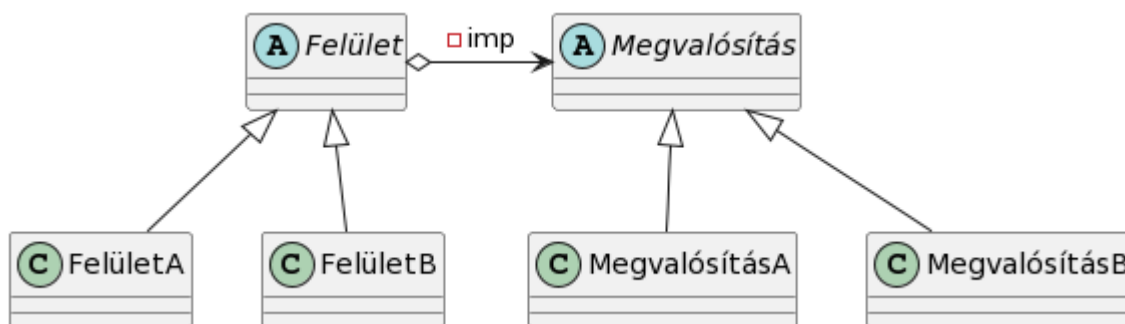
Vizsgáljunk meg néhány jól ismert tervezési mintát, hogy megjelenik-e bennük a DIP, és ha igen, hogyan. Kezdjük azokkal, amikről jól tudjuk, hogy szétválasztanak valamit, hiszen mint láttuk a DIP-nek van köze a szétválasztáshoz.

Két tervezési minta van a köztudatban, amik direkt szétválasztásra jók:

- Híd (angolul: Bridge): a híd tervezési minta a különböző lehetséges felületeket és a különböző lehetséges implementációkat választja szét.
- Megfigyelő (angolul: Observer): a megfigyelő tervezési minta szétválasztja a megfigyeltet és a megfigyelőket.

1.2.2. Híd

Nézzük meg a Híd (angolul: Bridge) tervezési mintának az UML ábráját:



PlantUML szkriptje:

```
@startuml
abstract class Felület { }
abstract class Megvalósítás { }
class FelületA {}
class FelületB {}
```



```

class MegvalósításA { }
class MegvalósításB { }

Felület <|-- FelületA
Felület <|-- FelületB

Megvalósítás <|-- MegvalósításA
Megvalósítás <|-- MegvalósításB

Felület o-right-> Megvalósítás : -imp

@enduml

```

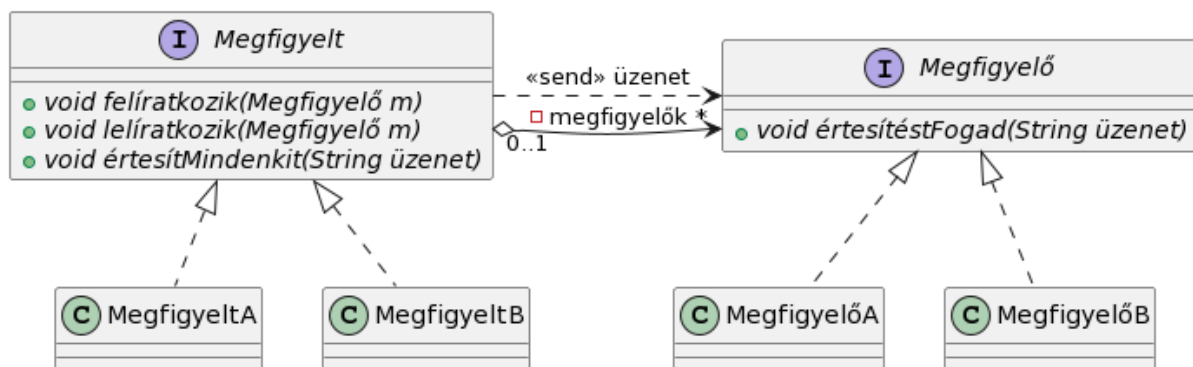
Tegyük fel a kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen! Ennyi elég is? Ha minden nyíl absztrakcióra mutat, akkor automatikusan kijelenthetjük, hogy a DIP valamilyen előfordulásával van dolgunk? Nem. A következő finomabb kérdést is fel kell tennünk:
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A Megvalósítás absztrakt osztályra HAS-A és két IS-A kapcsolat is mutat. Már csak egy kérdés maradt.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a két oldal közt nincs semmilyen más kapcsolat. Tehát, ez bizony DIP! Ezzel az absztrakt osztállyal választjuk szét a lehetséges felületeket és a lehetséges megvalósításokat.

Még egy kérdés marad: Minek a másik absztrakt osztály, a Felület? Az csak azért kell, hogy lehessen több konkrét gyermeke, mert hiszen több lehetséges felületet választunk el több lehetséges megvalósítástól.

1.2.3. Megfigyelő

Vizsgáljuk meg a Megfigyelő (angolul: Observer) tervezési minta UML ábráját:



PlantUML szkriptje:

```

@startuml
interface Megfigyelő {
+ {abstract} void értesítéstFogad(String üzenet)
}

```

```

interface Megfigyelt {
+ {abstract} void felíratkozik(Megfigyelő m)
+ {abstract} void leíratkozik(Megfigyelő m)
+ {abstract} void értesítMindenkit(String üzenet)
}

class MegfigyelőA {}
class MegfigyelőB {}
class MegfigyeltA { }
class MegfigyeltB { }

Megfigyelő <|.. MegfigyelőA
Megfigyelő <|.. MegfigyelőB
Megfigyelt <|.. MegfigyeltA
Megfigyelt <|.. MegfigyeltB

Megfigyelt "0..1" o-right-> "*" Megfigyelő : -megfigyelők

Megfigyelt ..> Megfigyelő : <<send>> üzenet

@enduml

```

Mint látható, ez egy teljesen egyszerű toló (angolul: push) változata a Megfigyelő tervezési mintának. Egy megfigyeltnek lehet több megfigyelője. A megfigyelők fel- és leíratkozhatnak, és ha valami érdekes történik, akkor arról értesítést kapnak a értesítéstFogad(String üzenet) metódus segítségével. A Megfigyelő minta hagyományos lerajzolásánál nem tesszük ki a <<send>> nyilat, de itt szeretnénk volna kihangsúlyozni, hogy csak az egyik oldalról, a HAS-A oldalról, megy kommunikáció a másik oldalra.

Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a Megfigyelő interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, más kapcsolat nincs a két oldal közt, a Megfigyelő interfész elválasztja egymástól a konkrét megfigyelőket a konkrét megfigyeltektől. Valahol a kliens kódban a konkrét megfigyelőket regisztráljuk a megfigyelőhöz. A konkrét megfigyelő semmit se tud a megfigyeltről, annak felületéről, csak azt tudja, hogy ha majd történik egy érdekes esemény, akkor majd megkapja ennek az eseménynek a leírását. Mit látjuk, ez egy szép DIP!

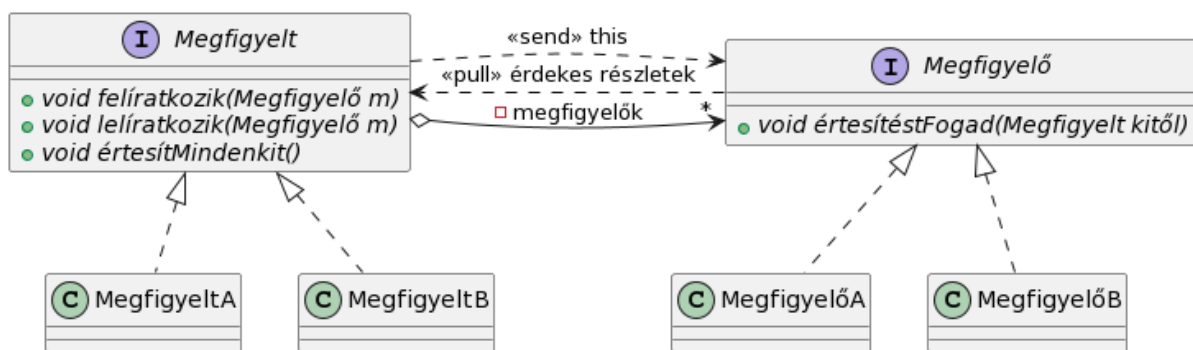
Talán érdemes kiemelni, hogy a két oldal közt nem kötelező egy-sok kapcsolat van. A sok oldalt a nyílon szereplő csillag (*) jelzi, azaz egy Megfigyelt-hez, több Megfigyelő is tartozhat. A csillag jelentése 0..n, azaz egy nem kötelező kapcsolat. Megfigyelt létezhet Megfigyelő nélkül is. A másik oldalon azt látjuk, hogy 0..1, ami azt jelenti, hogy a másik irányból sem kötelező a kapcsolat: létezhet

Megfigyelő anélkül is, hogy megfigyelne bárkit is. Ha kötelező a kapcsolat, akkor a sok oldalon ezt kell írunk: 1..n, vagy 1..*, illetve az egy oldalon ezt kell írunk: 1. Összefoglalva az egy-sok kapcsolatokat:

- A "0..1" --> "*" B, vagy: A "0..1" --> "0..n" B, vagy: A "0..1" --> "0..*" B: mindkét oldal nem kötelező;
- A "1" --> "*" B, vagy: A "1" --> "0..n" B, vagy: A "1" --> "0..*" B: B nincs A nélkül, de A lehet B nélkül, tehát egyik oldalról kötelező;
- A "0..1" --> "1..*" B, vagy: A "0..1" --> "1..n" B: B lehet A nélkül, de A csak úgy lehet, ha ismer legalább 1 B-t, tehát egyik oldalról kötelező;
- A "1" --> "1..*" B, vagy: A "1" --> "1..n" B: B nincs A nélkül, és A sincs B nélkül, tehát mindkét irányban kötelező.

Mivel UML-ben csak ritkán akarjuk kihangsúlyozni egy kapcsolat kötelező voltát, ezért az egy oldalra ritkán írunk bármit is, és a sok oldalra is csak sima csillagot szoktunk írni, ha kötelező, ha nem. A következő ábráról már elhagyjuk az egy oldalon a 0..1 jelölést.

Az előző változat egy szép DIP példa. A Megfigyelő mintának van egy másik változata is, a húzó (angolul: pull) változat. Ennek esetén van visszahívás! Le kell húzni a Megfigyelőből azokat az érdekes részleteket, amik a Megfigyelőt érdekli. Ehhez egy másik változatát szoktuk használni az értesítés fogadásnak: értesítéstFogad(Megfigyelő kitől). A kitől referencián keresztül tudja lehúzni az érdekes részleteket a Megfigyelőből. Lássuk ennek a változatnak az UML ábráját:



PlantUML szkriptje:

```

@startuml
interface Megfigyelő {
+ {abstract} void értesítéstFogad(Megfigyelt kitől)
}

interface Megfigyelt {
+ {abstract} void feliratkozik(Megfigyelő m)
+ {abstract} void leíratkozik(Megfigyelő m)
+ {abstract} void értesítMindenkit()
}

class MegfigyelőA {}

```

```
class MegfigyelőB {}
class MegfigyeltA { }
class MegfigyeltB { }

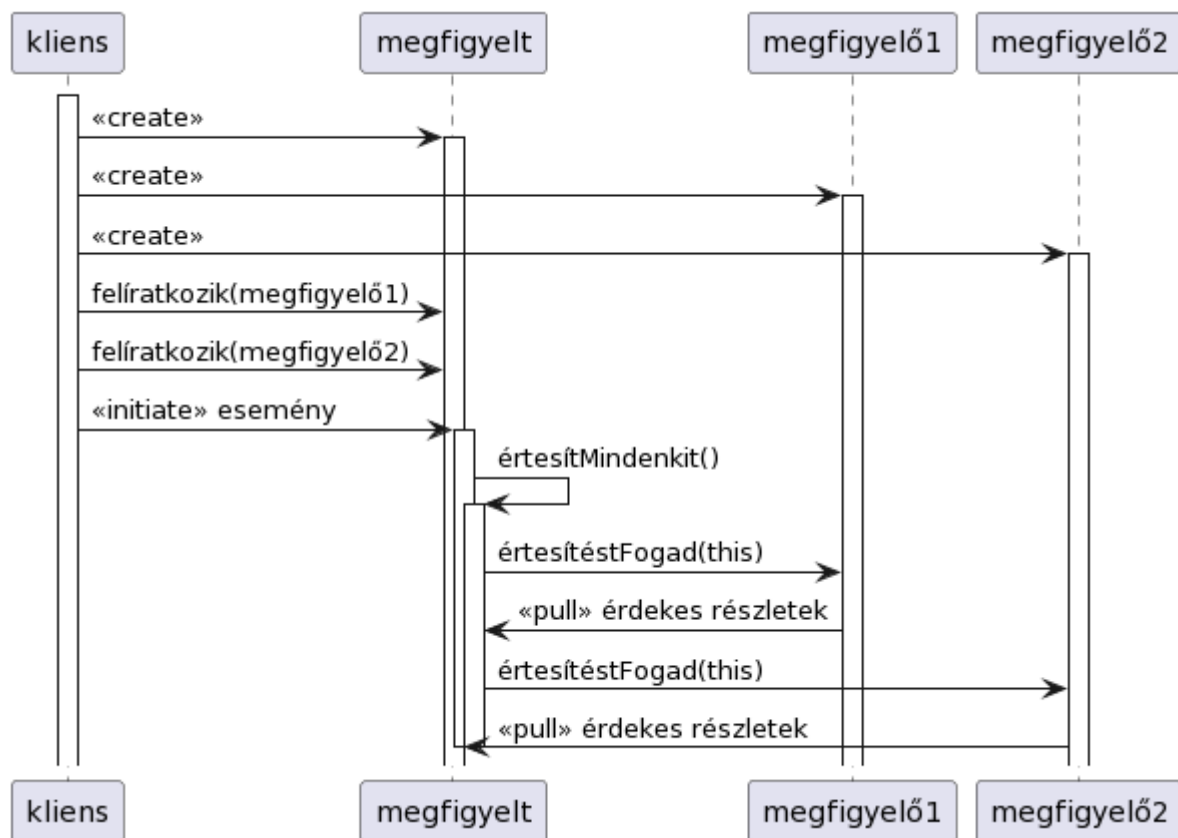
Megfigyelő <|.. MegfigyelőA
Megfigyelő <|.. MegfigyelőB
Megfigyelt <|.. MegfigyeltA
Megfigyelt <|.. MegfigyeltB

Megfigyelt o-right-> "*" Megfigyelő : -megfigyelők
Megfigyelt ..> Megfigyelő : <<send>> this
Megfigyelő ..> Megfigyelt : <<pull>> érdekes részletek

@enduml
```

Mint látjuk, itt van visszafelé kommunikáció. Ugyanakkor ez csak egy egyszerű barátság, ami belefér a DIP fogalmába, így ez a változat is DIP. Ugyanakkor ez a változat erősebb csatoltságot eredményez: mind a két oldal tud valamit a másikról. Tehát jobbnak tűnik az első változat.

Annyi kérdés maradt még, hogy a Megfigyelő minta hagyományos UML osztálydiagramján nincs rajta se a <<send>>, se a <<pull>> nyíl, pedig ezek fontos részletet írnak le, a két oldal kommunikál egymással. Akkor mégis, hol vannak ezek az érdekes részletek a hagyományos ábrázolásnál. A válasz az, hogy ezeket egy másik ábra mutatja, az úgynevezett szekvencia diagram. Lássuk a fent vázolt kommunikációt szekvencia diagram segítségével:



PlantUML szkriptje:

@startuml

activate kliens

kliens -> megfigyelt : <<create>>

activate megfigyelt

kliens -> megfigyelő1 : <<create>>

activate megfigyelő1

kliens -> megfigyelő2 : <<create>>

activate megfigyelő2

kliens -> megfigyelt : feliratkozik(megfigyelő1)

kliens -> megfigyelt : feliratkozik(megfigyelő2)

kliens -> megfigyelt : <<initiate>> esemény

activate megfigyelt

megfigyelt -> megfigyelt : értesítMindenkit()

activate megfigyelt

megfigyelt -> megfigyelő1 : értesítéstFogad(this)

megfigyelő1 -> megfigyelt : <<pull>> érdekes részletek

megfigyelt -> megfigyelő2 : értesítéstFogad(this)

megfigyelő2 -> megfigyelt : <<pull>> érdekes részletek

deactivate megfigyelt

deactivate megfigyelt

@enduml

A továbbiakban ritkán használunk szekvencia diagramot, inkább, ha egy kicsit szabálytalanul is, de az osztály diagramon fogjuk feltüntetni a résztvevők közti kommunikációt.

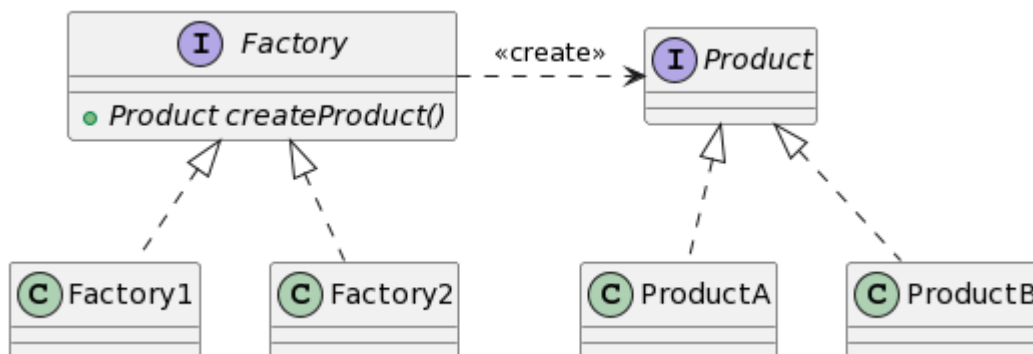
1.3. Létrehozási tervezési minták

A létrehozási tervezési minták közös tulajdonsága, hogy valamilyen objektum létrehozásának folyamatát zárják egységbe. Miért? Egyrészt, hogy a sok-sok new hívásoktól megtisztítsuk a programunkat és ha változtatni kell a létrehozás módján, akkor azt megtehetjük egy helyen. Másrészt, hogy a létrehozás előtt-után elvégezhesük azokat a tevékenységeket, amik még szükségesek. Ez lehet egyszerű log írás, de az is lehet, hogy egy nagy objektum több kisebből áll, amiket szintén létre kell hozni, esetleg egy tervrajzot követve kell létrehozni az objektumot.

Akármi is az ok, az objektum példányosítást gyakran érdemes egységbe zárni. Ráadásul ezzel szétválasztjuk a példányosítás folyamatát és a létrejövő példányt, azaz helye van a DIP-nek. Vizsgáljuk meg az egyes tervezési mintákat, tényleg alkalmaznak-e DIP alapelvet.

1.3.1. Absztrakt gyár

Absztrakt gyár (angolul: Abstract factory) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
interface Factory {
+{abstract} Product createProduct()
}
interface Product { }
class Factory1 { }
```

```

class Factory2 {}

class ProductA { }

class ProductB { }

Factory <|.. Factory1

Factory <|.. Factory2

Product <|.. ProductA

Product <|.. ProductB

Factory .right.> Product : <<create>>

@enduml

```

Eddig majd minden példa magyar nyelvű volt, de ez direkt angol nyelvű, mert egy fontos szóhasználatra szeretnénk felhívni a figyelmet. A gyárban általában a metódusok neve a create szóval kezdődik, hogy kihangsúlyozzuk, hogy ezek a metódusok valamit gyártanak.

Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A Product interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Product interfész elválasztja egymástól a konkrét gyárakat és a konkrét termékeket. Egy konkrét gyár tudja, hogyan kell egy konkrét terméket gyártani. A termék nem tud semmit a gyákról. Tehát ez egy DIP.

A Product-ra mutató HAS-A kapcsolat a leggyengébb, egy asszociáció, vagy más néven barátság, aminek ráadásul sztereotípiája is van: <<create>>. Ez azt fejezi ki, hogy a Factory tud készíteni, nyilván a createProduct() metódussal, Product példányt.

Itt a gyártás folyamatát választjuk el magától a terméktől. A konkrét gyárak konkrét terméket gyártanak, de hogy melyiket, az a felhasználó szempontjából mindegy, mert csak a termékek közös felületét ismeri. Nyilván, a konkrét termékek viselkedése eltér, mégis a felhasználónak mindegy, melyik konkrét terméket kapja. Ez a látszólagos ellentmondás úgy oldódik fel, ha a különböző viselkedés egy döntés eredménye, amit végig kell vinnünk a programon.

Illetve úgy is feloldható ez a látszólagos ellentmondás, ha a gyár nem csak egy terméket, hanem több terméket is gyárt és azoknak kompatibilisnek kell lenniük egymással. Úgy szoktok mondani, hogy az Opel gyár Opel termékeket gyárt, az Audi Audit. Az Opel termékek egymáshoz illenek, együtt működőképesek. Ugyan ez igaz az Audi termékekre. Ugyanakkor, ha az egyik termék Opel, a másik Audi, az biztos, hogy működőképes autót eredményez.

Nagyon érdekes megnézni a Wikipédia oldalon található leírást, hogy mit mitől választ el ez a minta (https://hu.wikipedia.org/wiki/Absztrakt_gy%C3%A1r_programtervez%C3%A9si_minta, megtekintve: 2023.08.03.):

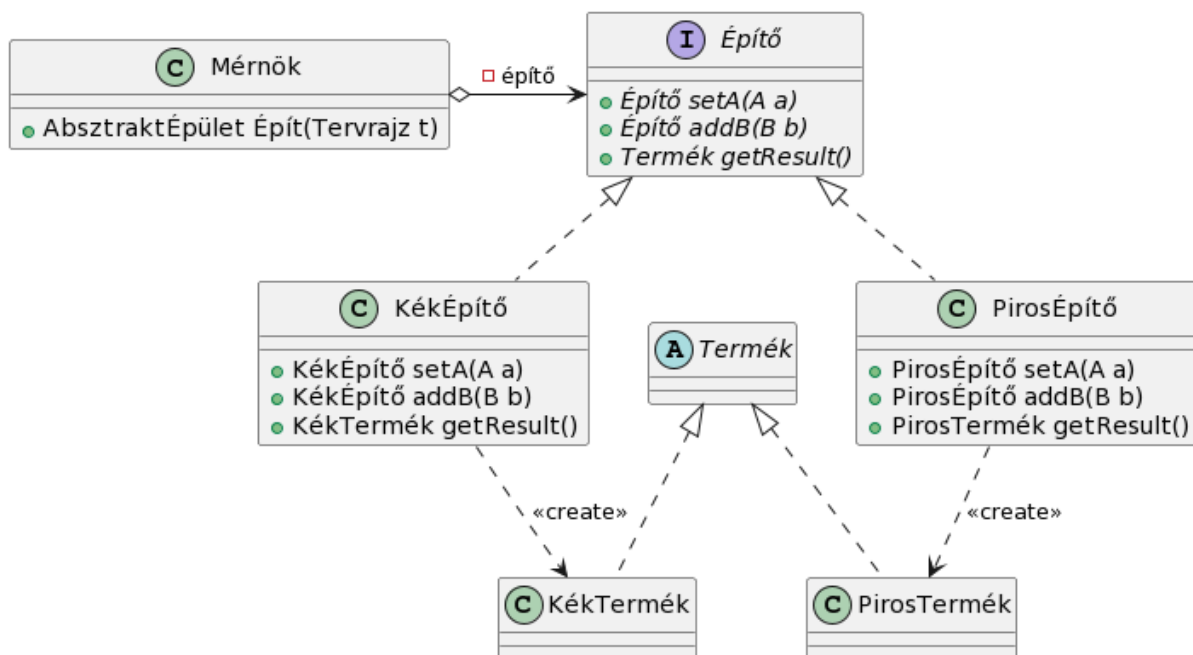
„Az Absztrakt gyár (angolul Abstract factory) programtervezési minta módot nyújt arra, hogy egységbe zárjuk közös témához kapcsolódó egyedi gyártó metódusok egy csoportját anélkül, hogy specifikálnák azok konkrét osztályait. Normál használatban, a kliens szoftver létrehozza az absztrakt gyár egy konkrét implementációját, és aztán a gyár általános interfészét használja a témához kapcsolódó konkrét objektumok létrehozásához. A kliens nem tudja (vagy nem törődik vele), milyen

konkrét objektumokat kap ezekből a belső gyárakból, mivel csak a termékeik általános interfészét használja. Ez a tervezési minta szétválasztja egymástól objektumok egy csoportjának implementációját azok általános használatától és objektum összetételre hagyatkozik, mivel az objektumok létrehozása olyan metódusokban van implementálva, amik a gyár interfészén vannak ismertté téve számára.”

Szerencsére nagyon hasonló a látásmódunk ehhez, bár a Wikipédiás megfogalmazás fordítás gyanús, nem teljesen gördülékeny szöveg.

1.3.2. Építő

Az Építő (angolul: Builder) tervezési minta UML ábrája:



PlantUML szkriptje:

```

@startuml
class Mérnök {
+ AbsztraktÉpület Épít(Tervrajz t)
}
interface Építő {
+ {abstract} Építő setA(A a)
+ {abstract} Építő addB(B b)
+ {abstract} Termék getResult()
}
class KékÉpítő {
+ KékÉpítő setA(A a)
}
class PirosÉpítő {
+ PirosÉpítő setA(A a)
}
class Termék {
}
class KékTermék {
}
class PirosTermék {
}
Mérnök --> Építő : építő
Építő <|.. KékÉpítő
Építő <|.. PirosÉpítő
KékÉpítő ..> KékTermék : «create»
PirosÉpítő ..> PirosTermék : «create»
Termék <|.. KékTermék
Termék <|.. PirosTermék
  
```



```

+ KékÉpítő addB(B b)
+ KékTermék getResult()
}

class PirosÉpítő {
+ PirosÉpítő setA(A a)
+ PirosÉpítő addB(B b)
+ PirosTermék getResult()
}

abstract class Termék {}

class KékTermék {}

class PirosTermék {}

Mérnök o-right-> Építő : -építő
Építő <|.. KékÉpítő
Építő <|.. PirosÉpítő
Termék <|.. KékTermék
Termék <|.. PirosTermék

KékÉpítő ..> KékTermék : <<create>>
PirosÉpítő ..> PirosTermék : <<create>>

@enduml

```

Az ábrán több érdekességet is felfedezhetünk, de előtte tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! Az Építő interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, az Építő interfész elválasztja egymástól a mérnököt a konkrét építőktől. A mérnök egy tervrajz alapján tudja, hogy melyik lépés melyik másik lépés után jön, de nem tudja, hogy ezeket konkrétan hogyan kell végrehajtani. A konkrét építő nem ismeri a tervrajzot, a lépések sorrendjét, de a lépések végrehajtásának módját igen. Az építkezés eredménye egy termék. Mivel a két oldal szétválik, ezért ez egy DIP.

Tehát, az Építő interfész segítségével választjuk el a tervrajzot olvasni képes Mérnök osztályt, és a konkrét kivitelezést végző KékÉpítő és PirosÉpítő osztályoktól. A mérnök olvassa a tervrajzot és utasítja az építőt, hogy mit csináljon. A példában lehet csinálni setA és setB lépéseket, amik nem túl izgalmas. Nyilván csak azt akartuk jelezni, hogy az építés folyamán többféle lépés lehet. Az építkezés végén a getResult metódus adja vissza az elkészült terméket.

A setA és setB körül több izgalmas dolgot is látunk. Először is, ezek a metódusok Építő típust adnak vissza. Ez azt eredményezi, hogy láncban lehet őket hívni, például:

```
new KépÉpítő().setA(a).setB(b).getResult();
```

Ez nagyon jellemző az Építő tervezési mintára. Megjegyezzük, hogy ez látszólag ellent mond a legkisebb tudás elvének, ami szerint ez a fajta metódus hívás elkerülendő:

```
o1.getBarát().getCimbora().getHaver().valamiHasznos();
```

A legkisebb tudás elve azt mondja ki, hogy minden osztálynak csak a saját baráti köréhez van köze, a barátjának a barátjára hivatkozás kerülendő. Ezt gyakran úgy fordítjuk le a szintaxis szintjére, hogy kerüljük a hosszú hívási láncokat, az olyan kifejezéseket, amiben több, mint 1 pont van.

Habár ebben a kifejezésben több pont 1 pont van:

```
new KépÉpítő().setA(a).setB(b).getResult();
```

mégis megfelel a legkisebb tudás elvének, hiszen nem hivatkozok egy másik osztályra és abból megint egy másikba, hanem maradok egy osztályon belül maradunk, ahol mindent ismerünk, nincs veszély.

A másik érdekesség, hogy a setA és a setB metódus visszatérési típusa az Építőben Építő, a KépÉpítőben KépÉpítő, és, csodák-csodája, a PirosÉpítőben PirosÉpítő. Hasonlóan, a getResult visszatérési típusa az Építőben Termék, a KépÉpítőben KékTermék, és valószínűleg már mindenki kitalálta, a PirosÉpítőben PirosTermék. Azaz, a gyermekosztályban a visszatérési típus az eredeti visszatérési típus gyermekosztálya. Ez Java5 előtt hiba volt. Java5-től viszont bejött a kovariáns visszatérési típus (angolul: covariant return type), ami ezt lehetővé teszi. Ez sok felesleges típuskényszerítéstől óvja meg az embert.

A harmadik érdekesség, hogy bizony itt van két párhuzamos osztályhierarchia: A Termék és az Építő hierarchia párhuzamos. Ha most felvennék egy SárgaTermék típust a Termék alá, akkor előbb utóbb elvárás lesz, hogy legyen egy SárgaÉpítő osztály is az Építő alatt. Ez bizony két párhuzamos osztályhierarchia.

Minden párhuzamos osztályhierarchia tervezési hibára utal! Arra kell törekednünk, hogy ne legyenek ilyen esetek a tervünkbe! Nyilván egyszerű a javítás, a két hierarchiát össze kell vonni!

Ugyanakkor ez nem mindig lehetséges. Gondoljunk arra, hogy az egyik termékünk egy PDF dokumentum. Nyilván, a PDF dokumentum osztályába nem nyúlhatok bele, valószínűleg nem is lehet örökölni belőle. Igazán nincs más választásom, mint, hogy külön osztályban legyen a PDFÉpítő.

Az ilyen esetek elkerülésére találták ki .NET framework-ben a osztály kiegészítését. Például a String osztály egy lepecsételt osztály, semmilyen módon nem nyúlhatok bele, de a this string szintaxissal kiegészíthetem. Példa C# nyelven:

```
public static class MyExtensions

{
    public static int WordCount(this string str)
    {
        return str.Split(new char[]{' '},
            StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Sajnos ilyen lehetőség Java nyelven nincs, de van egy Java fordító plugin, aminek neve Manifold, amivel mégis megoldható az osztály kiegészítés. Lásd: www.manifold.systems. Konkrét példa:

```
import manifold.ext.api.*;
@Extension
public class MyStringExtension {
    public static void int wordCount(@This String str) {
        String[] wordArray = str.trim().split("\\s+");
    }
}
```

```

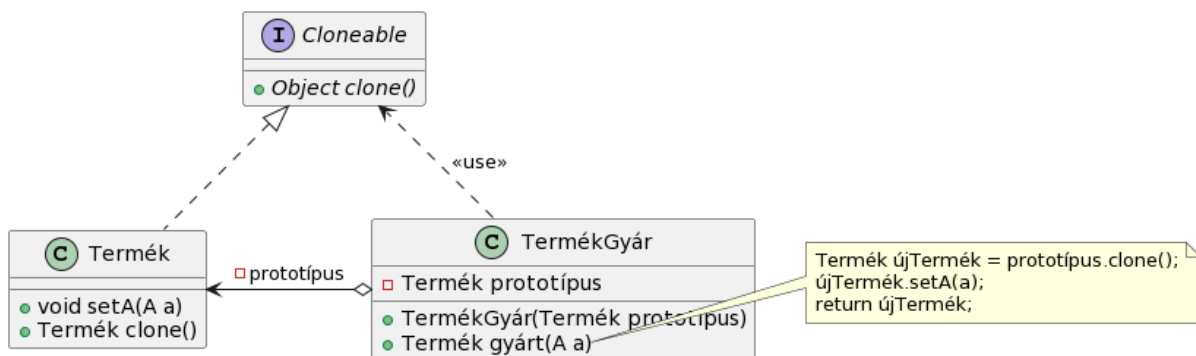
    return wordArray.length;
}
}

```

Ez azt jelenti, hogy az Építő tervezési minta rossz? Ha az Építő tervezési mintával párhuzamos osztály hierarchiák jönnek létre, akkor igen, jobb háromszor végiggondolni, hogy tényleg a legjobb megoldás-e az Építő? Viszont, ha mindig ugyanazt a típust építjük, csak más-más módszerrel, akkor nem lehet vita az Építő minta hasznosságáról. Pl. nyaralás tervezésnél az egyik építő mindig 3 csillagos szállodába foglal szállást, a másik 4-5 csillagosba, akkor mindkét építő nyaralást épít, csak más-más módon. Ilyen esetben nincs párhuzamos Építő-Termék osztályhierarchia, nyugodtan használható az Építő.

1.3.3. Prototípus

A Prototípus (angolul: Prototype) tervezési minta UML ábrája:



PlantUML szkriptje:

```

@startuml
interface Cloneable {
    +{abstract} Object clone()
}
class TermékGyár {
    -Termék prototípus
    +TermékGyár(Termék prototípus)
    +Termék gyárt(A a)
}
class Termék {
    +void setA(A a)
    +Termék clone()
}
note right of TermékGyár::gyárt
    Termék újTermék = prototípus.clone();
endnote
  
```

```
újTermék.setA(a);
```

```
return újTermék;
```

```
end note
```

```
TermékGyár o-left-> Termék : -prototípus
```

```
TermékGyár ..> Cloneable : <<use>>
```

```
Termék .up.|> Cloneable
```

```
@enduml
```

Tegyük fel a szokásos kérdéseket:

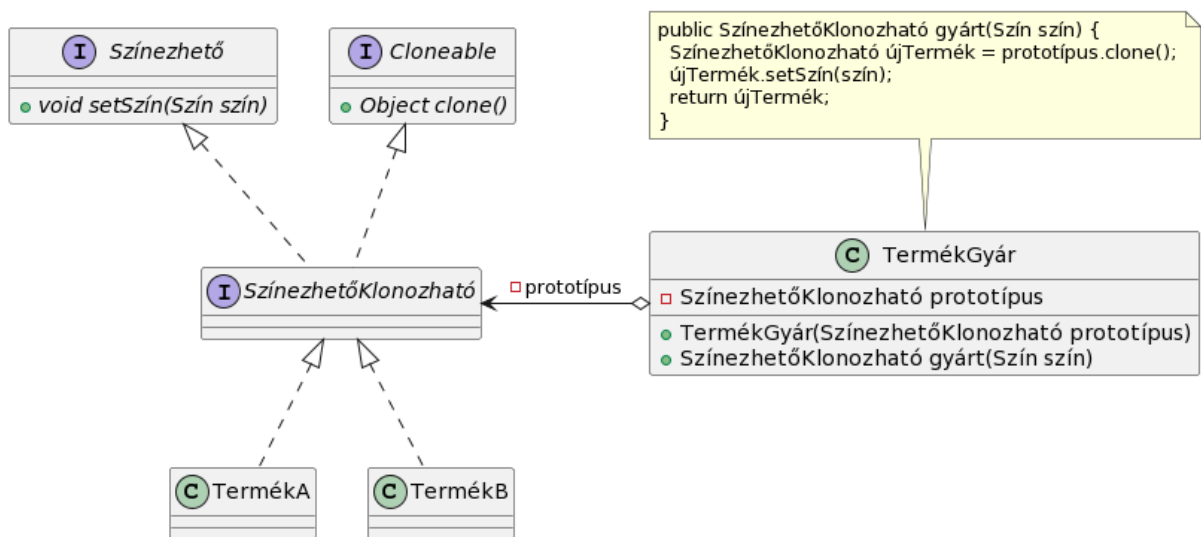
- Minden nyíl absztrakcióra mutat? Nem! A TermékGyárból egy HAS-A kapcsolat mutat a Termék osztályra, és a Termék osztály nem absztrakt. Ettől még lehet DIP.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! Az Clonable interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Nem, mert a HAS-A és a IS-A oldal közt ott van a „-prototípus” HAS-A kapcsolat. Sajnos ez nem DIP. DIP akkor lenne, ha Termék interfész vagy absztrakt osztály lenne, de a Termék konkrét osztály. Tehát ez nem DIP!

A fentieket nézzük meg részletesen is. A Cloneable interfészre mutat egy asszociáció (vagy más néven barátság), azaz egy HAS-A kapcsolat, és egy IS-A kapcsolat is, azaz úgy tűnik, hogy van DIP. Ugyanakkor az az asszociáció talán nem teljesen indokolt, hiszen habár a TermékGyár tényleg hívja a Cloneable interfészben lévő clone metódust, de azt a Termék osztályra mutató prototípus referencián keresztül hívja, amit azért tehet meg, mert a Termék osztály megvalósítja a Cloneable interfészt. Tehát nem tűnik jogosnak a barátság használata a TermékGyár és a Clonable közt. Szerencsére körkörös hivatkozás nincs, tehát tervezési hiba nincs az ábrán.

Persze nagyon könnyen lehetne DIP-et bevezetni. Csak annyi kellene, hogy a Termék legyen absztrakt és legyen több gyermekosztálya, de általában nem így használjuk a Prototípus tervezési mintát. Általában úgy használjuk, hogy a gyártás költségének csökkentésére van egy prototípus, amit egy az egyben lemásolok a clone metódus segítségével, majd egy-egy részletet megváltoztatok rajta a megrendelő igényeinek megfelelően. Ehhez ismernem kell azt a néhány érdekes részletet, amit a gyár módosítani képes.

Képzeljünk el egy 3D nyomtató gyárat, ami kap egy prototípust, azt képes lemásolni, és egy megszólalásig hasonlót kinyomtatni, majd az eljárás végén olyan színűre (ez az érdekes részlet) festi, amelyet a kedves vevő választ.

A fenti ábrából úgy lesz DIP, és így lesz jobb a terv, ha az érdekes részleteket is kiemeljük egy interfészbe. Lássuk ezt az UML ábrát, ahol az érdekes részlet a szín lesz:



PlantUML szkriptje:

@startuml

interface Cloneable {

+{abstract} Object clone()

}

interface Színezhető {

+{abstract} void setSzín(Szín szín)

}

interface SzínezhetőKlonozható {

}

class TermékA {}

class TermékB {}

class TermékGyár {

-SzínezhetőKlonozható prototípus

+TermékGyár(SzínezhetőKlonozható prototípus)

+SzínezhetőKlonozható gyárt(Szín szín)

}

note top of TermékGyár

public SzínezhetőKlonozható gyárt(Szín szín) {

SzínezhetőKlonozható újTermék = prototípus.clone();

újTermék.setSzín(szín);

```

    return újTermék;
}

end note

TermékGyár o-left-> SzínezhetoKlonozható : -prototípus

SzínezhetoKlonozható .up.|> Cloneable

SzínezhetoKlonozható .up.|> Színezheto

SzínezhetoKlonozható <|.. TermékA

SzínezhetoKlonozható <|.. TermékB

@enduml

```

Ezen az ábrán a SzínezhetoKlonozható interfész már valóban elválasztja egymástól a termék gyárat és a konkrét termékeket. Ez már DIP! Ugyanakkor, az ábra komplikáltabb lett!

A tervezőnek azt az álláspontot kell képviselnie, hogy a projekten ezt a szép tervet használjuk, ahol a DIP teljes szépségében ragyog. A vezető programozónak, aki átnézi elfogadás előtt a tervet, azt az álláspontot kell képviselnie, hogy az első ábra is teljesen rendben van, mert úgylis csak egy termék van, és ha lenne több, akkor majd kiemeljük a közös részeket egy absztrakt űsbe, amit majd használ a TermékGyár. Erre nyilván a tervező azt válaszolja, ha előre látható, hogy lesz több termék, miért ne készüljünk fel erre már a tervben, hogy könnyű legyen új terméket hozzáadni. Erre nyilván azt mondja a vezető fejlesztő, hogy egyáltalán nem biztos, hogy lesz több termék, és amíg ez ki nem derül, minek bonyolítsuk el a tervet. Nem minden csapattag zseni, hogy átlássa ezt a pókhálót. Erre a tervező azt mondja Erre a vezető fejlesztő azt mondja Satöbbi, satöbbi.

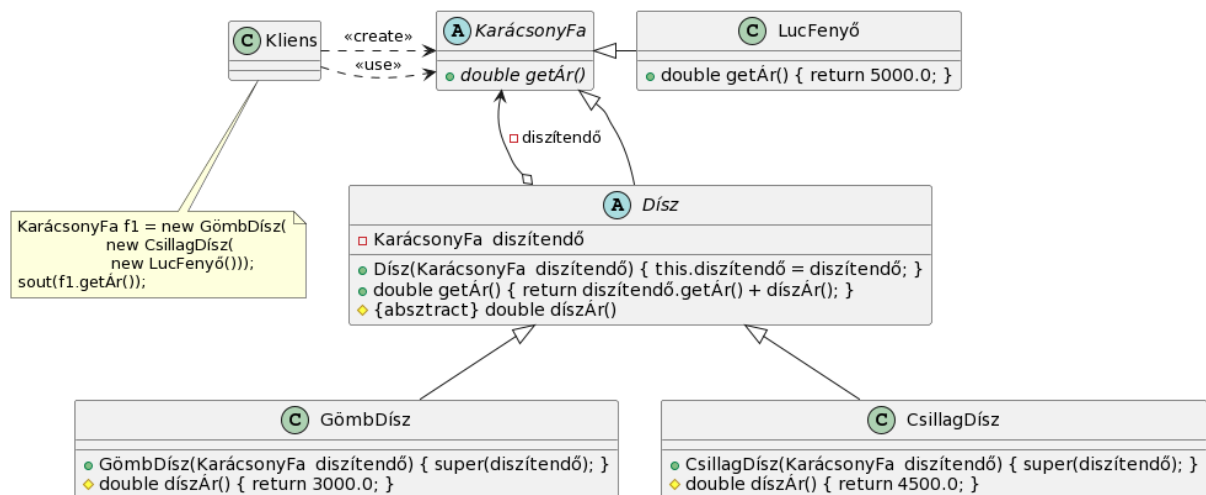
A két fél szakmai vitájából születik majd egy jó terv, mivel mindkét fél rákényszerül, hogy átgondolja a projektet, hogy legyenek szakmai érvei. És ez a tervezés lényege! Még a megvalósítás előtt gondoljuk át százszor, mit, hogyan, mivel és miért fogunk csinálunk.

1.4. Szerkezeti tervezési minták

A szerkezeti minták közös tulajdonsága, hogyan kapcsoljuk össze az osztályainkat a céljainknak megfelelően. Nagyon gyakran az IS-A és a HAS-A kapcsolatok olyan kombinációját láthatjuk, amik nem tiszta DIP megoldások, hiszen nem annyira a szétválasztásról szólnak ezek a minták, hanem az építkezésről. Így ebben a fejezetben csak 4 mintát találunk.

1.4.1. Díszítő és Összetétel

A Díszítő (angolul: Decorator) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

class Kliens {}

abstract class KarácsonyFa {

+{abstract} double get Ár()

}

class LucFenyő {

+double get Ár() { return 5000.0; }

}

abstract class Dísz {

-KarácsonyFa díszítendő

+Díz(KarácsonyFa díszítendő) { this.díszítendő = díszítendő; }

+double get Ár() { return díszítendő.get Ár() + dísz Ár(); }

#{absztract} double dísz Ár()

}

class GömbDísz {

+GömbDísz(KarácsonyFa díszítendő) { super(díszítendő); }

#double dísz Ár() { return 3000.0; }

}

class CsillagDísz {

+CsillagDísz(KarácsonyFa díszítendő) { super(díszítendő); }

#double dísz Ár() { return 4500.0; }

```
}
```

```
LucFenyő -left-|> KarácsonyFa
```

```
Dísz -up-|> KarácsonyFa
```

```
GömbDísz -up-|> Dísz
```

```
CsillagDísz -up-|> Dísz
```

```
Dísz o-up-> KarácsonyFa : -díszítendő
```

```
Kliens .right.> KarácsonyFa : <<create>>
```

```
Kliens .right.> KarácsonyFa : <<use>>
```

```
note bottom of Kliens
```

```
KarácsonyFa f1 = new GömbDísz(
```

```
    new CsillagDísz(
```

```
        new LucFenyő());
```

```
sout(f1.getÁr());
```

```
end note
```

```
@enduml
```

Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A KarácsonyFa absztrakt osztály ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Ez egy nehéz kérdés, mert több HAS-A kapcsolat is mutat a KarácsonyFa osztályra. A „-díszítendő” kapcsolat oldaláról nézve ez nem DIP, hiszen a KarácsonyFa és a DIP közt IS-A és HAS-A kapcsolat is van. Ugyanakkor a Kliens oldaláról nézve ez DIP, hiszen a KarácsonyFa szétválasztja Klient és a KarácsonyFa adatszerkezetet. Van ennek értelme? Hiszen a Kliens mindig elvállalja a program tervezés alatt lévő részétől. Igenis van értelme, mert ha a Kliens oldalt nem vizsgáljuk, akkor nem vesszük észre, hogy a Díszítő tervezési minta használata maga után vonja az úgynevezett teleszkópikus konstruktor használatát. Tehát ez egy DIP!

Nézzük meg egy kicsit részletesebben is, mi is ez a teleszkópikus konstruktor? Erről van szó:

```
KarácsonyFa f1 = new GömbDísz( new CsillagDísz( new LucFenyő()));
```

Ha tovább akarjuk bővíteni a karácsonyfánkat, akkor egyre hosszabb lesz ez a konstruktor hívás, mint amikor egy teleszkópot kinyújtunk. Erről nagyon könnyen felismerhető ez a tervezési minta.

A fenti példa csak az egyik változata a Díszítő tervezési mintának. Ez az a változat, amikor a díszítés nem ad új szolgáltatásokat a létrejövő entitáshoz. A másik változat a díszítéssel nem csak az adatszerkezetet bővíti, hanem új szolgáltatásokkal is a létrejövő entitást. Erre példát például a java.io csomagban találunk. A FileReader osztálynak nincs readLine() metódusa, de ha egy FileReader-et BufferedReader-be csomagoljuk, akkor annak már van:

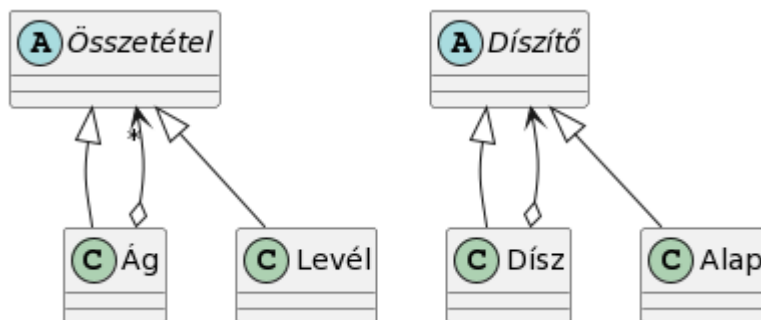

```
BufferedReader br = new BufferedReader(new FileReader("a.txt"));
```

```
String elsőSor = br.readLine();
```

Persze a fenti kódrészletet try catch blokkba kellene rakni, hiszen kiválthat kivételt, de a lényeg jól látszik: A legkülső díszítés típusát kell használni a referencia típusaként! Összefoglalva:

- első változat: Ős név = new Dísz1(new Dísz2(.... new DíszN (new Alap())...));
- második változat: Dísz1 név = new Dísz1(new Dísz2(.... new DíszN (new Alap())...));

Ráadásul itt kell tárgyalnunk a Összetétel (angolul: Composite) tervezési mintát is, hiszen a két minta közt csak annyi a különbség, hogy a visszamutató HAS-A kapcsolat egy-sok kapcsolat, ahogy a lenti UML ábra is mutatja:

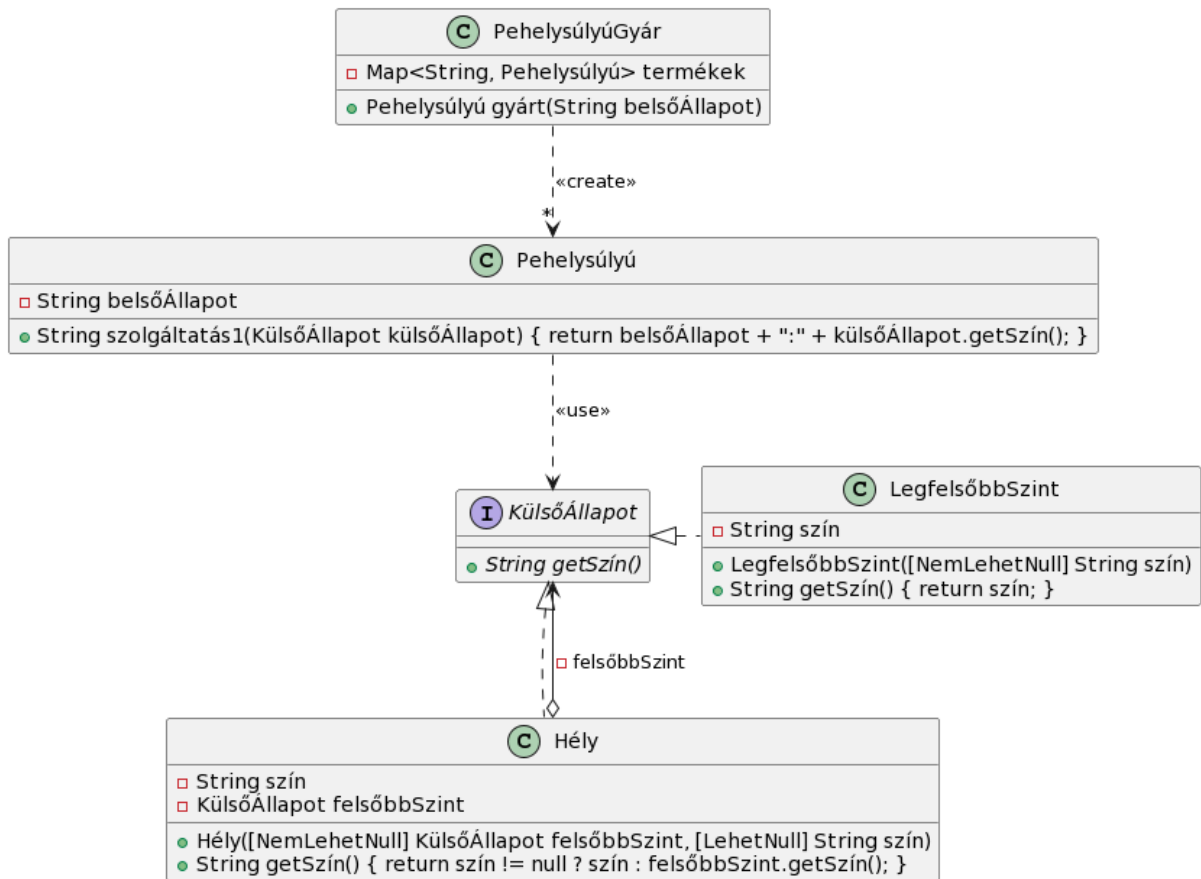


PlantUML szkriptje:

```
@startuml
abstract class Díszítő {}
class Dísz{}
class Alap{}
Dísz -up-|> Díszítő
Dísz o-up-> Díszítő
Alap -up-|> Díszítő
abstract class Összetétel {}
class Ág {}
class Levél {}
Ág -up-|> Összetétel
Ág o-up-> "*" Összetétel
Levél -up-|> Összetétel
@enduml
```

1.4.2. [Pehelysúlyú](#)

Az Pehelysúlyú (angolul: Flyweight) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

class Pehelysúlyú {

-String belsőÁllapot

+String szolgáltatás1(KülsőÁllapot külsőÁllapot) { return belsőÁllapot + ":" + külsőÁllapot.getSzín(); }
}

class PehelysúlyúGyár {

-Map<String, Pehelysúlyú> termékek

+Pehelysúlyú gyárt(String belsőÁllapot)

}

interface KülsőÁllapot {

+{abstract} String getSzín()

}

class Hély {

-String szín

-KülsőÁllapot felsőbbSzint

```
+Hély([NemLehetNull] KülsőÁllapot felsőbbSzing, [LehetNull] String szín)
+String getSzing() { return szín != null ? szín : felsőbbSzing.getSzing(); }
}
```

```
class LegfelsőbbSzing {
-String szín
+LegfelsőbbSzing([NemLehetNull] String szín)
+String getSzing() { return szín; }
}
```

PehelysúlyúGyár .down.> "*" Pehelysúlyú : <<create>>

Pehelysúlyú .down.> KülsőÁllapot : <<use>>

Hély .up.|> KülsőÁllapot

LegfelsőbbSzing .left.|> KülsőÁllapot

Hély o--> KülsőÁllapot : -felsőbbSzing

@enduml

Ez a tervezési minta egy összetett minta, a Többke (angolul: Multiton) és a Felelősséglánc (angolul: Chain-of-Responsibility) kombinációja. A tervezési minta lényege, hogy Pehelysúlyú belső állapota önmagában nem teljesen írja le a kívánt állapotot, ahhoz kell az úgynevezett külső állapot is. Itt két lehetőség van, vagy a Pehelysúlyú könnyen elérhetővé teszi a belső állapotát, hogy azt bárki összekombinálhassa a külső állapottal. Vagy, a Pehelysúlyú a saját metódusaiban fogadja a külső állapotot, hogy a belső és a külső állapottal együttesen tudja szolgáltatni a megfelelő szolgáltatásokat.

Ez utóbbi változatot mutatja be a fenti UML ábra. A gyár azért felelős, hogy ne jöjjön létre kétszer ugyanaz az állapot feleslegesen. A felelősséglánc pedig a környezet tárolásáért felelős.

Ezt nagyon nehéz megérteni egy példa nélkül. A klasszikus példa az objektumorientált szövegszerkesztő. Egy objektumorientált szövegszerkesztőben kívánatos lenne, hogy minden egyes karakter külön objektum legyen, hiszen minden karakternek lehet külön betűmérete, színe, betűtípusa. Ugyanakkor a gyakorlatban nagyon-nagyon ritka, hogy tényleg minden karakter más színű legyen. E helyett egy szűkebb, vagy tágabb környezetben a karakterek színe ugyanaz. Ezért elegendő a Karakter osztályban, azaz a pehelysúlyú osztályban, csak a karakter ASCII kódját tárolni, a karakter színét pedig a környezetben. A karakter környezete a szó, a szó környezete a bekezdés, stb..., így kifelé haladva valahol lesz szín tulajdonság. Ezt látjuk a fenti ábrán modellezve.

Tegyük fel a szokásos kérdéseket:

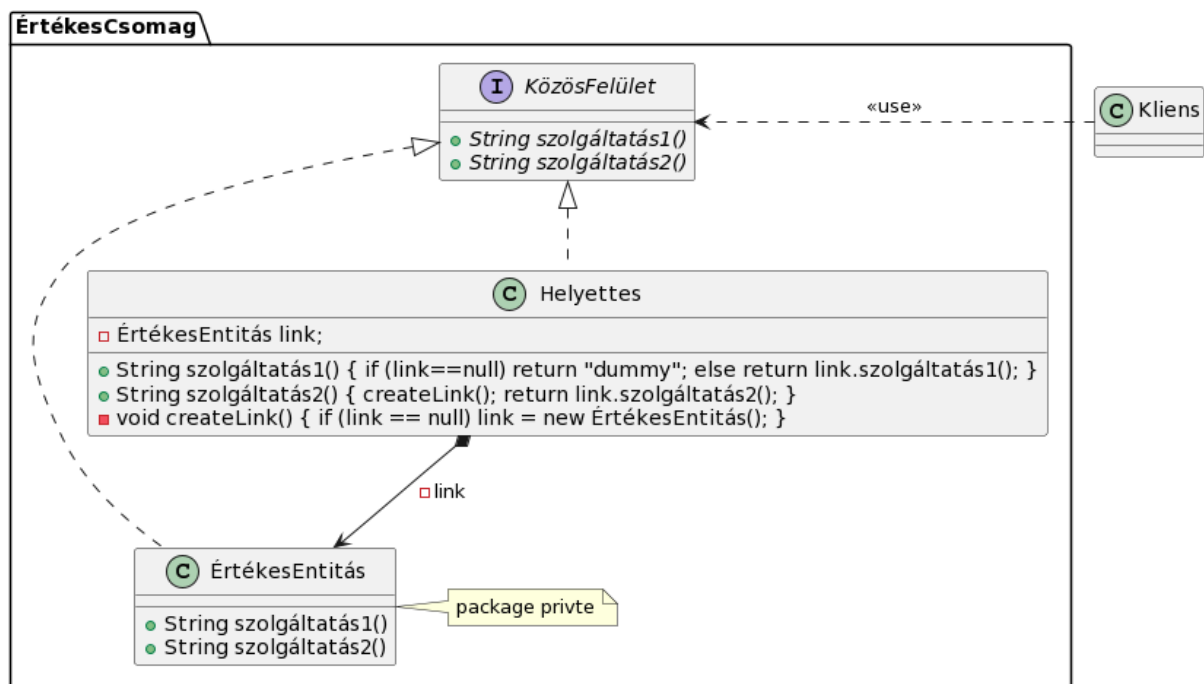
- Minden nyíl absztrakcióra mutat? Nem! A gyárból a Pehelysúlyú osztályra mutató nyíl nem ilyen, mert a Pehelysúlyú osztály nem absztrakt. Ez még nem gond, ettől lehet DIP az ábrán. Ha Többkét alkalmaztunk volna gyár helyett, akkor nem is lenne ott ez a nyíl, de a hagyományoknak inkább megfelelő ábra készítése volt a cél.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A KülsőÁllapot interfész ilyen.

- Ez az absztrakció szétválasztja a két oldalt? Igen, a KülsőÁllapot interfész segítségével választjuk el a Pehelysúlyú osztályt, ami a belső állapotot tartalmazza, a küldő állapotától. Tehát ez egy DIP!

A KülsőÁllapot, illetve az azt megvalósító osztályok zárják egységbe azt a logikát, hogy hogy keressük a színt, ami a külső állapot része. Ha a Pehelysúlyú közvetlen környezetében nincs szín megadva, akkor a felsőbb szint referencia segítségével eggyel tovább lépünk a láncon, hátha ott már van szín. Ha nincs, akkor még kijebb és kijebb lépünk, egészen a legfelsőbb szintig, ahol már kötelező, hogy legyen szín. Ehhez a [NemLehetNull] megszorítást használtuk.

1.4.3. Helyettes

Az Helyettes (angolul: Proxy) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
```

```
package ÉrtékesCsomag {
```

```
interface KözösFelület {
```

```
+{abstract} String szolgáltatás1()
```

```
+{abstract} String szolgáltatás2()
```

```
}
```

```
class Helyettes {
```

```
-ÉrtékesEntitás link;
```

```
+String szolgáltatás1() { if (link==null) return "dummy"; else return link.szolgáltatás1(); }
```

```
+String szolgáltatás2() { createLink(); return link.szolgáltatás2(); }
```

```

-void createLink() { if (link == null) link = new ÉrtékesEntitás(); }
}

class ÉrtékesEntitás {
+String szolgáltatás1()
+String szolgáltatás2()
}

Helyettes .up. |> KözösFelület
ÉrtékesEntitás .up. |> KözösFelület

Helyettes *--> ÉrtékesEntitás : -link

note right of ÉrtékesEntitás
    package privte
end note
}

class Kliens {}

Kliens .right.> KözösFelület : <<use>>

@enduml

```

Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A KözösFelület interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a KözösFelület interfész segítségével választjuk el a klienst és az értékes entitást, amihez a kliens csak a helyettesen (angolul: proxy) keresztül férhet hozzá. Tehát ez egy DIP!

Úgy érezzük el, hogy a kliens ne példányosíthassa közvetlenül az értékes entitást, hogy külön csomagba helyezzük, és a csomagon belül csomag privát láthatósági szintet adunk neki. Így a csomagon belül lévő helyettes látja, példányosíthatja, de a kliens nem. A kliens csak a helyettest tudja példányosítani, hiszen az publikus osztály.

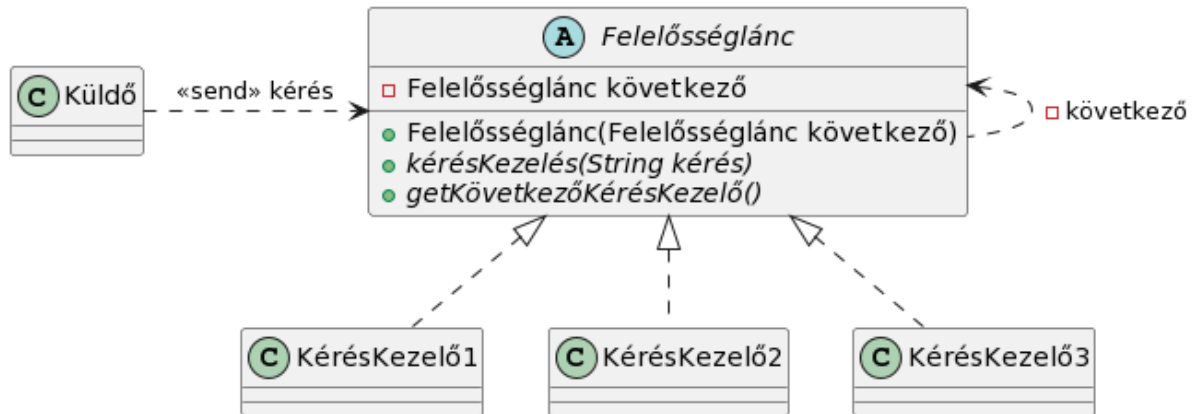
Habár PlantUML bő lehetőséget kínál a mezők és a metódusok láthatósági szintjének beállítására, de az osztályok láthatósági szintje nem állítható, ezért használtunk megjegyzést ennek jelzésére.

1.5. Viselkedési tervezési minták

A viselkedési tervezési minták arra koncentrálnak, hogy hogyan zárjuk egységbe egy entitás viselkedését, hogyan bővítjük a viselkedést egy-egy új alosztály bevezetésével. Mint látni fogjuk majd mindegyik tervezési minta használja a DIP alapelvet, habár egy-egy konvenciók kényszer kétségessé teszi, hogy tényleg DIP alkalmazásával állunk szemben, vagy sem. Végül a Látogató tervezési minta kivételével mindegyik mintában sikerült tetten érünk a DIP-et, mint ahogy látni fogjuk.

1.5.1. Felelősséglánc

A Felelősséglánc (angolul: Chain-of-Responsibility) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
class Küldő {}
abstract class Felelősséglánc {
-Felelősséglánc következő
+Felelősséglánc(Felelősséglánc következő)
+{abstract} kérésKezelés(String kérés)
+{abstract} getKövetkezőKérésKezelő()
}
class KérésKezelő1 {}
class KérésKezelő2 {}
class KérésKezelő3 {}
Küldő .right.> Felelősséglánc : <<send>> kérés
Felelősséglánc .left.> Felelősséglánc : -következő
KérésKezelő1 .up.|> Felelősséglánc
KérésKezelő2 .up.|> Felelősséglánc
KérésKezelő3 .up.|> Felelősséglánc
@enduml
```

Már a Pehelysúlyú tervezési minta esetén láttunk egy felelősségláncot. Láttuk, hogy a láncan addig megyünk, amíg a szín mező valamelyik láncszemben már nem null. Ettől kicsit bonyolultabb felelősségláncok is készíthetők a fenti osztálydiagram mintájára.

Az továbbra is nagyon fontos, hogy legyen egy lánc, amit következő referencia segítségével tudunk bejárni. Nagyon érdekes, hogy ez a referencia egy visszamutató nyíl a Felelősséglánc absztrakt

osztályra, de ha belegondolunk, akkor ezt pont így kell csinálni. Sajnos e miatt a mező miatt a Felelősséglánc nem lehet interfész, de ezzel együtt tudunk élni. Ami újdonság, az az, hogy az egyes láncszemek lehetnek külön-külön osztályban, így olyan logikát írhatunk le, amelyet csak szeretnénk. Ez a logika dönti el, hogy a kliens (a fenti ábrán kliens helyett küldő van) által küldött kérést képesek vagyunk-e feldolgozni vagy sem. Ha igen, megadjuk a választ, ha nem, továbbpasszoljuk a láncon a következő láncszemnek. Harmadik lehetőség, hogy a kérésnek csak egy részét válaszoljuk meg, és mivel még nincs meg a teljes válasz, ezért továbbküldjük.

Például a kérés lehet egy szövegrészlet fordítása magyarról angolra, de a szövegben van egy vers és néhány latin kifejezés. Az első láncszem lefordít mindent, kivéve a verset és a latin kifejezéseket. A második láncszem kikeresi a vers fordítását és beilleszti. A harmadik láncszem az idegen kifejezésekről dönti el, hogy maradjanak úgy, ahogy vannak, vagy lehet fordítani. A negyedik láncszem a cirill betűs részeket nézi, de ilyen nincs a szövegben. Végül összeáll a fordítás, a választ megkapja a kliens.

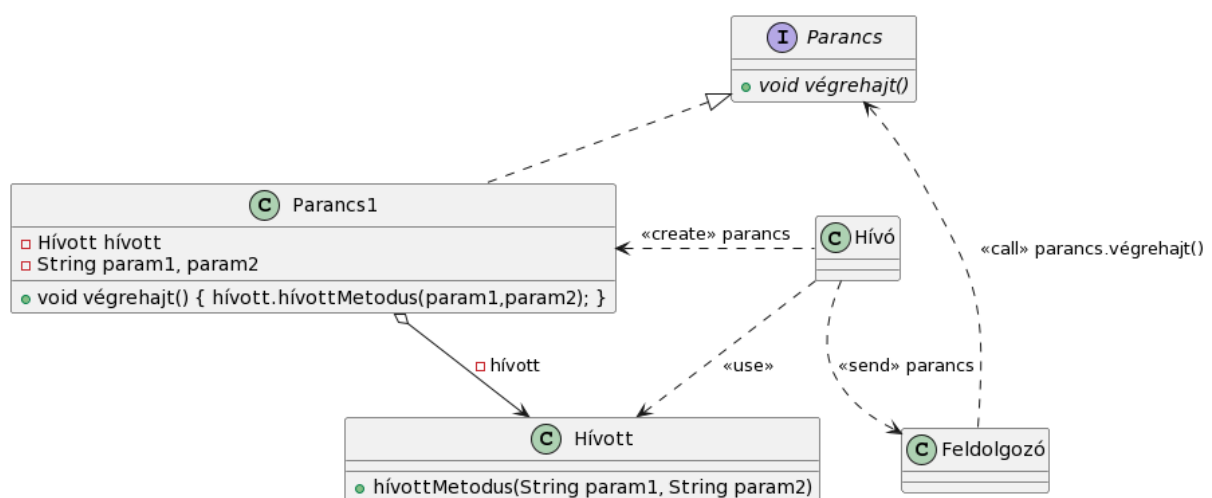
Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A Felelősséglánc absztrakt osztály ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Felelősséglánc absztrakt osztállyal választjuk szét a kérést összeállító és a választ váró részt a kérést megválaszolni képes felelősséglánctól. A felelősséglánc minden láncszemét külön osztályba zárjuk, amiről a másik oldal nem tud semmit. Tehát ez egy DIP!

Itt még lehet megfogalmazni kritikát: Miért zárjunk minden láncszemet külön osztályba? Ki hozza létre a felelősséglánct? Az első kérdés könnyen megválaszolható. Ez csak egy lehetőség. Ha nem túl bonyolult a logika, akkor persze nem kell minden láncszemet külön osztályba zárni. A második kérdés már nehezebb. Nyilván kell egy kliens vagy gyártó kód, ami létrehozza a felelősséglánct és eljuttatja az első láncszem referenciáját a kérdés küldőjének. Ez nincs rajta a fenti ábrán.

1.5.2. Parancs

A Parancs (angolul: Command) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

```

interface Parancs {
    +{abstract} void végrehajt()
}

class Parancs1 {
    -Hívott hívott
    -String param1, param2
    +void végrehajt() { hívott.hívottMetodus(param1,param2); }
}

class Hívó {}

class Feldolgozó {}

class Hívott {
    +hívottMetodus(String param1, String param2)
}

Hívó .left.> Parancs1 : <<create>> parancs

Hívó ..> Hívott : <<use>>

Hívó ..> Feldolgozó : <<send>> parancs

Feldolgozó .right.> Parancs : <<call>> parancs.végrehajt()

Parancs1 .up.|> Parancs

Parancs1 o--> Hívott : - hívott

@enduml

```

A Parancs tervezési mintát leggyakrabban egy szövegszerkesztő példáján keresztül mutatják be. A beilleszt gomb implementációjába nem érdemes beírni a beillesztés logikáját, mert a Ctrl+V gomb megnyomására is ugyanaz kell, hogy történjen. Ezért ezt a logikát érdemes kiemelni, és a beilleszt gombnak csak átadni egy parancs objektumot, aminek csak egy végrehajt (angolul: execute) metódusa van. Ugyanezt a parancsot odaadni a Ctrl+V gombnak is. Ha a felhasználó megnyomja a beilleszt gombot, vagy a Ctrl+V gombot, akkor ugyanaz történik.

Ha a fenti példát a fenti UML osztály diagramra akarjuk vetíteni, akkor a beilleszt gomb és a Ctrl+V gomb felelnek meg a Feldolgozó egy-egy példányának, mindketten ugyanazt a Parancs1 objektumot kapják meg, ahol a Hívott valósítja meg a szükséges logikát.

Könnyű észrevenni, hogy a Hívó és a Hívott elválik egymástól, a kettő közé bekerül a Feldolgozó. Ezek máskor annyira szorosan összetartoznak. Miért érdemes elválasztani egymástól a Hívót és Hívottat? Azért, mert így magát a hívást becsomagolhatjuk egy Parancs objektumba, amely tartalmazza, hogy ki a Hívott, annak melyik metódusát és milyen paraméterekkel kell meghívni, de a hívás nem történik meg. Majd csak akkor, ha a végrehajt metódust meghívja a Feldolgozó. A Feldolgozó a parancs objektumot beteheti egy várakozási sorba, elmentheti, átküldheti egy távoli szerverre, azaz minden csinálhat vele, amit egyébként egy üzenettel (angolul: message) megtehetünk. A Parancs tervezési

minta segít nekünk, hogy egy egyszerű metódus hívásból egy üzenetküldés (angolul: message passing) legyen.

Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Nem! A Hívóból induló több nyíl is konkrét osztályra mutat. Ettől még lehet DIP a tervben.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A Parancs interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Parancs interfész szétválasztja a Feldolgozó és a Hívott osztályokat. Egyik se tud a másikról semmit, csak azt, hogy ha a Feldolgozó meghívja a parancs objektum végrehajt metódusát, akkor valami hasznos fog történni. Tehát ez egy DIP!

Itt még lehet megfogalmazni néhány kritikát: A fentiekben azt írtuk, hogy a Parancs interfész a Feldolgozót és a Hívott osztályt választja szét. Egy kicsivel feljebb pedig azt, hogy ez a tervezési minta arra jó, hogy a Hívó és Hívott osztályt válasszuk szét. Mindkettő igaz? Csak az egyik? Egyik sem? Mindkettő igaz, a Hívó és a Hívott azzal választjuk el, hogy köztük van a Feldolgozó. A Feldolgozót pedig a Parancs interfész választja el a Hívott osztálytól.

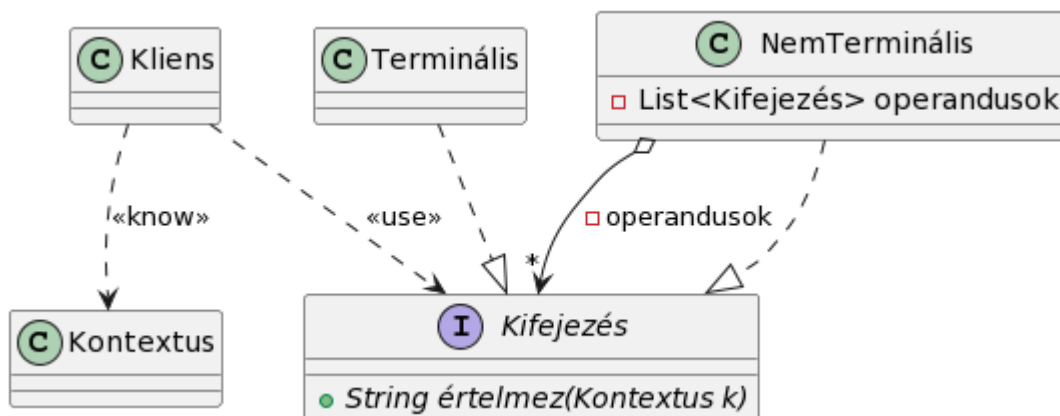
A másik kritika, hogy a Hívó nem is válik el a Hívott osztálytól, hisz van a kettő közt egy asszociáció. Ez teljesen igaz, a Hívónak ismernie kell a Hívott egy példányát, hogy elkészítse a parancs objektumot, amit beinjektál egy parancs példányba. Olyan értelemben válnak el egymástól, hogy ez a két időpont különül el egymástól:

- az az időpillanat, amikor minden részlet összeáll, ami kell a híváshoz; illetve
- az az időpillanat, amikor megtörténik a hívás.

A következő kritika, hogy aki már ült Programozási technológiák előadáson, az a Parancs tervezési mintával sokat hallott arról, hogy a hívó és a hívott közt ott van egy postás / elosztó (angolul: broker), és ez a postás lehet okos, buta, kíváncsi és ennek az egésznek köze van az üzenetközvetítés (angolul: message broker) témaköréhez. A kérdés, miért nem került elő ez a téma ebben a fejezetben? Arra könnyű rájönni, hogy a feldolgozó felel meg a postásnak. Mivel a Bróker tervezési minta egy külön architektúráis tervezési minta, ezért döntöttünk a fenti ábrán a Feldolgozó szó használata mellett.

1.5.3. Értelmező

Az Értelmező (angolul: Interpreter) tervezési minta UML ábrája:



PlantUML szkriptje:

```

@startuml
class Kliens {}
class Kontextus {}
interface Kifejezés {
+{abstract} String értelmez(Kontextus k)
}
class Terminális {}
class NemTerminális {
-List<Kifejezés> operandusok
}
Kliens ..> Kontextus : <<know>>
Kliens ..> Kifejezés : <<use>>
NemTerminális o--> "*" Kifejezés : -operandusok
NemTerminális ..|> Kifejezés
Terminális ..|> Kifejezés
@enduml

```

Az Értelmező tervezési mintát domén specifikus nyelvek (angolul: Domain Specific Language – DSL) létrehozására és felhasználására alkalmas. Mivel ez a terület inkább a Fordító programok tárgyhoz tartozik, ezért ezt a mintát itt nem tárgyaljuk részletesen.

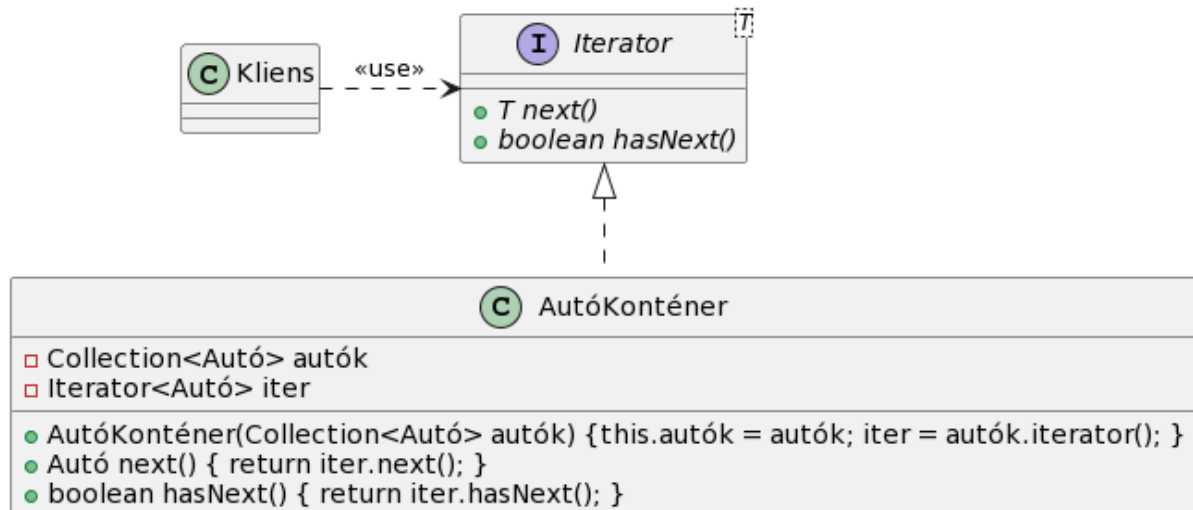
Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Nem! Kontextus osztályba vezető nyíl nem ilyen. Ráadásul a Kifejezés interfészből is kellene egy asszociáció nyilat indítani a Kontextus osztályba.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A Kifejezés interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Kifejezés interfész szétválasztja a Kliens részt a DSL-től. A DSL nem tud semmit a Kliensről. A Kliensnek viszont tudnia kell, hogyan épülnek fel a DSL mondata, de azt nem, hogy ezek a mondatok mit jelentenek. Tehát a Kliensnek ismernie kell a DSL szintaxisát, de nem kell ismernie a szemantikáját, ugyanakkor az értelmezett mondatok eredményét felhasználja a kliens. Tehát ez egy DIP!

A DSL-ek az MDA (angolul: Model Driven Architecture) egyre fontos területe. Azt szoktuk mondani, hogy egy DSL a saját területén nagyon hatékony, minden más területen nagyon rossz. Ezzel szemben egy általános célú programozási nyelv (angolul: General-Purpose Programming Language) mindenre célra jó egy kicsit, de semmire se jó igazán.

1.5.4. Iterátor

Az Iterátor (angolul: Iterator) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

class Kliens {}

interface Iterator<T> {

+{abstract} T next()

+{abstract} boolean hasNext()

}

class AutóKonténer {

-Collection<Autó> autók

-Iterator<Autó> iter

+AutóKonténer(Collection<Autó> autók) {this.autók = autók; iter = autók.iterator(); }

+Autó next() { return iter.next(); }

+boolean hasNext() { return iter.hasNext(); }

}

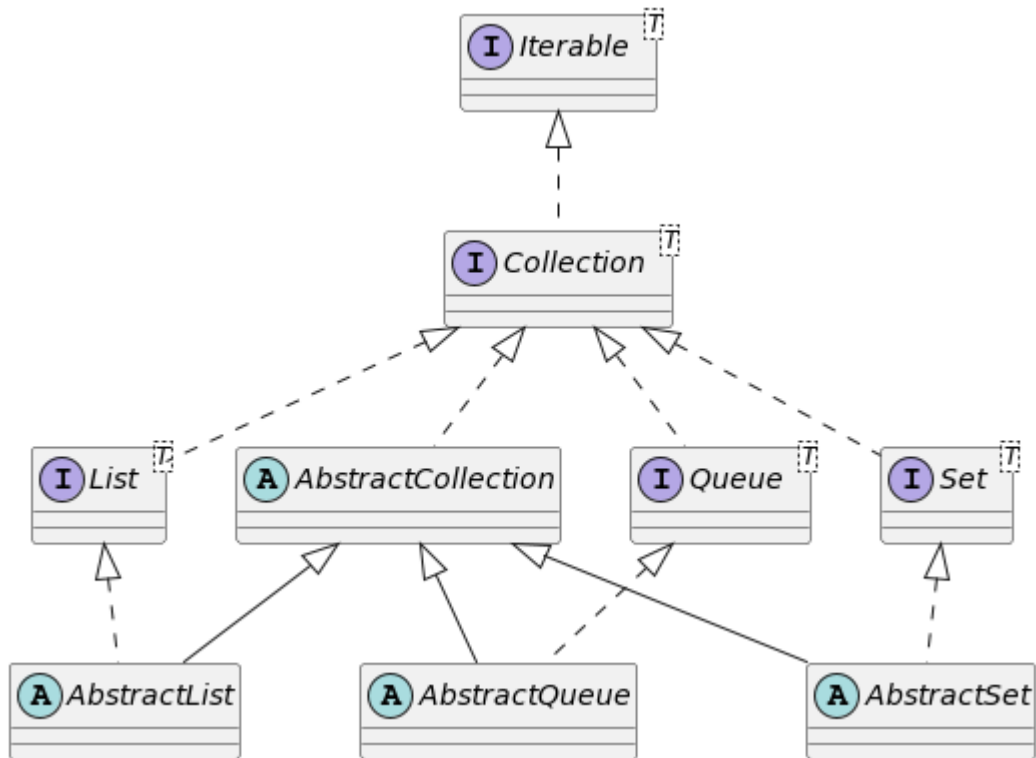
Kliens .right.> Iterator : <<use>>

AutóKonténer .up.|> Iterator

@enduml

Ez az ábra nem teljesen elvont, ami az eddigi ábráknál majd mindig cél volt: AutóKonténer helyett használhattunk volna valami elvontabbat is. Ugyanakkor a példánál törekedtünk úgy bemutatni az Iterátor tervezési mintát, ahogy az a Java nyelvnek része. A Java programozási nyelvben van Iterator<T> interfész, és tényleg két metódusa van: next és hasNext (illetve van még másik kettő is, de azoknak van default implementációjuk). Továbbá a Collection interfésznek tényleg van egy iterator metódusa, aminek a visszatérési típusa Iterator<T>, úgy ahogy az ábrán használjuk. Illetve már a Collection egyik ősének, az Iterable interfésznek megvan ez a metódusa, de azt kevésbé ismerik a

programozók. Nézzük meg ezt a hierarchiát, a teljesség igénye nélkül, hiszen ezzel léptem nyomon találkozik minden programozó:



PlantUML szkriptje:

```

@startuml
interface Iterable<T> {}
interface Collection<T> {}
interface List<T> {}
interface Queue<T> {}
interface Set<T> {}
abstract class AbstractCollection {}
abstract class AbstractList {}
abstract class AbstractQueue {}
abstract class AbstractSet {}
Iterable <|.. Collection
Collection <|.. List
Collection <|.. Queue
Collection <|.. Set
Collection <|.. AbstractCollection
  
```

```
AbstractCollection <|-- AbstractList
AbstractCollection <|-- AbstractQueue
AbstractCollection <|-- AbstractSet
Set <|.. AbstractSet
List <|.. AbstractList
Queue <|.. AbstractQueue
@enduml
```

Tehát ez a tervezési minta széles körben használatos a modern OOP nyelvekben, a modern Collection osztályok mindegyike implementálja. Ennek ellenére a foreach használata elterjedtebb, mint a while hasNext használata, habár mindkettőt az Iterable interfész teszi lehetővé.

Ha saját konténer osztályt készítünk, és szeretnénk, hogy az könnyen feldolgozható legyen mások számára is, akkor a legegyszerűbb megvalósítani az Iterator<T> interfészt. Ez egy nagyon vékony interfész, csak két metódus van benne, amiket könnyű megírni. Ugyanakkor van egy mögöttes szerződés, ugyanúgy, ahogy a foreach-nél is. Amíg tart a while hasNext alapú feldolgozás, addig a Collection-ből nem szabad se törölni, se hozzáadni, de még módosítani sem.

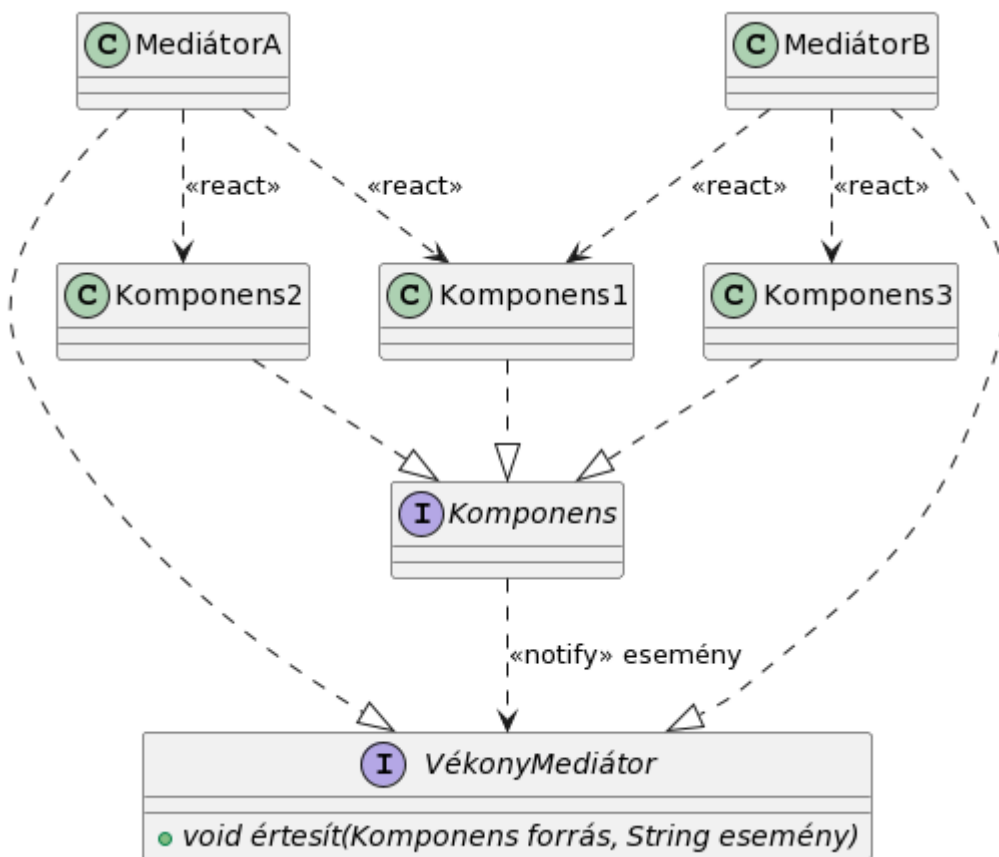
Térjünk vissza a tervezési minta osztály diagramjához. Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! Az Iterator<T> interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, az Iterator<T> interfész szétválasztja a kliens részt a konténer résztől. A konténer nem tud semmit a kliensről, a kliens is csak annyit tud a konténerről, hogy a konténerben tárolt elemeket egy while ciklussal feldolgozhatja. Tehát ez egy DIP!

Habár ez a tervezési minta mélyen beépült a modern programozási nyelvekbe, mégis a kevésbé hasznos minták közé szokták sorolni a szakirodalomban, lásd például a „Impact of design patterns on software quality: a systematic literature review” című cikket.

1.5.5. [Közvetítő](#)

A Közvetítő (angolul: Mediator) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

interface VékonyMediátor {

+{abstract} void értesít(Komponens forrás, String esemény)

}

interface Komponens {}

class MediátorA {}

class MediátorB {}

class Komponens1 {}

class Komponens2 {}

class Komponens3 {}

MediátorA ..|> VékonyMediátor

MediátorB ..|> VékonyMediátor

Komponens1 ..|> Komponens

Komponens2 ..|> Komponens

Komponens3 ..|> Komponens

Komponens ..> VékonyMediátor : <<notify>> esemény

MediátorA ..> Komponens1 : <<react>>

MediátorA ..> Komponens2 : <<react>>

MediátorB ..> Komponens1 : <<react>>

MediátorB ..> Komponens3 : <<react>>

@enduml

Vége, sikerült szívecskét rajzolni UML-ben! Na, de komolyra fordítva a szót, a Mediátor tervezési minta a Megfigyelő tervezési mintának az a változata, amikor az esemény hatására nem minden megfigyelő, ez esetben Komponens, kap értesítést, hanem valamilyen logika szerint csak néhány. Érdekes, hogy az események is a Komponensekben keletkeznek. Az események válasz tevékenységeket, reakciókat (angolul: reaction, igealak: react) váltanak ki.

Nézzük mindezt egy kicsit részletesebben. Először is szögezzük le, hogy most a vékony mediátort vizsgáljuk, ahol a mediátor interfésznek csak egy metódusa van, az értesít. Mint látni fogjuk, van a mediátornak egy vastag változata is.

Tegyük fel, hogy van egy okos otthonunk, okos órával, okos kávéfőzőfel, okos redőnnel, okos világítással, okos hűtőszekrénnel, okos rádióval, stb.... Ezek a komponensek. A komponensek eseményeket generálhatnak. Pl. az okos óra reggel 6-kor küldhet egy eseményt a mediátornak, amiben benne van a forrás, azaz az okos óra, és az esemény leírása: „ideje felkelni”. A mediátor átnézi, hogy van-e ehhez valamilyen logika rendelve. A vékony mediátornál ehhez kapcsolódó metódusok nincsenek. Majd a vastag mediátornál lesz ehhez támogatás.

Tegyük fel, hogy a mediátor azt találja, hogy ilyenkor utasítania kell az okos kávéfőzőt, hogy kezdjen kávé főzni. Ez egy reakció. Illetve utasítania kell az okos világítást, hogy legyen világosság. Ez egy másik reakció. Mint látható, a reakciók, azok komponenseknek adott utasítások. Tegyük fel, hogy ha kész a kávé, akkor az okos kávéfőző küld egy eseményt, hogy „kész a kávé”. A mediátor megnézi, mit kell tennie. Mondjuk, bekapcsol parancsot kell küldenie az okos rádiónak, hogy a ház tulajdonosa felébredjen.

Nyilván a Mediátor minta erősen kapcsolódik a Parancs tervezési mintához is, mint ahogy látni fogjuk ezt a vastag változatban, hiszen honnan tudná szegény mediátor, hogyan kell bekapcsolni egy okos rádiót, vagy utasítani az okos világítást.

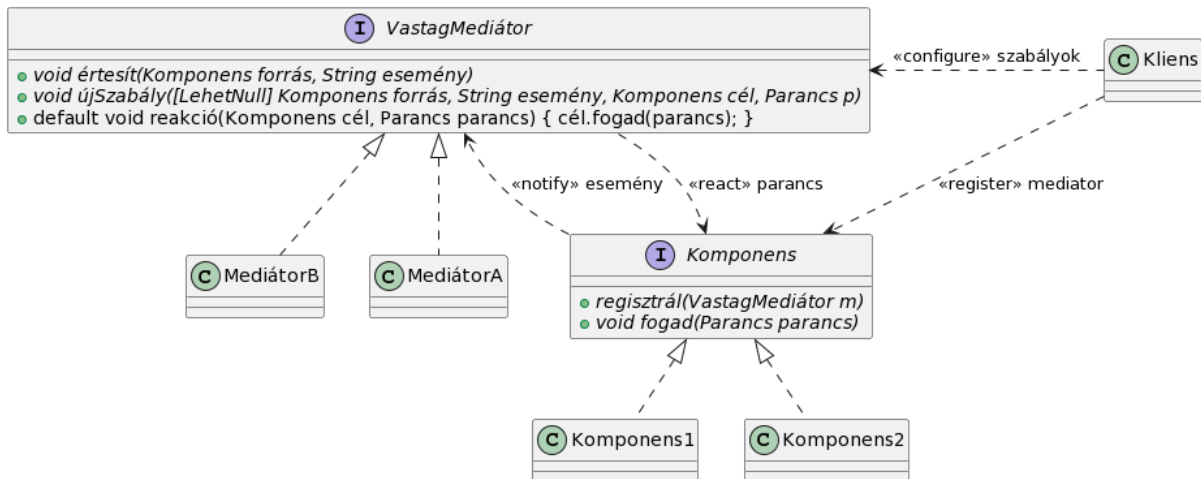
Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! A VékonyMediátor interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Nem, mert a konkrét mediátoroknak és a konkrét komponenseknek is tudniuk kell egymásról. A komponensnek tudnia kell, hogy kinek küldje az eseményeket, a mediátornak pedig tudnia kell, hogy melyik komponensnek küldjön reakciót. Ez nem DIP! Legalább annyi, hogy a komponensek a közös felületen keresztül hívják az értesít metódust. Pedig gyanús, hogy a mediátor arra jó elválassza egymástól a komponenseket, hogy azoknak ne kelljen egymással beszélgetniük. Ez meg is valósul, hiszen bármelyik komponensből indulva nem lehet a nyilak mentén egy másik komponensbe eljutni, de a

konkrét mediátorok és a konkrét komponensek elég sokat kell, hogy tudjanak egymásról. Ennek oka az elválasztó interfész vékonysága. Nézzük meg a vastag változatot.

Itt még lehet megfogalmazni néhány kritikát:

A vastag változat UML ábrája:



PlantUML szkriptje:

@startuml

interface VastagMediátor {

+{abstract} void értesít(Komponens forrás, String esemény)

+{abstract} void újSzabály([LehetNull] Komponens forrás, String esemény, Komponens cél, Parancs p)

+default void reakció(Komponens cél, Parancs parancs) { cél.fogad(parancs); }

}

interface Komponens {

+{abstract} regisztrál(VastagMediátor m)

+{abstract} void fogad(Parancs parancs)

}

class MediátorA {}

class MediátorB {}

class Komponens1 {}

class Komponens2 {}

MediátorA .up. |> VastagMediátor

MediátorB .up. |> VastagMediátor

Komponens1 .up. |> Komponens

Komponens2 .up. |> Komponens

Komponens ..> VastagMediátor : <<notify>> esemény

VastagMediátor ..> Komponens : <<react>> parancs

```
class Kliens {}
```

Kliens .left.> VastagMediátor : <<configure>> szabályok

Kliens ..> Komponens : <<register>> mediator

@enduml

A VastagMediátor egyrésztől kényelmetlenebb, mert nem 1 hanem 3 metódusa van, igaz az egyiknek van default implementációja. Másrésztől sokkal kényelmesebb, mert sokkal világosabb, hogy kell használni. A mediátorban van egy szabály rendszer, hogy milyen forrásból jövő, milyen eseményre, kit kell utasítani (cél), és mire (parancs), lásd újSzabály metódus paramétereit. Mivel gyakran mindegy, hogy mi a forrás, csak az esemény számít, ezért a forrás akár lehet null is. A forrás és a cél is komponens, ahogy ezt feljebb már tárgyaltuk. Azt, hogy mire kell utasítani a komponenst, azt egy parancs objektumként kapja meg a mediátor. Ezt a konfigurációt a kliens végzi.

Továbbá, a kliens állítja be azt is, hogy a komponensek kinek kell, hogy küldjék az eseményeket. Ezután, ha a komponensben esemény történik, akkor értesítést küld a mediátornak. A mediátor megnézi a szabályrendszert, és az ott rögzített megfelelő reakciókat adja. Egy reakció: egy komponensnek egy parancs küldés.

Térjünk vissza az okosotthonhoz. Megvettük a sok okos eszközt, de ettől ezek még nem működnek együtt. Először is minden okos eszköznek beállítjuk, hogy melyik mediátorhoz küldje az eseményeket. Ez gyakran egy tablet, vagy fali panel, de akár több is lehet, bár ezt az ábrán nem jelöltük, de nem is zártuk ki. Aztán a fali panelen jön egy sziszifuszi munka: konfigurálni kell, hogy melyik eseményre milyen reakciókat adjunk. Ha ez kész, akkor, a beállításoknak megfelelően, reggel 6-kor friss kávé illatára ébredhetünk. Viszont addig bújhatjuk az okos eszközök és a fali panel használati utasítását.

Az újSzabály metódusnak lehet egy ötödik paraméter, a fontosság:

újSzabály([LehetNull] Komponens forrás, String esemény, Komponens cél, Parancs p, int fontosság)

Ezt arra használhatjuk, hogy ha esemény hatása több reakció, akkor ezek sorrendjét beállítsuk. Esetleg valamilyen globális kritériumot adjunk meg: ha az áram ára magasabb, mint 100 forint/kWh, akkor csak az 1-es fontosságú reakciók lépjenek hatályba.

Tegyük fel a szokásos kérdéseket:

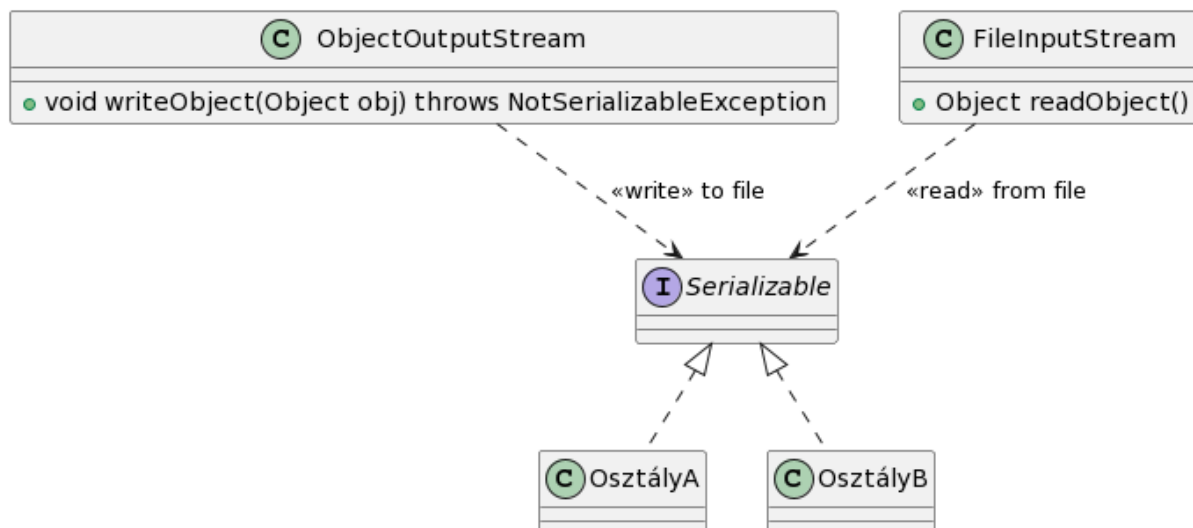
- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen! Több ilyen is van, a VastagMediátor és a Komponens interfész.
- Ezek közül az absztrakciók közül van olyan, ami szétválasztja az IS-A és a HAS-A oldalt? Igen, ez mindkét interfészre igaz. A mediátoroknak elegendő csak azt tudniuk a komponensekről, hogy azok képesek parancsot fogadni, a komponenseknek pedig csak annyit kell tudniuk, hogy a mediátorok képesek értesítéseket fogadni. Ettől többet a két oldal nem tud egymásról. Még a kliens számára van egy-egy metódus, de a két oldal szétválasztását ez nem zavarja. Mivel a két oldal szépen szétválik, ezért ez egy DIP!

Itt még lehet megfogalmazni néhány kritikát: Hogy lehet, hogy az első ábra nem DIP, a második pedig már az? Ez a kliens miatt van. Az első ábrán nincs kliens, azt kell feltételeznünk, hogy a mediátor sokat tud a komponensekről, hogy vezérelni tudja őket. A második ábrán már van kliens, akinél ott van a tudás, tudja konfigurálni a mediátort, ehhez van megfelelő metódus a VastagMediátorban. Így már a mediátor alig kell, hogy tudjon valamit a komponensekről. Ha az első ábrán lenne kliens, akkor se lenne DIP, mert a VékonyMediátorban nincs megfelelő metódus, amivel a kliens konfigurálni tudná a mediátort.

Hát senkit nem zavar a körkörös kommunikáció a VastagMediátor és a Komponens interfész közt? De igen, ez egy potenciális veszély forrás. Az óvatlan kliens definiálhat olyan szabályt, hogy ha kész a kávéfőzés, akkor adjunk ki kávéfőzés parancsot. Ez akár még tűzveszélyes is lehet! Ezért a szabályrendszer kialakításánál mindig ellenőrizni kell, hogy van-e végtelen rekurzió, és ha igen, akkor hibaüzenetet kell adnunk. Erre jól jöhet egy Értelmező tervezési minta, hiszen a szabályrendszer megadása biztosan kényelmesebb egy DSL segítségével, mint mondjuk Java programozási nyelven.

1.5.6. Pillanatkép

A Pillanatkép (angolul: Memento, vagy: Snapshot) tervezési minta UML ábrája:



PlantUML szkriptje:

```

@startuml
class ObjectOutputStream {
+void writeObject(Object obj) throws NotSerializableException
}
class FileInputStream {
+Object readObject()
}
interface Serializable {}
class OsztályA {}
  
```

```
class OsztályB {}
```

```
ObjectOutputStream ..> Serializable : <<write>> to file
```

```
FileInputStream ..> Serializable : <<read>> from file
```

```
OsztályA .up.|> Serializable
```

```
OsztályB .up.|> Serializable
```

```
@enduml
```

A Pillanatkép tervezési minta nagyon hasonló az Iterátor tervezési mintához, olyan tekintetben, hogy mélyen beépült a modern programozási nyelvekbe, mégis a szakirodalom a kevésbé hasznos minták közé sorolja. Java programozási nyelven a Serializable interfész felel meg a Pillanatkép tervezési mintának, habár ez egy teljesen üres interfész, nincs benne egy metódus se. Mondhatni, szuper vékony!

Ez kicsit ellentmondásosnak hangzik, hiszen a Pillanatkép klasszikus leírásában pár metódust előír pl. a GOF könyv előírja, hogy legyen egy setState és egy getState metódusa. Ezzel szemben a Serializable interfészben egy metódus sincs. Ez azért van, mert ahhoz, hogy serializálható, azaz byte folyamattá alakítható legyen egy objektum, ahhoz csak az kell, hogy a mezői serializálhatóak legyenek.

Ide kapcsolódik még a transient kulcsszó, ami mező deklarációban szerepelhet. A tranziens (angolul: transient), vagy más szóval, átmeneti mezők nem vesznek részt a serializáció, deserializáció folyamatában. Ha van olyan mezőnk, ami számított értéket tartalmaz, vagy ez a mező nem serializálható, de szeretnénk, hogy mégis serializálható legyen az osztály, akkor kell használni ezt a kulcsszót.

A serializálás azt jelenti, hogy az objektum pillanatnyi belső állapotát byte folyamattá (angolul: byte stream) alakítjuk, azt kiírjuk egy file-ba. Amikor vissza kell állítani az elmentet állapotot, ezt nevezzük deserializációnak, akkor ez ennek a file-nak a visszaolvasásával lehetséges.

Ehhez a ObjectOutputStream writeObject metódusát tudjuk használni, legalábbis a kiíráshoz, illetve a FileInputStream readObject metódusa használható a beolvasáshoz. A kiíráshoz csak annyi kell, hogy az objektumnak legyen Serializable típusa is. Ha nincs, akkor a writeObject metódus NotSerializableException kivételt dob. Akkor is, ha esetleg valamelyik nem transient mező mégse serializálható.

Ha távoli metódus hívásra (angolul: Remote Procedure Call – RPC) van szükségünk, akkor is szükséges a serializáció, hiszen objektumot nem, de byte folyamatot képesek vagyunk átküldeni a hálózaton. Az átküldött byte folyamból pedig a másik gépen ugyanazt az objektumot visszaállíthatjuk.

Lényegében az összes Java Core API osztály serializálható, például minden kollekció, minden alaptípus becsomagoló (angolul: wrapper) osztálya, a String, stb...

Tegyük fel a szokásos kérdéseket:

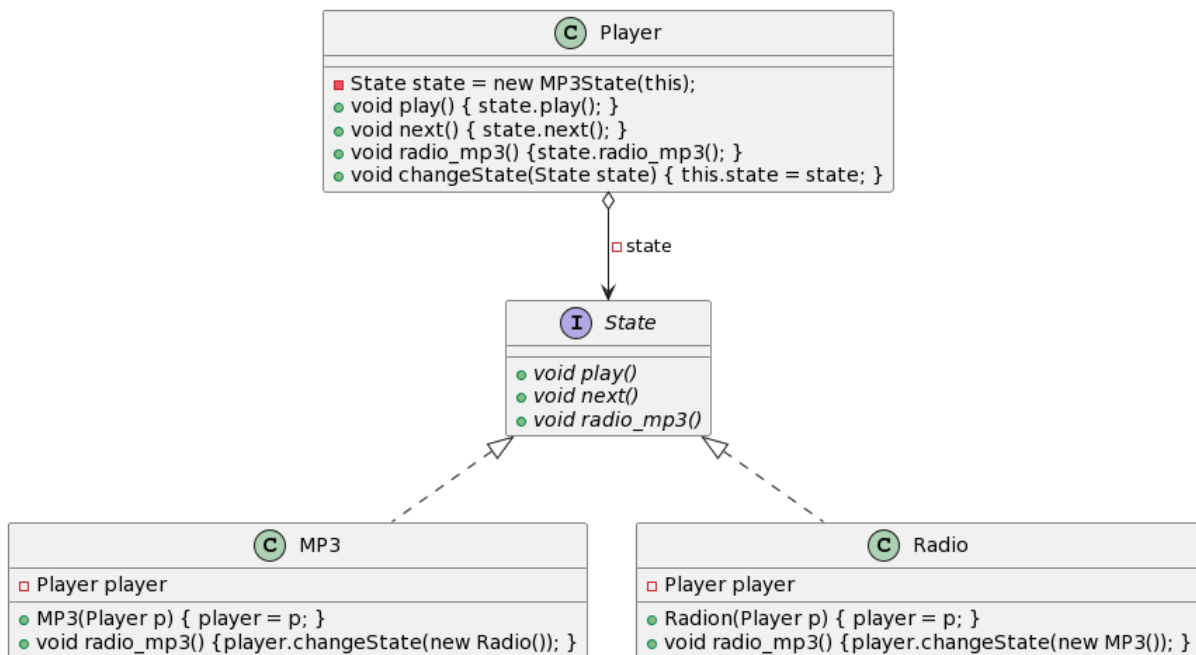
- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a Serializable interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Serializable interfész szétválasztja a serializálható objektumokat és a serializáció folyamatát. A két oldal nagyon-nagyon keveset

tud egymásról. Lényegében csak annyit, hogy egy szeriálizálható objektum minden nem tranziens mezője szeriálizálható. Mivel a két oldal szépen szétválik, ezért ez egy DIP!

Itt még lehet megfogalmazni néhány kritikát: A Serializable interfész nem Pillanatkép, hiszen nagyon eltér a klasszikus GOF könyvben található leírástól. Ez teljesen igaz, jogos kritika. Ugyanakkor a két interfésszel ugyanazt lehet elérni, pillanatfelvételt (angolul: snapshot) készíthetünk egy objektum belső állapotáról, hogy azt később vissza lehessen állítani. Ezért jogosnak érezzük, hogy ebben a fejezetben ezt a Java interfészt elemeztük egész jó mélységben. Egyszerűen a Serializable interfész terjedt el, nem a tankönyvi változat.

1.5.7. Állapot

Az Állapot (angolul: State) tervezési minta UML ábrája:



PlantUML szkriptje:

@startuml

```
class Player {
    -State state = new MP3State(this);
    +void play() { state.play(); }
    +void next() { state.next(); }
    +void radio_mp3() { state.radio_mp3(); }
    +void changeState(State state) { this.state = state; }
}

interface State {
    +{abstract} void play()
    +{abstract} void next()
}
```

```

+{abstract} void radio_mp3()
}

class MP3 {
-Player player
+MP3(Player p) { player = p; }
+void radio_mp3() {player.changeState(new Radio()); }
}

class Radio {
-Player player
+Radion(Player p) { player = p; }
+void radio_mp3() {player.changeState(new MP3()); }
}

MP3 .up.|> State
Radio .up.|> State
Player o-down-> State : -state

@enduml

```

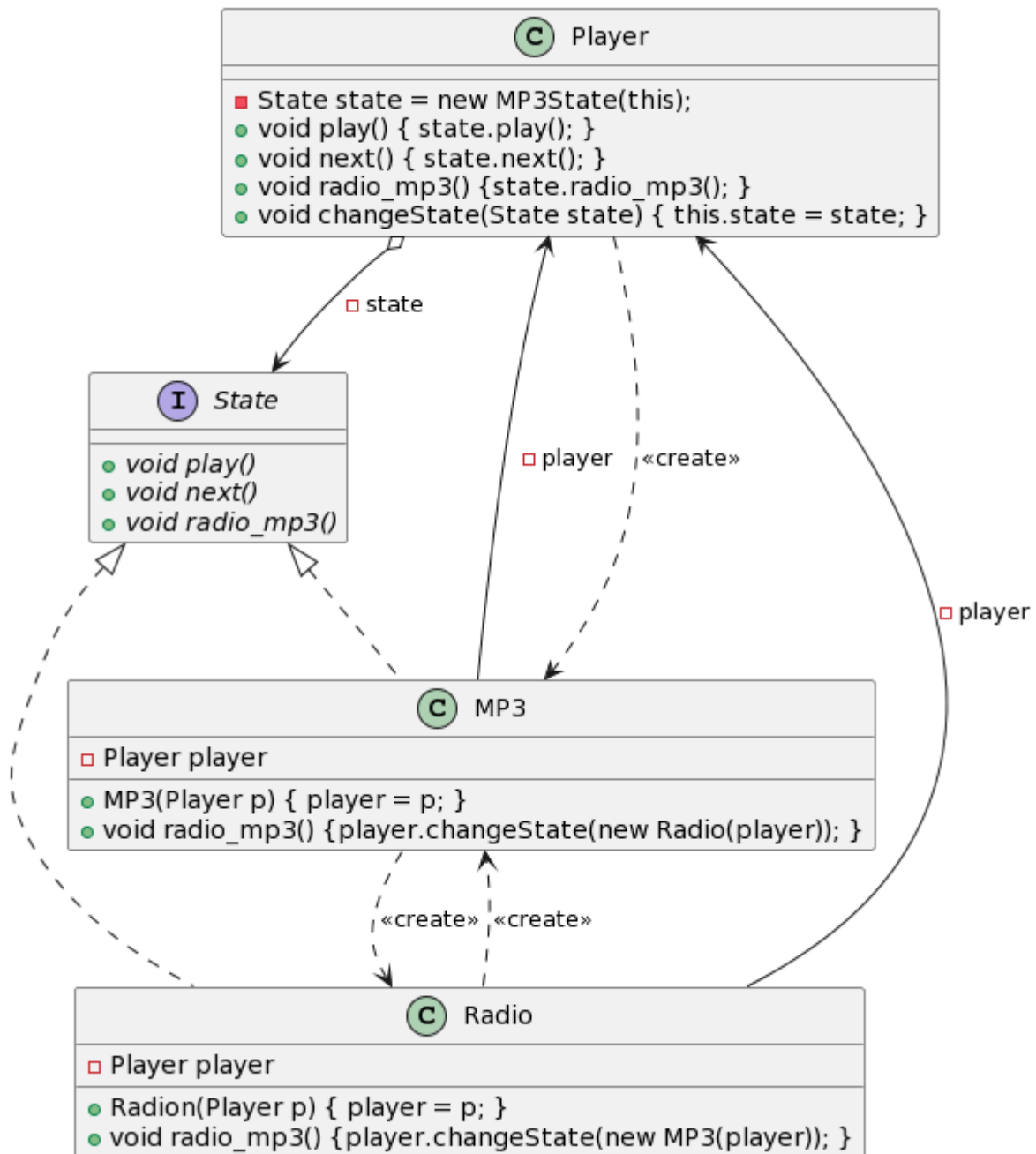
Habár ez az ábra nagyon közel van az Állapot minta hagyományos osztály diagramjához, azzal a kivétellel, hogy ez az ábra konkrétabb, mint a szokásos ábrázolások. Ez egy MP3 lejátszó modellje, ami rádióként is használható. Az Állapot mintára nagyon jellemző, hogy van egy eszközünk néhány gombbal, funkcióval, amik más-más állapotban más-más viselkedéssel bírnak, ezért az egyes konkrét állapotokat az State interfész egyes konkrét alosztályában zárjuk egységbe. Ha szükséges állapotátmenet, akkor azt a Player changeState metódussal lehet megtenni, ami a Player-ben lévő state mezőt változtatja. Eza mező hozza létre a Player és a State közti HAS-A kapcsolatot.

Minden más metódus viselkedése a State alosztályaiban van kidolgozva, ezért a Player csak delegálja a feladatot a State felé. Például így: void play() { state.play(); }.

Ahhoz, hogy az egyes állapotok képesek legyenek meghívni a Player osztályban lévő changeState metódust, ahhoz ismerniük kell. Erre szolgál az MP3 és a Radio osztály player mezője.

Ez a megoldás kritikára ad alapot, hiszen így mindkét fél ismeri a másikat, bár a player referenciát csak egy visszahívásra használjuk. A másik kritika, hogy a changeState hívásánál miért csinálunk új MP3, illetve Radio példányt, amikor a meglévő példányokat nyugodtan újrahasználhatnánk. Ez egy teljesen jogos kritika, amit egy későbbi ábrán javítunk.

Először tegyük ki a hiányzó asszociáció nyilakat:



PlantUML szkriptje:

@startuml

```

class Player {
- State state = new MP3State(this);
+ void play() { state.play(); }
+ void next() { state.next(); }
+ void radio_mp3() { state.radio_mp3(); }
+ void changeState(State state) { this.state = state; }
}

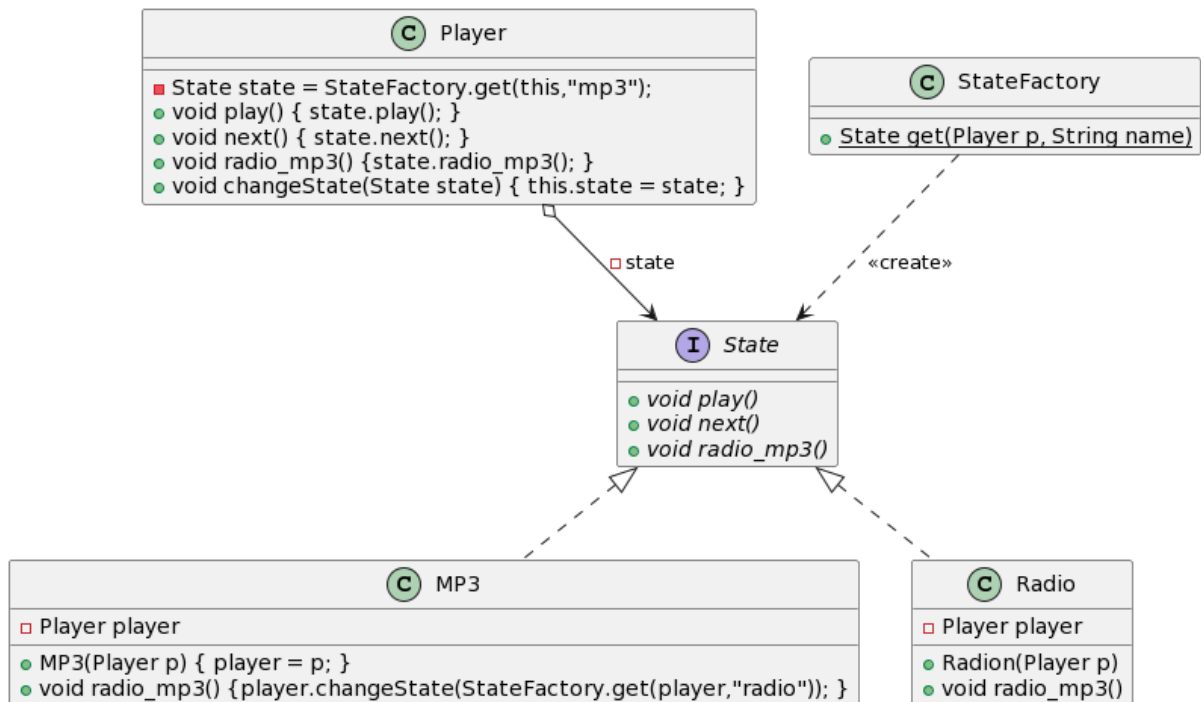
```

```

}
interface State {
+{abstract} void play()
+{abstract} void next()
+{abstract} void radio_mp3()
}
class MP3 {
-Player player
+MP3(Player p) { player = p; }
+void radio_mp3() {player.changeState(new Radio(player)); }
}
class Radio {
-Player player
+Radion(Player p) { player = p; }
+void radio_mp3() {player.changeState(new MP3(player)); }
}
MP3 .up.|> State
Radio .up.|> State
Player o-down-> State : -state
MP3 -up-> Player : -player
Radio -up-> Player : -player
Player ..> MP3 : <<create>>
MP3 ..> Radio : <<create>>
Radio ..> MP3 : <<create>>
@enduml

```

Mit az ábrán is látható, túl sok a <<create>>. Ezen egy gyár bevezetésével lehet segíteni:



PlantUML szkriptje:

@startuml

class Player {

```

- State state = StateFactory.get(this,"mp3");
+ void play() { state.play(); }
+ void next() { state.next(); }
+ void radio_mp3() { state.radio_mp3(); }
+ void changeState(State state) { this.state = state; }

```

}

interface State {

```

+ {abstract} void play()
+ {abstract} void next()
+ {abstract} void radio_mp3()

```

}

class MP3 {

```

- Player player
+ MP3(Player p) { player = p; }
+ void radio_mp3() { player.changeState(StateFactory.get(player,"radio")); }

```

}


```

class Radio {
    -Player player
    +Radion(Player p)
    +void radio_mp3()
}

class StateFactory {
    +{static} State get(Player p, String name)
}

MP3 .up.|> State
Radio .up.|> State

Player o-down-> State : -state

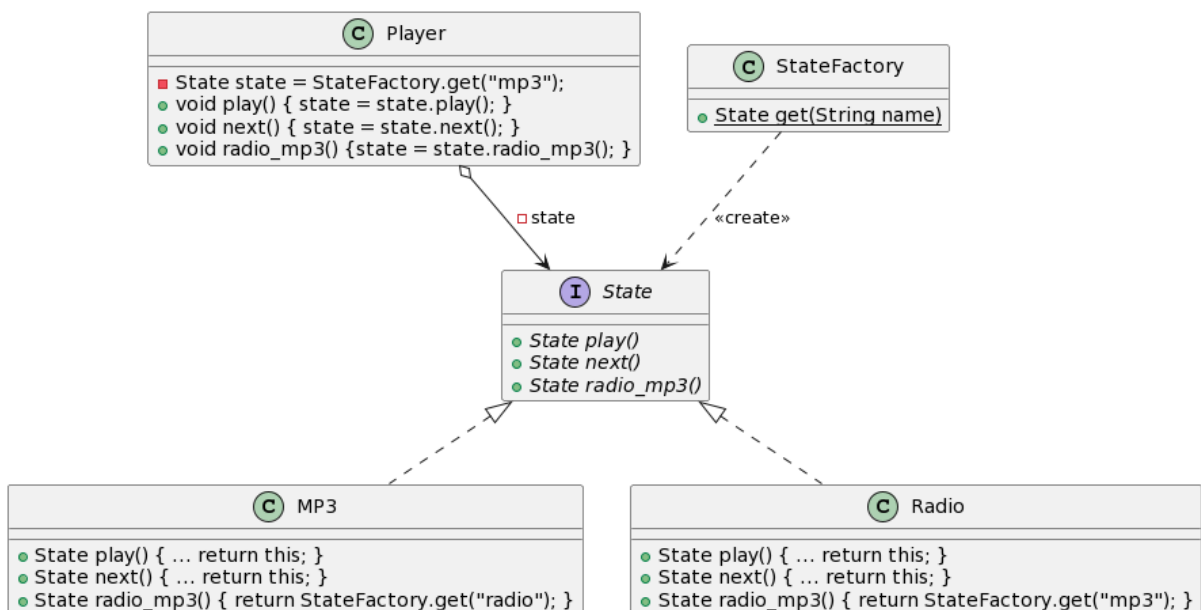
StateFactory ..> State : <<create>>

@enduml

```

Már csak az a gond, hogy körkörös kommunikáció van: Player hívja valamelyik állapotot, mondjuk Radio, ami hívja a Player-t. A jó hír, hogy a Radio, illetve az MP3 csak a `changeState` metódust hívják, amit ki lehetne emelni egy interfészbe, de ez se oldaná meg a körkörös kommunikációt, hiszen ez a minta lényege, egész addig, amíg a `changeState` metódust használjuk.

A `changeState` metódust elhagyhatjuk, ha az egy metódusok visszatérési típusa `void` helyett `State` típus lesz, ami azt tartalmazza, hogy milyen állapotba kell átmenni. Ennek a változatnak az UML ábrája:



PlantUML szkriptje:

```
@startuml
```

```

class Player {
    -State state = StateFactory.get("mp3");
    +void play() { state = state.play(); }
    +void next() { state = state.next(); }
    +void radio_mp3() {state = state.radio_mp3(); }
}

interface State {
    +{abstract} State play()
    +{abstract} State next()
    +{abstract} State radio_mp3()
}

class MP3 {
    +State play() { ... return this; }
    +State next() { ... return this; }
    +State radio_mp3() { return StateFactory.get("radio"); }
}

class Radio {
    +State play() { ... return this; }
    +State next() { ... return this; }
    +State radio_mp3() { return StateFactory.get("mp3"); }
}

class StateFactory {
    +{static} State get(String name)
}

MP3 .up.|> State
Radio .up.|> State
Player o-down-> State : -state
StateFactory ..> State : <<create>>

@enduml

```

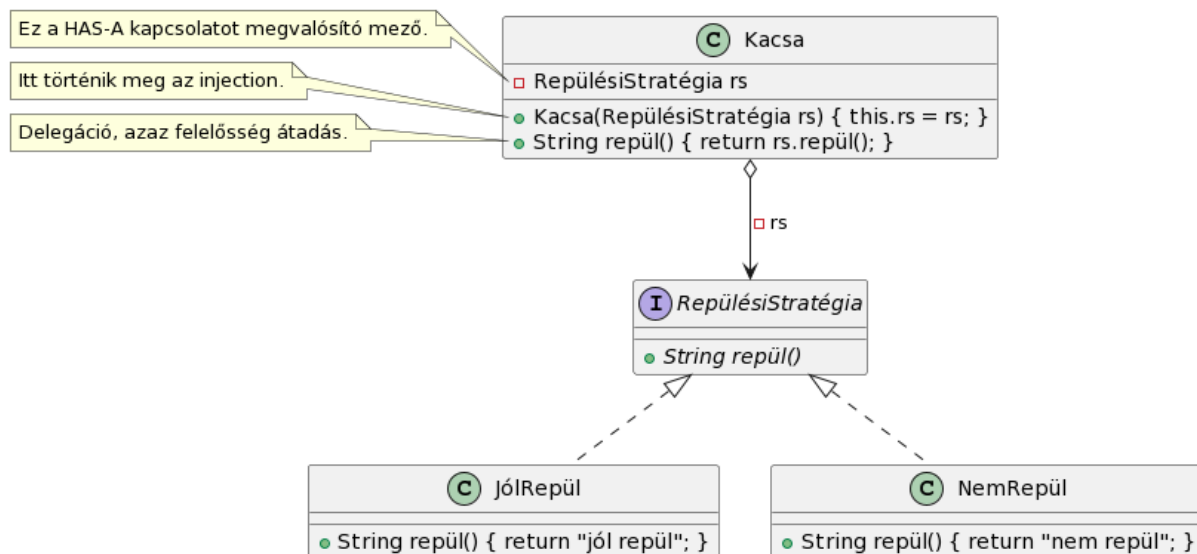
Ez a verzió már nehezen kritizálható. Ezért ezt fogjuk megvizsgálni DIP szemszögéből. Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a State interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Az ábra szerint szépen szétválik a két oldal, nincs köztük nyíl. Ugyanakkor a két oldal nagyon sok feltételezéssel él a másiktól: ha megnézzük az UML ábrát lényegében az összes metódus törzsét megírtuk. A Player tudja, hogy nyugodtan átdelegálhatja a felelőséget a State oldalnak. A State oldal pedig tudja, hogy amit visszaad visszatérési értéként, azt a Player oldalon beállítjuk új állapotként. Tehát lehet úgy is érvelni, hogy ez nem DIP, túl sokat tud a két oldal egymásról. Lehet úgy is érvelni, hogy ez DIP, a két oldal közt nincs nyíl. Illene ezt a kérdést ezen a helyen eldönteni: A fenti megoldás technikailag DIP, a Player oldalra erőltetett konvenció (a visszaadott érték legyen a state mező új értéke) miatt mégse egy tiszta DIP. A changeState-es megoldás esetén viszont nincs konvenció, tehát az utolsó előtti ábra biztosan DIP.

Egy kérdés merül még fel. Van egy jó hosszú rész a DIP és az állapotgép kapcsolatáról. Aztán egy másik jó hosszú rész, ez a rész, az Állapot tervezési minta és a DIP kapcsolatáról. És a két rész közt alig van hasonlóság. Miért? Az első rész arra koncentrál, hogy egy állapotgép hogyan írható le állapotgép UML diagrammal. Ez a rész pedig osztály diagramokat tartalmaz. Ezért van az, hogy alig látunk hasonlóságot, habár a két résznek tényleg sok köze van egymáshoz.

1.5.8. Stratégia

A Stratégia (angolul: Strategy) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
```

```
class Kacsa {
```

```
-RepülésiStratégia rs
```

```
+Kacsa(RepülésiStratégia rs) { this.rs = rs; }
```

```
+String repül() { return rs.repül(); }
```

```
}
```

note left of Kacsa::rs

Ez a HAS-A kapcsolatot megvalósító mező.

end note

note left of Kacsa::Kacsa

Itt történik meg az injection.

end note

note left of Kacsa::repül

Delegáció, azaz felelősség átadás.

end note

```
interface RepülésiStratégia{
```

```
  +{abstract} String repül()
```

```
}
```

```
class JólRepül {
```

```
  +String repül() { return "jól repül"; }
```

```
}
```

```
class NemRepül {
```

```
  +String repül() { return "nem repül"; }
```

```
}
```

Kacsa o--> RepülésiStratégia : -rs

JólRepül .up.|> RepülésiStratégia

NemRepül .up.|> RepülésiStratégia

@enduml

A Stratégia tervezési minta jól ismert. Ha van egy változékony metódusunk, akkor azt érdemes kiemelni egy osztály hierarchiába, amire egy referencia mutat. A referencián keresztül azt a változatot érjük el, amelyikre szükség van.

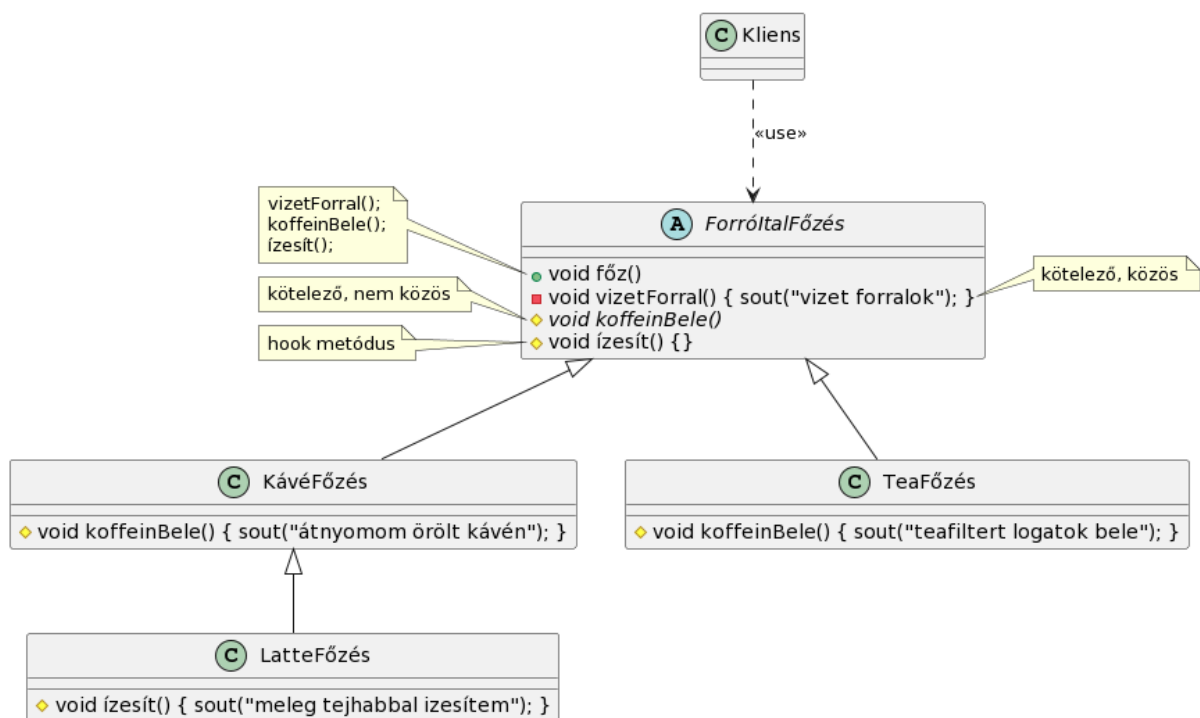
Mivel ez egy jól ismert tervezési minta, ezért itt csak a DIP-pel való kapcsolatát vizsgáljuk. Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a RepülésiStratégia interfész ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a kacsa csak annyit tud a repülési stratégiákról, hogy nyugodtan átdelegálhatja arra az oldalra a repülés felelősségét, de nem tud semmit a megvalósításról. A repülési stratégia nem tud semmit a kacsáról. Mivel a két oldal szépen szétválik, ezért ez egy DIP!

Itt még lehet megfogalmazni néhány kritikát: A fenti példa elég konkrét. Nem képzelhető el olyan eset, amikor a stratégiának vissza kell hívnia a HAS-A oldalra? De igen, ez az eset gyakori. A fenti példánál maradva elképzelhető, hogy a repüléshez a kacsza szárnyával kell csapkodni valahogy. Ehhez a stratégiának vissza kellene hívnia a kacsába. Ahhoz, hogy ezt megtehesse, kell a kacsza referenciája, azaz lesz egy visszamutató asszociáció, de attól ez még DIP marad.

1.5.9. Sablon- és Gyártó Metódus

A Sablon- és Gyártó Metódus (angolul: Template- and Factory Method) tervezési minta UML ábrája:



PlantUML szkriptje:

```

@startuml
abstract class ForróItalFőzés {
+void főz()
-void vizetForral() { sout("vizet forralok"); }
#{abstract} void koffeinBele()
#void ízesít() {}
}
note left of ForróItalFőzés::főz
vizetForral();
koffeinBele();
ízesít();
end note
Kliens ..> ForróItalFőzés : «Use»
ForróItalFőzés <|-- KávéFőzés
ForróItalFőzés <|-- TeaFőzés
KávéFőzés <|-- LatteFőzés
KávéFőzés : koffeinBele() { sout("átnyomom örölt kávé"); }
TeaFőzés : koffeinBele() { sout("teafiltert logatok bele"); }
LatteFőzés : ízesít() { sout("meleg tejhabbal ízesítem"); }

```

note right of ForróItalFőzés::vizetForral

kötelező, közös

end note

note left of ForróItalFőzés::koffeinBele

kötelező, nem közös

end note

note left of ForróItalFőzés::ízesít

hook metódus

end note

```
class KávéFőzés {
```

```
#void koffeinBele() { sout("átnyomom örölt kávé"); }
```

```
}
```

```
class TeaFőzés {
```

```
#void koffeinBele() { sout("teafiltert logatok bele"); }
```

```
}
```

```
class LatteFőzés {
```

```
#void ízesít() { sout("meleg tejhabbal ízesítem"); }
```

```
}
```

```
ForróItalFőzés <|-- KávéFőzés
```

```
ForróItalFőzés <|-- TeaFőzés
```

```
KávéFőzés <|-- LatteFőzés
```

```
class Kliens {}
```

```
Kliens ..> ForróItalFőzés : <<use>>
```

```
@enduml
```

A Sablon- és a Gyártó Metódus nagyon hasonló. Van egy receptünk, vagy algoritmusunk, amiben a lépések sorrendje fix, de, hogy az egyes lépések mit csinálnak, azt nem muszáj fixálni, azt ráhagyhatjuk a gyermekosztályokra. Általában 3 féle lépést különböztetünk meg:

- kötelező, és közös: ezeket már kidolgozzuk az Ősben,
- kötelező, de nem közös: ezek az Ősben absztraktok, a gyerekosztályokra bízuk a kidolgozását,
- nem közös, azaz opcionális: ezek az Ősben hook metódusok, a gyerekosztályok vagy kidolgozzák, vagy nem.

Az utolsó két lépés az IoC klasszikus megjelenése: nem a gyerek hívja az Ősét, hanem az Ős a gyermekét.

Itt még elemezni lehetne, hogy csak egy publikus metódus van, maga a sablon-, illetve a gyártó metódus, ami a lépések sorrendjét fixálja. Az egyes lépéseket nem szabad publikussá tenni, mert akkor azokat a kliens bármilyen, akár rossz sorrendben is hívhatná. Persze, ha a lépéseken kívül van más metódusa is, akkor azok lehetne publikusak. Mivel ezt a kérdéskört máshol már részletesen elemeztük, ezért itt ezt nagyobb mélységben nem tárjuk fel.

Érdekes kérdés viszont, hogy miért lehet a két tervezési mintát összevonni, amikor a GOF könyv szerint a Gyártó Metódus létrehozási tervezési minta, a Sablon Metódus pedig viselkedési minta. Meglátásunk szerint a két minta közt csak annyi a különbség, hogy a lépéseket sorrendjét lefixáló metódus void-os vagy sem. Ha void-os, akkor Sablon Metódus. Ha nem void-os, hanem visszaad egy terméket, akkor viszont Gyártó Metódus. Ez az álláspont még nem terjedt el a szakirodalomban, ezért ezt az állítást érdemes kritikusan fogadni.

Mivel ez egy jól ismert tervezési minta, ezért bővebben itt nem elemezzük, csak a DIP-pel való kapcsolatát vizsgáljuk a továbbiakban. Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Igen!
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a ForróItalfőzés absztrakt osztály ilyen.
- Ez az absztrakció szétválasztja a két oldalt? Igen, kliens csak annyit tud sablon- illetve gyártó metódusról, hogy az egy szem publikus metódusát kell hívnia és az valami hasznosat csinál, de hogy azt hogyan csinálja, azt nem tudja. A másik oldal tényleg nem tud semmit a kliensről. Mivel a két oldal szépen szétválik, ezért ez egy DIP!







Itt még lehet megfogalmazni néhány kritikát: Most akkor mindig csak 1 publikus metódusa van, vagy sem? A fenti szövegből mindkettő kiolvasható. Ha ezt a mintát nem kombináljuk semmilyen más mintával, akkor tényleg csak 1 publikus metódusa van. Viszont, ha kombináljuk más tervezési mintákkal, akkor már lehetnek olyan metódusai is, amik se nem sablon metódusok, se nem a sablon valamelyik lépése. Ezek a metódusok valószínűleg publikusak lesznek.

A Sablon tervezési minta nem arra jó, hogy a lépések sorrendjének fixálását és lépések kidolgozását elválassza egymástól? A fentiekben azt írtuk, hogy a klienst választja el a sablon kidolgozásától. Mindkettő igaz. IoC-vel választjuk el egymástól a lépések sorrendjének fixálását és a lépések kidolgozását. DIP-pel választjuk el a klienst a sablon kidolgozásától.

1.6. Azok a tervezési minták, amiben nincs DIP

Ebben a fejezetben vegyesen látunk létrehozási-, szerkezeti- és viselkedési tervezési minták is. Ezek azok, ahol habár a DIP egyes nyomai felfedezhetőek, de nem elég tiszta az alkalmazása ahhoz, hogy nyugodt szívvel kijelenthessük, hogy a tervezési minta megfelel a DIP alapelvnek.

1.6.1. Egyke

 Singleton
 String globálisErőforrás
 Singleton instance = new Singleton();
 Singleton()
 Singleton getInstance() { return instance; }
 String globálisSzolgáltatás()

PlantUML szkriptje:

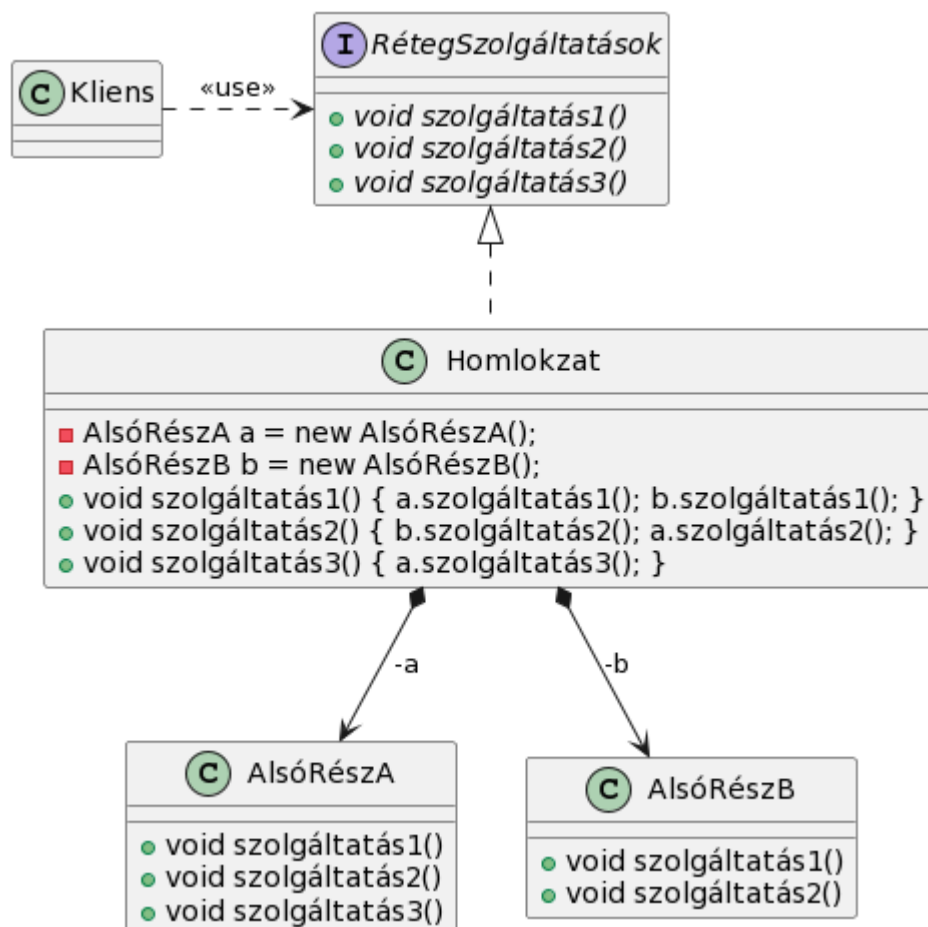
```
@startuml
class Singleton {
- {static} Singleton instance = new Singleton();
- String globálisErőforrás
- Singleton()
+ {static} Singleton getInstance() { return instance; }
+ String globálisSzolgáltatás()
}
@enduml
```

Ez a minta jól ismert, ezért leírását, elemzését mellőzzük. Ugyanakkor megvizsgáljuk, hogy milyen kapcsolatban áll a DIP alapelvvel. Mivel ebben a tervezési mintában nincs se IS-A, se HAS-A kapcsolat, hiszen az egész minta egy osztályból áll, ezért itt nyilván nincs DIP.

Érdekes kérdésként felmerülhet, hogy ha nincs is DIP, de van-e szétválasztás, mondjuk IoC. Az IoC, illetve más szétválasztás nyomát sem sikerült felfedeznünk.

1.6.2. Homlokzat

Az Homlokzat (angolul: Façade) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
class Kliens {}
interface RétegSzolgáltatások {
+{abstract} void szolgáltatás1()
+{abstract} void szolgáltatás2()
+{abstract} void szolgáltatás3()
}
class Homlokzat {
-AlsóRészA a = new AlsóRészA();
-AlsóRészB b = new AlsóRészB();
+void szolgáltatás1() { a.szolgáltatás1(); b.szolgáltatás1(); }
+void szolgáltatás2() { b.szolgáltatás2(); a.szolgáltatás2(); }
+void szolgáltatás3() { a.szolgáltatás3(); }
}
class AlsóRészA {
+void szolgáltatás1()
+void szolgáltatás2()
+void szolgáltatás3()
}
class AlsóRészB {
+void szolgáltatás1()
+void szolgáltatás2()
}
Kliens .right.> RétegSzolgáltatások : <<use>>
RétegSzolgáltatások <|.. Homlokzat
Homlokzat *--> AlsóRészA : -a
Homlokzat *--> AlsóRészB : -b
@enduml
```

A Homlokzat tervezési minta úgy él a köztudatban, hogy egy bonyolult osztályrendszer egyszerűsített interfészét adja, amin keresztül elérhető az osztályrendszer szolgáltatásai. Tehát, mint interfész szoktunk rá gondolni. Ugyanakkor ez szemben áll azzal leírással, hogy a homlokzatnak kell tudnia,

hogy az egyes szolgáltatások elvégzéséhez mely metódusokat kell meghívni a nehezen átlátható rendszeren belül. Ehhez bizony referenciák kellene, amin keresztül meghívhatja a megfelelő metódusokat. Ha referenciákat tárol, akkor van belső állapota, ha van belső állapota, akkor nem lehet interfész.

Az ellentmondást úgy lehet feloldani, hogy:

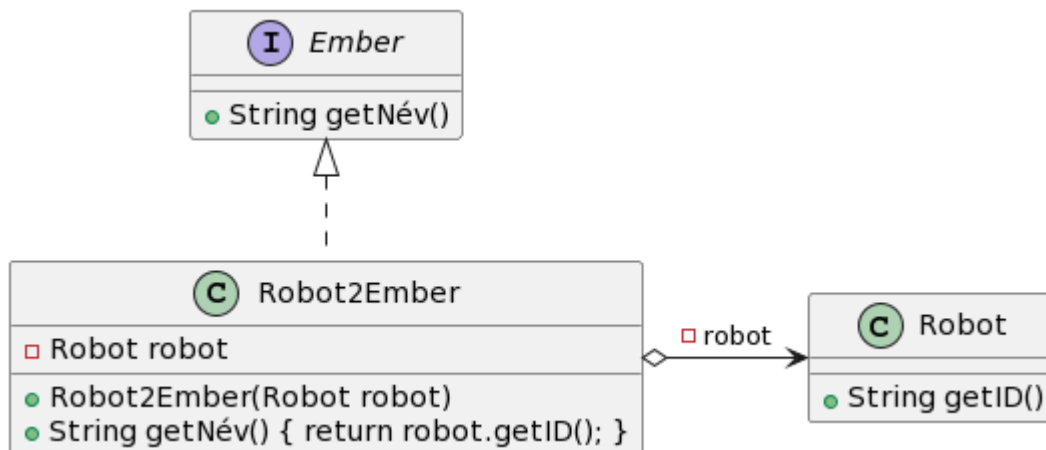
1. valóban van egy interfész, ami a szolgáltatásokat sorolja fel, és
2. van egy megvalósító osztály, ami tudja, hogyan kell ezeket a szolgáltatásokat megvalósítani.

Hogy a kettő közül melyiket nevezzük Homlokzatnak, az már értelmezés függő. Mi most a második entitást tekintjük Homlokzat osztálynak, az első RétegSzolgáltatások interfésznek.

Nyilván a RétegSzolgáltatások interfész egy DIP, hiszen pont ez a DIP definíciója, hogy a felső és az alsó réteget elválasztó absztrakció. Ugyanakkor a tervezési minta széleskörben elfogadott leírása a Homlokzat osztálynak felel meg, ami nyilván nem DIP eredménye, hiszen még csak nem is absztrakt.

1.6.3. Illesztő

Az Illesztő (angolul: Adapter) tervezési minta UML ábrája:



PlantUML szkriptje:

```
@startuml
class Robot {
+String getID()
}
interface Ember {
+String getNév()
}
class Robot2Ember {
- Robot robot
+ Robot2Ember(Robot robot)
}
```

@startuml

```

interface BinFa {
    +{abstract} void acceptVisitor(Visitor v)
}

class Ág {
    -BinFa bal, jobb
    +Ág(BinFa bal, BinFa jobb)
    +BinFa getBal()
    +BinFa getJobb()
    +void acceptVisitor(Visitor v) {v.visit(this);}
}

class Levél {
    -int szám
    +Levél(int szám)
    +int getSzám()
    +void acceptVisitor(Visitor v) {v.visit(this);}
}

class NullFa {
    +void acceptVisitor(Visitor v) {v.visit(this);}
}

interface Visitor {
    +{abstract} void visit(Ág ág)
    +{abstract} void visit(Levél levél)
    +{abstract} void visit(NullFa nullFa)
    +{abstract} Object getResult()
    +{abstract} void reInit()
}

class SumVisitor {
    -int sum
    +void visit(Ág ág) { ág.getBal().acceptVisitor(this); ág.getJobb().acceptVisitor(this); }
    +void visit(Levél levél) { sum += levél.getSzám(); }
    +void visit(NullFa nullFa) {}
}

```

```
+Object getResult() { return szum; }

+void reInit() { sum = 0; }

}
```

Ág .up.|> BinFa

Levél .right.|> BinFa

NullFa .up.|> BinFa

Ág o--> BinFa : -ball, jobb

Visitor <|.. SumVisitor

Visitor .up.> Ág : <<visit>>

Visitor .up.> Levél : <<visit>>

Visitor .up.> NullFa : <<visit>>

BinFa ..> Visitor : <<accept>>

@enduml

Azt szoktuk mondani, hogy a Látogató tervezési minta az egyetlen, amit józan paraszti ésszel nehéz kitalálni. Minden más tervezési mintát egy kis tapasztalat után és egy kis józan paraszti ésszel bárki képes újra felfedezni. Ennek oka a dupla hivatkozásfeloldás (angolul: double dispatch).

Általában csak egyszeres hivatkozásfeloldásra van szükségünk, a késő kötésre. Ha van egy polimorfikus metódus hívásunk, például: referencia.valamiHasznos(), akkor a késő kötés oldja fel, hogy az osztály hierarchiában lévő több valamiHasznos() nevű metódus közül melyik fog lefutni. Ehhez a késő kötés megnézi, hogy a referencia milyen példányra mutat, és a példány metódusa kerül meghívásra.

Ugyanakkor van egy másik eset is, amikor hivatkozásfeloldásra van szükségünk, ez a metódus túlterhelés (angolul: overloading), amikor egy metódusból több változat van más-más paraméter listával. Például, tegyük fel, hogy a valamiHasznos metódust túlterheltük:

- void valamiHasznos(int i) { ... }
- void valamiHasznos(String s) { ... }

Most, ha belefutunk egy hívásba: valamiHasznos(x), akkor a x kifejezés típusától függ, hogy melyik változat fog lefutni.

Dupla hivatkozásfeloldás (angolul: double dispatch) esetén mind a két fajta hivatkozásfeloldásra szükségünk van. Például a **referencia.valamiHasznos(x)** hívásnál először késői kötéssel kell meghatároznunk, hogy melyik osztály példányából kell futtatni a metódust, majd az x kifejezés típusának segítségével kell meghatároznunk, hogy melyik túlterhelt változatnak kell futnia.

Pontosan ezt használja ki a Látogató tervezési minta. Nézzük részletesen, hogy hogyan. Látogató tervezési mintát akkor használunk, ha már van egy jól kiforrott, kitesztelt adatszerkezetünk, amit már nem akarunk bővíteni. Ez a fenti példában a BinFa interfész, illetve megvalósító osztályai, amik egy bináris fa adatszerkezetet írnak le, amiben a levél elemek tárolnak egy-egy számot, az ágak nem,

minden ágnak két részfája van, ha mégis csak egy lenne, akkor a másik ág egy nullfa, azaz null értéket nem engedünk.

Ahhoz, hogy az adatszerkezetet képes legyen feldolgozni egy látogató, ehhez minden osztály implementálja az acceptVisitor metódust, méghozzá mindenki teljesen ugyanúgy:

```
void acceptVisitor(Visitor v) { v.visit(this); }
```

Azaz fogadok egy látogatót, v, majd rögtön meghívom a visit metódusát önmagammal: **v.visit(this)**. Erről nagyon könnyen felismerhető a Látogató tervezési minta, és itt van a dupla hivatkozásfeloldás. Ahhoz, hogy tudjam, hogy melyik metódusnak kell lefutnia, tudnom kell milyen látogatóra mutat a v referencia. És hogy ezen belül a visit metódusnak melyik változata fusson le, ahhoz tudnom kell a this típusát, ugyanis a visit-nek az Ősön kívül minden adatszerkezet osztályra fel kell készülnie:

- void visit(Ág ág)
- void visit(Levél levél)
- void visit(NullFa nullFa)

Ha a this típusa Ág volt, akkor az első változat fut le, ha Levél, akkor a második, ha NullFa, akkor a harmadik. Igen, jól látja a kedves olvasó, ez a funkcionális nyelveknél megszokott paraméter alapú esetszétválogatás (angolul: case distinction), ami egy nagyon erős programozói eszköz. Lám, OOP oldalon is megvalósítható!

Az Ős osztályra / interfészre nem szabad visit változatot készíteni, mert az Ős minden gyerekosztályával kompatibilis.

Így, hogy már tudjuk, hogy mit kell feldolgozni, így már könnyű dolgunk van. Ha ágat kapunk, akkor az ág bal és jobb oldalára is elküldjük magunkat:

```
void visit(Ág ág) { ág.getBal().acceptVisitor(this); ág.getJobb().acceptVisitor(this); }
```

Ha levelet kapunk, akkor fel kell dolgozni a levélben lévő számot, lásd az ábrán. Nullfa esetén általában nem kell csinálni semmit, szintén lásd az ábrát.

Még kell két metódus, a getResult, amivel visszaadja a látogató eredményét, és a reInit, ami újra inicializálja a látogatót. Ugyanis a látogatónak van belső állapota, amit két látogatás közt megőrizne, ha a két látogatás közt nem adnánk ki a reInit parancsot.

Tényleg nem egy egyszerű tervezési minta. Azon túl, hogy elég alaposan elemeztük, nézzük meg, hogy milyen kapcsolatban áll a DIP alapelvvel. Tegyük fel a szokásos kérdéseket:

- Minden nyíl absztrakcióra mutat? Nem! Mivel a látogatónak ismernie kell a konkrét osztályokat, akiket meg kell látogatnia.
- Van olyan absztrakció, amire HAS-A és IS-A kapcsolat is mutat? Igen, a BinFa és a Visitor interfész is ilyen. Ugyanakkor a BinFa interfészt kizárhatjuk az elemzésből, mert az az adatszerkezet része, helyette bármilyen más adatszerkezet is lehetne. Tehát a Visitor interfészt vizsgáljuk.
- Ez az absztrakció szétválasztja a két oldalt? Igen, a Visitor interfész elválasztja egymástól az adatszerkezetet és a látogatók oldalt, de mint látni fogjuk, ez nem elég. Az adatszerkezetet úgy tudjuk új műveletekkel bővíteni, hogy az adatszerkezetet kódja változatlan, csak egy új látogató alosztályt kell készítenünk. Ehhez a látogatónak alaposan ismernie kell az adatszerkezetet, de csak az adatszerkezet felületét. Az adatszerkezetnek csak annyit kell

tudnia, hogy ez `acceptVisitor(Visitor v)` metódust így kell implementálnia: `v.visit(this)`. Sajnos ez egy konvenció, amit, ha nem tart be az adatszerkezet, akkor nem fog működni a tervezési minta. Konvenciókat ráerőltetése egyik, vagy másik oldalra, nem fér bele a DIP alapelvbe. Ezért ez nem DIP!

Itt még lehet megfogalmazni néhány kritikát: Miért nem DIP? Az adatszerkezetet úgyse kell változtatni, ha már egyszer lehetővé tettük, hogy tudjon látogatót fogadni. A konvenció nem csak akkor káros, amikor később hozzá akarunk nyúlni a kódhoz, és ha már elfelejtettük a konvenciót? Ez mind igaz, a konvenció, akkor okoz gondot, ha megfeledezünk róla, vagy nem is tudjuk, hogy van. A gond ott van, hogy a DIP kéz a kézben kell, hogy járjon a szerződésekkel, és a fenti konvenciót nem könnyű szerződésbe önteni. Ezért ez a tervezési minta nem felel meg a DIP elvnek.

Nem lehet valahogy kikényszeríteni vagy elkerülni ezt a fránya konvenciót, mint ahogy az Állapot tervezési mintánál a `changeState` metódus elkerülhetővé tette a konvenciót? Őszintén, nagyon örülnénk egy ilyen megoldásnak, de egyelőre nem látjuk, hogy lehetne a visszahívást elkerülni, vagy kikényszeríteni, hiszen a visszahívás nélkül nincs dupla hivatkozásfeloldás, ami a minta fő lépése.

1.7. Mikor ne használjuk a DIP tervezési alapelvet

Mit láttuk, a DIP egy erős eszköz, amit sok helyen használnak nagyon sikeresen. Ugyanakkor a DIP-nek van veszélye is. Hajlamosak vagyunk túlzásba vinni a használatát! Például, ha előre láthatóan egy rétegben csak egy megvalósító osztály lesz, akkor felesleges tenni felé egy interfészt. Gondoljunk arra a programozó társunkra, akinek majd használnia kell ezt az osztályt. A hívás helyéről először egy interfészhez fog eljutni, és csak utána a konkrét megvalósító osztályhoz.

Erre persze mondhatnánk, hogy mit érdekli a másik programozót, hogy hogyan implementáltunk egy konkrét osztályt. Nézze meg az API leírást, ami az interfészhez tartozik, annak elegendő információt kell tartalmaznia. Ugyanakkor lehet, hogy ez a programozó most ismerkedik a rendszerrel, a használt megoldásokkal, vagy egyszerűen csak hibát keres. A feleslegesen felhúzott interfészek gyakran felesleges bosszúságot okoznak.

Mi van akkor, ha nincs interfész, mert csak egy megvalósító osztály van, és kiderül, hogy mégis kell egy második megvalósítás, és ezért már indokolt az közös dolgok kiemelése egy közös interfészbe, vagy absztrakt osztályba? Igazán semmi gond, refaktroláljuk a programot és bevetünk egy DIP megoldást. Ez teljesen belefér az agilis gondolkozásba.

A gond akkor van, ha előre látható volt, hogy lesz második megvalósító osztály 1-2 sprinten belül. Ilyenkor, ha nem használjuk a DIP-et már ebben a sprintben, akkor az idő pocséklás.

Továbbá, nem szabad szétválasztani dolgokat, habár szét lehetne őket választani, ha józan paraszti ésszel gondolkozva azok inkább egybe tartoznak. Itt nehéz szabályokat megfogalmazni, de azért nézzünk két jó tanácsot.

1. Ha a követelmény specifikációban ez a két dolog majd mindig együtt szerepel, akkor azokat inkább ne válasszuk szét. Erre példa az állapottér reprezentáció mesterséges intelligenciából. Az állapot és az állapot operátorai, habár szétválaszthatók, de nem érdemes őket szétválasztani.
2. Ha két dolog könnyen leírható egy jelzős szerkezettel, ugyanakkor egyenrangú félként használva csak nyakatekert módon tudunk rájuk hivatkozni, akkor jobb nem szétválasztani őket. Ilyen lehet a piros labda, ahol a piros egy jelzője, informatikai szóhasználattal, jellemzője a labdának. Ha piros labdát úgy akarom megfogalmazni, hogy a piros és a labda

egyenrangú entitások, akkor ilyen nyakatekert megfogalmazásokat kapunk: Olyan játékszer, ami egyszerre piros és labda.

Ezektől függetlenül, a tervezőnek, ha csak egy kis esélyt is lát arra, hogy szükséges a szétválasztás, akkor a használnia kell a tervben a DIP-et. Ugyanakkor a tervet átnéző csapattagoknak, programozóknak, pedig rögtön jelezniük kell, ha csak egy kicsit is, de feleslegesnek látják a DIP használatát. Ezekből a vitákból, ezekből a harcokból születik a jó terv! Persze a Scrum mester rögtön avatkozzon közbe, ha a tervezők és a programozók nagyon összevesznének. Kicsit összeveszni, a jó terv érdekében, lehet, sőt kötelező!