# *Programozás III*

KOMPOZÍCIÓ,
TERVEZÉSI MINTÁK

és EGYEBEK

## ELŐZŐ ELŐADÁS FELMÉRÉS:

Eddigi OOP ismeretekhez képest

| | | |
|---|---|---|
| a) | semmi újat nem kaptam | 13 |
| b) | minden új volt | 8 |
| c) | megértettem a korábbiakat | 46 |

A tempó

| | | |
|---|---|---|
| a) | lassú | 3 |
| b) | megfelelő | 36 |
| c) | gyors | 26 |

Ismétlés

| | | |
|---|---|---|
| d) | Túl sok volt az ismétlés | 3 |
| e) | Sok volt az ismétlés, de szükség van rá. | 45 |
| f) | Nem volt sok ismétlés | 15 |

 35  0

Megjegyzés:

## ELŐZŐ ELŐADÁS - ISMÉTLÉS:

Kompozíció, aggregáció fogalma

Reguláris kifejezés:

Bizonyos szintaktikai szabályok alapján leírt string, amely segítségével meghatározható stringek egy halmaza.

Az ilyen kifejezés valamilyen minta szerinti szöveg keresésére, cseréjére, illetve a szöveges adatok ellenőrzésére használható.

Jó kiindulás:

https://hu.wikipedia.org/wiki/Reguláris_kifejezés

## MÉG MINDIG ISMÉTLÉS

Különböző **sportoló** típusok vannak:
**futó, magasugró, focista**,
akik teljesítményét más-más módon határozzuk meg.

Az egyszerűség kedvéért mindegyiket a nevük alapján regisztráljuk, és egy kérdésre adott válaszként dől el, hogy milyen sportágat űznek.

## MEGOLDÁSRÉSZLET

Vezérlő osztály:

```
   private List<Sportolo> sportolok = new ArrayList<>();

  public void adatBe(){
do{
    System.out.print("\n" +(i+1) + ". sportolo neve: ");
    nev = scanner.next();
    System.out.print("Sportága (1: ugró; 2: focista; 3: futó): ");
    sportag = scanner.next();
        // ellenőrzött beolvasás kell!!
    if(sportag == 1) sportolok.add(new Ugro(nev));
    if(sportag == 2) sportolok.add(new Focista(nev));
    if(sportag == 3) sportolok.add(new Futo(nev));
        // switch case-zel elegánsabb
    System.out.print("Van meg jelentkezo? (i/n)");
 }while(Input.readChar() == 'i');
```

## MEGOLDÁSRÉSZLET

Kevésbé „fapadosan”: egy újabb osztály közbeiktatásával

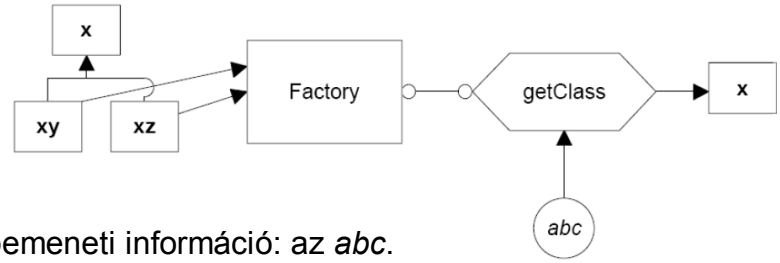
```
public class Versenyzo {

    public Sportolo getVersenyzo(String nev, String sportag){
        if (sportag.equals("foci")){
          return new Focista(nev);
         }
        if (sportag.equals("ugras")) {
           return new Ugro(nev);
        }
        if (sportag.equals("futas")){
           return new Futo(nev);
        }
        return new Sportolo(nev);
    }
}
```

## FACTORY

Ez az ú.n. „gyártó függvény” (**Factory Method**) tervezési minta.

A minta célja:
egy objektum létrehozása különböző információk alapján.




bemeneti információ: az *abc*.
Ez alapján a gyártófüggvény az *x* ősosztály valamelyik leszármazottját (*xy* vagy *xz*) példányosítja.
A *getClass* hívására létrejövő objektum tényleges típusáról többnyire nem is kell tudnia a felhasználónak.

## A PÉLDA MEGOLDÁSA FACTORY-VAL

Vezérlő osztály:

```
    private List<Sportolo> sportolok = new ArrayList<>();

public void adatBe(String sportag){

     ...
     do{
     ...
         sportolok.add(new Versenyzo().getVersenyzo(nev,sportag));

         System.out.print("Van meg jelentkezo? (i/n)");
      }while(Character.toUpperCase(Input.readChar()) == 'I');
 }
```

## TERVEZÉSI MINTÁK

Gyakran előfordul, hogy a fejlesztés során hasonló feladatokat kell megoldani.

A tervezési minták olyan objektum alapú megoldásokat, megoldásvázlatokat jelentenek, amelyek már bizonyítottak a gyakorlatban, és amelyeket érdemes felhasználni a saját tervezés során.

Ezek felhasználásával rugalmasabb, könnyebben újra-hasznosítható, módosítható alkalmazásokat készíthetünk.

## TERVEZÉSI MINTÁK CSOPORTOSÍTÁSA

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

## TERVEZÉSI MINTÁK

**Létrehozási minták célja:**
Az objektumok létrehozása során speciális igényeket is ki lehessen elégíteni.

**Szerkezeti minták célja:**
Annak leírása, hogy hogyan lehet komplexebb struktúrákat létrehozni;

jól használható programfelületet nyújtó öröklési hierarchia kialakítására;

az objektumok összeillesztési módjának meghatározására.

## TERVEZÉSI MINTÁK

**Viselkedési minták célja:**
Az objektumok egymás közti kommunikációjának segítése.

**Annak, aki programozói állást szeretne:**

Erich Gamma, Ralph Johnson, Richard Helm,
John Vlissides: **Programtervezési minták** (Kiskapu kiadó)

Rövidke kedvcsináló bevezető:
Nagy Gusztáv: Java programozás

http://nagygusztav.hu/java-programozas

Rövid és jó összefoglaló:
https://www.tutorialspoint.com/design_pattern/index.htm

## MÉG EGY FELADAT

Probléma:

Ha az 5 fül 5 különböző panel-osztály, akkor állandóan át kell adogatni egymásnak a bankkártyák listáját.


Megoldás:

a/ statikus változó

b/ singleton tervezési minta

## SINGLETON (EGYKE) TERVEZÉSI MINTA

Olyankor alkalmazható, amikor garantáltan egyetlen példányt akarunk létrehozni az osztályból.

(Rugalmasabb, mint a statikus változó, mert bővíthető.)

```
public class PeldaOsztaly {
    private static PeldaOsztaly peldany = null;

    private PeldaOsztaly(){
    }

    public static PeldaOsztaly getPeldany() {
        if(peldany == null){
            peldany = new PeldaOsztaly();
        }
        return peldany;
    }
}
```

## SINGLETON (EGYKE) TERVEZÉSI MINTA

Használata:

```
public class Egyik {
    public void egyikMetodus(){
        PeldaOsztaly peldany = PeldaOsztaly.getPeldany();
        System.out.println(peldany);
    }
}
   public class Masik {
        public void masikMetodus(){
          PeldaOsztaly peldany = PeldaOsztaly.getPeldany();
          System.out.println(peldany);
       }
   }
        public static void main(String[] args) {
            Egyik egyik = new Egyik();
            Masik masik = new Masik();

            egyik.egyikMetodus();
            masik.masikMetodus();
        }
```

## MÉG EGY SINGLETON PÉLDA

```
public class DiakLista extends ArrayList<Diak>{

    private static DiakLista peldany = null;

    private DiakLista(){
    }

    public static DiakLista getPeldany() {
        if(peldany == null){
            peldany = new DiakLista();
        }
        return peldany;
    }
}
```

vagy: `public class DiakLista<Diak> implements List<Diak>{`

Ekkor mindent implementálnunk kell.

## MÉG EGY SINGLETON PÉLDA

Használata:

```
public class Egyik {
    public void egyikMetodus(){
        String nev = "Kati", eha ="12345";
        int szulEv = 1992;

        DiakLista diakLista = DiakLista.getPeldany();
        diakLista.add(new Diak(nev, eha, szulEv));
    }
}
   public class Masik {
       public void masikMetodus() {
           DiakLista diakLista = DiakLista.getPeldany();
           System.out.println(diakLista.size());
       }
   }
```

## MÉG EGY PÉLDA

```
public class Main {

    public static void main(String[] args) {
        getMain().indit();
    }

    private static Main peldany;

    private Main() {
    }

    public static Main getMain() {
        if (peldany == null) {
            peldany = new Main();
        }
        return peldany;
    }

    private void indit() {
        // ...
    }
}
```

## TERVEZÉSI MINTÁK – ENYHE RÉSZLETEZÉS

**Létrehozási tervezési minták:**
A létrehozási minták feladata, hogy megszüntessék a sok new kulcsszóval ránk szakadó függőségeket. Ha úgy írjuk meg a programunkat, hogy mindenhová a new Kutya() hívást írjuk, amikor Kutya példányra van szükségünk, akkor nehéz lesz ezt lecserélni egy későbbi new SzuperKutya() hívásra. Jobban járunk, ha a „gyártást" a létrehozási mintákra hagyjuk és például így készítjük a kutyáinkat: kutyaGyár.createKutya(). Ilyenkor, ha változnak a követelmények, akkor csak egy helyen kell változtatni a létrehozás módját. Ott, ahol létrehozzuk a kutyaGyár példányt.

## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

**Factory method (gyártó fv):**
Ezzel a mintával lehet szépen kiváltani a programunkban lévő rengeteg hasonló new utasítást. A minta leírja, hogyan készítsünk gyártófüggvényt.
A gyártófüggvény a nevében magadott terméket adja vissza, tehát a készítKutya (createDog) egy kutyát, a készítMacska (createCat) egy macskát. Ez azért jobb, minta a new Kutya() vagy a new Macska() konstruktor hívás, mert itt az elkészítés algoritmusát egységbe tudjuk zárni. Ez azért előnyös, mert ha a gyártás folyamata változik, akkor azt csak egy helyen kell módosítani. Általában a gyártás folyamata ritkán változik, inkább az a kérdés, mit kell gyártani, azaz ez gyakran változik, ezért ezt a gyermek osztály dönti el.

## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

Az ősben lévő gyártómetódus leírja a gyártás algoritmusát, a gyermek osztály eldönti, hogy mit kell pontosan gyártani. Ezt úgy érjük el, hogy az algoritmus 3 féle lépést tartalmazhat:

1. A gyártás közös lépései: Ezek az ősben konkrét metódusok, általában nem virtuálisak, illetve Java nyelven final metódusok.
2. A gyártás kötelező változó lépései. Ezek az ősben absztrakt metódusok, amelyeket a gyermek felülír, hogy eldöntse, mit kell gyártani. A gyermek osztályok itt hívják meg a termék konstruktorát.
3. A gyártás opcionális lépései: Ezek az ősben üres törzsű metódusok.

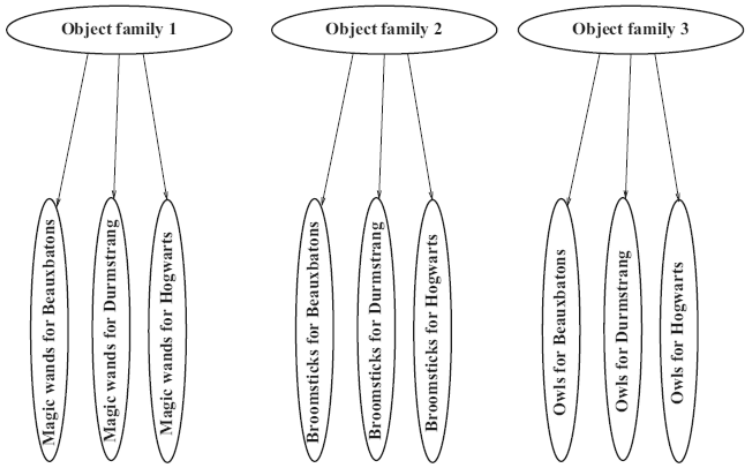
## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

1. példa:
Az Office csomag alkalmazásiban lévő Új menüpont. Ez minden alkalmazásban létrehoz egy új dokumentumot, és megnyitja. A megnyitás közös, de a létrehozás más és más. A szövegszerkesztő esetén egy üres szöveges dokumentumot, táblázatkezelő esetén egy üres táblázatot kell létrehozni.

2. példa:
„Sörgyár” – ld.:
http://wiki.javaforum.hu/display/JAVAFORUM/Abstract+Factory

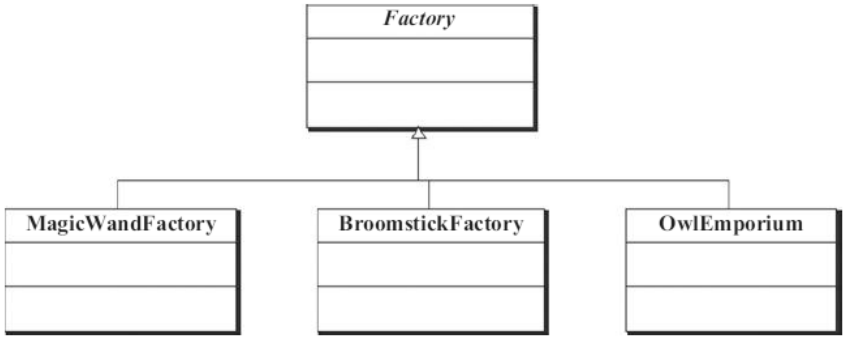
## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

3. példa – Harry Potter fanoknak:
Különböző varázslatos dolgokra van szükségünk:



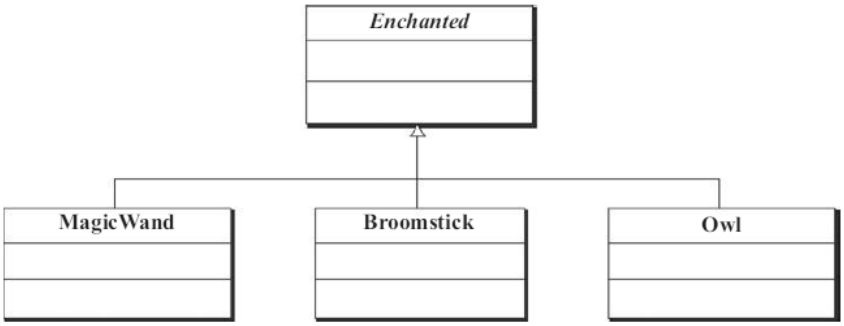

## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

Ezeket különböző „varázs-gyárakban” gyártjuk le, a gyártási folyamatban azonban sok közös vonás van:



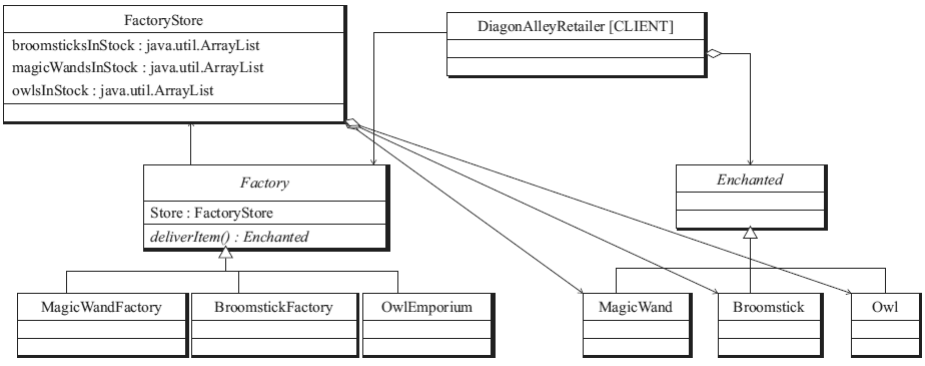
## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

A gyártott példányok mindegyike valamilyen varázslásra (bűbájra) szolgál:




## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

A két osztályhierarchiát a Diagon Alley kereskedője fogja össze:

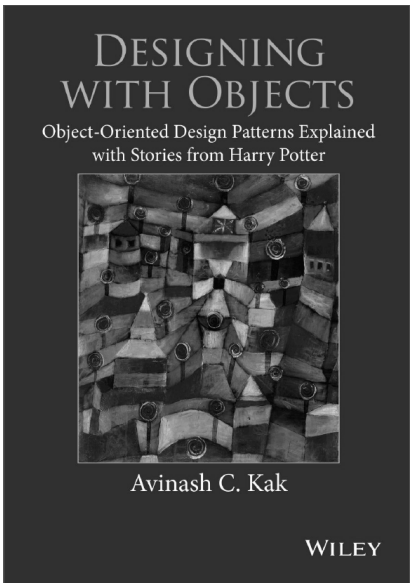


## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

A 3. (és a hozzá hasonló) példák forrása:

Letölthető kódok:

https://engineering.purdue.edu/kak/DesigningWithObjects/dwocode.html




## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

4. példa – Java standard library:

Paraméterezéstől függően a Calendar osztály getInstance() metódusa más-más példányt ad vissza:

java.util.Calendar – getInstance()
java.util.Calendar – getInstance(TimeZone zone)
java.util.Calendar – getInstance(Locale aLocale)
java.util.Calendar – getInstance(TimeZone zone, Locale aLocale)

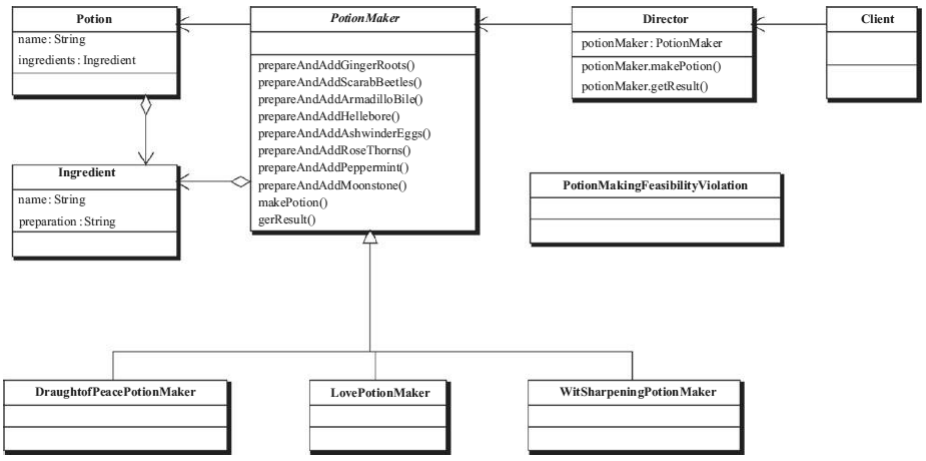
más példa:
java.text.NumberFormat – getInstance()
java.text.NumberFormat – getInstance(Locale inLocale)

## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

**Builder tervezési minta:**
Szintén az a cél, hogy egy építési folyamattal több, különböző szerkezetű elemet lehessen létrehozni.
Akkor használjuk, ha lépésről lépésre kell létrehozni az elemet.
Gyakran Factory mintával kezdődik a tervezés, fejlesztés, de ha túl sok lépésből áll, akkor áttérünk Builder-re.
Akkor alkalmazzuk, ha a létrehozási folyamatnak (vagyis a létrehozás algoritmusának) függetlennek kell lennie a szerkezettől. Ha egy osztály sok rész-osztályt használ (komplex osztály), mindenképp használjuk ezt, mert a részek változásakor változtatni kell a létrehozó kódokat is.
Új részek felvételét is kezeli, hiszen a használó (kliens) nem változik.

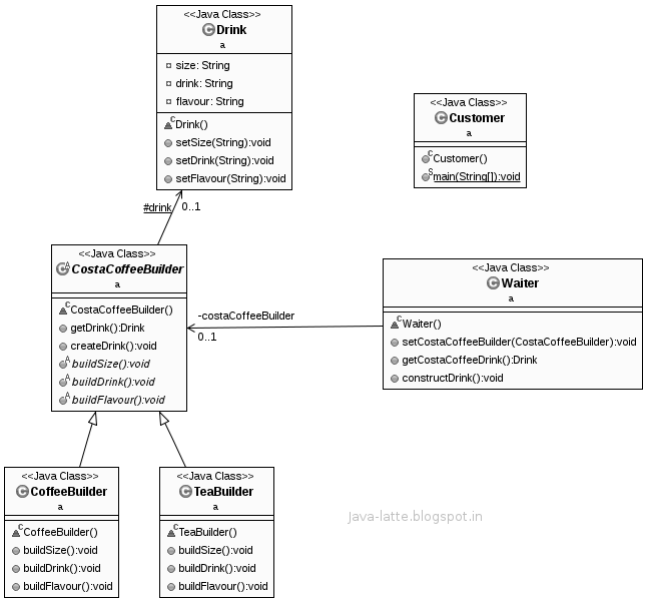
## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

1. példa



## LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

2. példa




## BUILDER vs FACTORY METHOD MINTA

A Factory egyetlen hívással adja át a paramétereket, és lényegileg egyetlen lépésben kapja meg az eredmény objektumot.

A Builder több egymás utáni lépést tesz lehetővé, és setterekkel lehet felépíteni a saját paraméterlistát.

Hétköznapi példa: étterem – a „nap főztje”, illetve pl. pizza-rendelés különböző összetevők alapján.

Forrás pl.:
https://myjavalatte.wordpress.com/tag/builder-pattern-vs-factory-pattern/

http://java-latte.blogspot.hu/2014/10/builder-design-pattern-vs-factory-pattern-example-in-java.html

## TERVEZÉSI MINTÁK – ENYHE RÉSZLETEZÉS

**Szerkezeti tervezési minták:**
A szerkezeti minták azt mutatják meg, hogy hogyan használjuk a gyakorlatban az objektum összetételt, hogy az igényeinknek megfelelő objektum szerkezetek létrejöhessenek futási időben.
Az objektum összetétel három típusa:

1. Aggregáció: az összetételben szereplő objektum nem kizárólagos tulajdona az őt tartalmazó objektumnak,

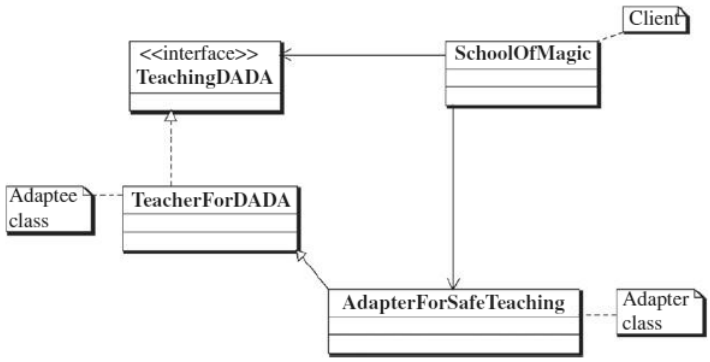
2. Kompozíció: amikor kizárólagos tulajdona,

3. Átlátszó csomagolás (wrapping): amikor a tulajdonos átlátszó.

## SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS

**Illesztő – Adapter**
Az illesztő (angolul: adapter) tervezési minta arra szolgál, hogy egy meglévő osztály felületét hozzá igazítsuk saját elvárásainkhoz. Leggyakoribb példa, hogy egy régebben megírt osztályt akarunk újrahasznosítani úgy, hogy beillesztjük egy osztály hierarchiába. Mivel ehhez hozzá kell igazítani az ős által előírt felülethez, ezért illesztő mintát kell használnunk.
A régi osztályt ilyen estben gyakran illesztendőnek (adaptee) hívjuk. Az illesztő és az illesztendő között általában kompozíció van, azaz az illesztő kizárólagosan birtokolja az illesztendőt. Ezt gyakran úgy is mondjuk, hogy az illesztő becsomagolja az illesztendőt.

## SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS



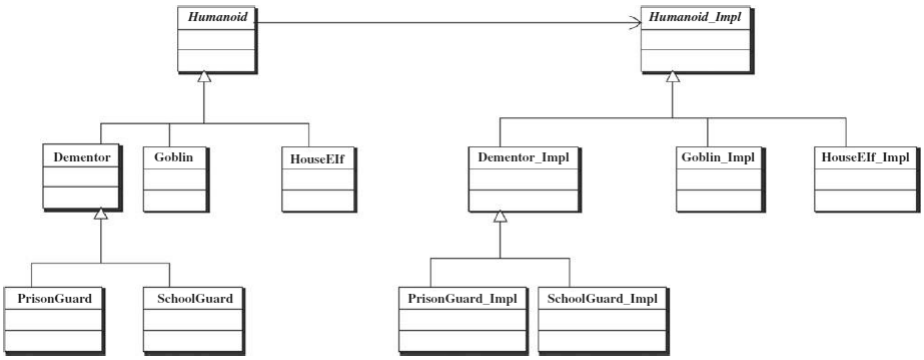

## SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS

**Híd – Bridge**

Cél: az elvont ábrázolást elválasztani a megvalósítástól, hogy azok egymástól függetlenül is módosíthatóak legyenek.

Olyan estekben, amikor egy alkalmazást több felületen szeretnénk megvalósítani, az öröklés használata nem mindig szerencsés, mert maradandó kötést hoz létre. Ilyenkor célszerű a Híd (Bridge) minta használata. Ha például egy ablakkezelőt több felületen szeretnénk megvalósítani, akkor az ablaktípusokat nem szükséges megírni minden felületre. Elég a Híd (Bridge) minta használata, amivel megvalósítunk minden felületre egy alosztályt. Az ablaktípusok pedig a Híd (Bridge) felület-függvényeit használják.

## SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS




http://themananvyas.blogspot.hu/2012/09/adapter-vs-bridge.html

## SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS

**Összetétel tervezési minta:**
Az összetétel minta az írja le, hogy az objektumok egy csoportját ugyanúgy kell kezelni, mint egy adott objektum példányait külön-külön. Az összetétel itt arra utal, hogy fa struktúrába szervezzünk objektumokat így reprezentálva a rész-egész hierarchiákat. Az összetétel minta lehetővé teszi, hogy a kliensek az önálló objektumokat és összetételeket egységes módon kezeljék.

## TERVEZÉSI MINTÁK – ENYHE RÉSZLETEZÉS

**Viselkedési tervezési minták**
A viselkedési minták az osztályok és az objektumok közötti kommunikációt írják le. A középpontban az algoritmusok megvalósítása és a felelősségi körök elosztása (kommunikáció) áll. Segítenek abban, hogy a kapcsolatokra helyezzük a hangsúlyt, ahelyett hogy a vezérlésre kellene figyelnünk. Vannak osztály minták és objektum minták. Az osztályminták öröklődéssel rendelik az osztályokhoz a szükséges viselkedést. Az objektum minták meghatározzák a viselkedés és objektum kompozíciót, azaz hogyan működjenek együtt társobjektumok egy csoportja a több objektumot igénylő műveleteknél. A viselkedési objektumminták öröklés helyett összetételt alkalmaznak.

## NÉHÁNY IRODALOM


https://www.tutorialspoint.com/design_pattern/index.htm

http://www.tankonyvtar.hu/hu/tartalom/tamop425/0038_informatika_Projektlabor/ch01s04.html

http://java-latte.blogspot.hu/2014/02/factory-method-design-pattern-in-java.html

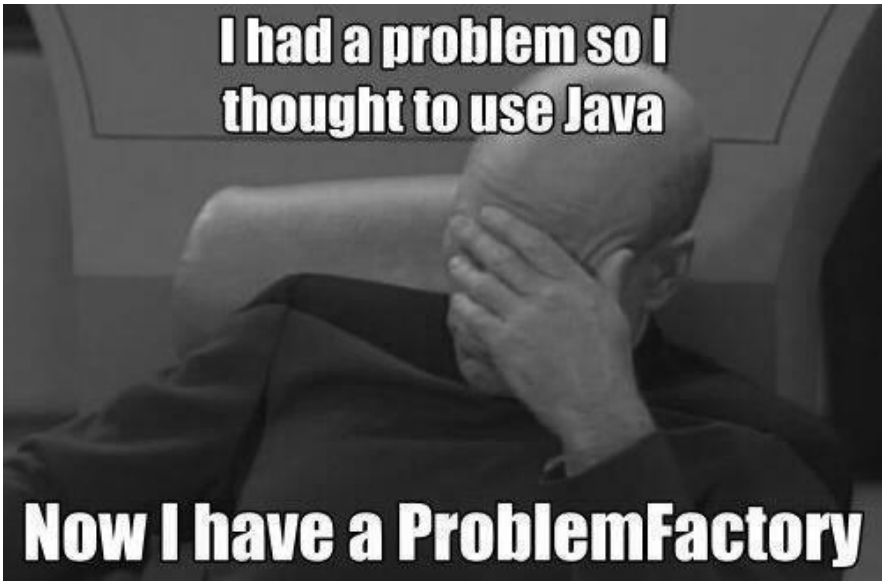
https://myjavalatte.wordpress.com/tag/builder-pattern-vs-factory-pattern/

http://java-latte.blogspot.hu/2014/10/builder-design-pattern-vs-factory-pattern-example-in-java.html

http://khaledsmulti.blogspot.hu/2013/09/adapter-vs-bridge-pattern-with-real.html

http://themananvyas.blogspot.hu/2012/09/adapter-vs-bridge.html

Google ☺


I had a problem so I
thought to use Java
Now I have a ProblemFactory