Programozás III

KOMPOZÍCIÓ,
TERVEZÉSI MINTÁK

és EGYEBEK

OBJEKTUM KOMPOZÍCIÓ

Az objektumorientált programozás magasabb lépcsőfoka, melynek lényege, hogy olyan osztályokat hozunk létre, melyek más osztályokkal paraméterezhetőek.
Így olyan építőelemeket tudunk kialakítani, amelyeket jól kombinálhatunk egymással, és eltérő variációikkal eltérő működést érhetünk el.

Pl:

```
public class Vonat(){
    private Varos induloAllomas, celAllomas;
    public Vonat(Varos idulo, Varos cel){
        this.induloAllomas = indulo; …}
…. }
```

KIINDULÓ PÉLDA

Vannak **vonat**ok. Minden vonatot jellemez egy egyedi azonosító, a kezdőállomása, végállomása, indulási ideje. Az állomások a nevükkel és koordinátáikkal adhatók meg.

A vonat konstruktorának paraméterei:

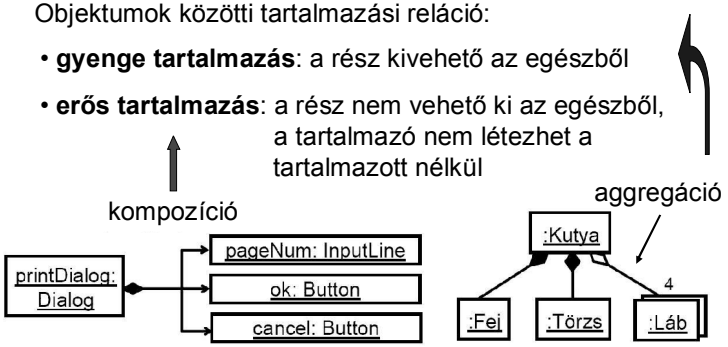
Az indulási időn kívül két **Varos** típusú objektum, ahol Varos egy olyan osztály, amely leírja egy város tulajdonságait, pl. a város nevét, földrajzi koordinátáit.

Vagyis NEM string varosNev, int x, int y !

KOMPOZÍCIÓ ÁLTALÁBAN

Objektumok közötti tartalmazási reláció:

- **gyenge tartalmazás**: a rész kivehető az egészből
- **erős tartalmazás**: a rész nem vehető ki az egészből, a tartalmazó nem létezhet a tartalmazott nélkül



Kompozíció: nem távolítható el a benne lévő obj., mert különben a külső is megsemmisülne.

KOMPOZÍCIÓ vs AGGREGÁCIÓ

Composition

```
final class Car {

  private final Engine engine;

  Car(EngineSpecs specs) {
    engine = new Engine(specs);
  }

  void move() {
    engine.work();
  }
}
```

Aggregation

```
final class Car {

  private Engine engine;

  void setEngine(Engine engine) {
    this.engine = engine;
  }

  void move() {
    if (engine != null)
      engine.work();
  }
}
```

Forrás: http://stackoverflow.com/questions/11881552/implementation-difference-between-aggregation-and-composition-in-java
http://javarevisited.blogspot.hu/2014/02/ifference-between-association-vs-composition-vs-aggregation.html

KOMPOZÍCIÓ vs ÖRÖKLÉS

```
class GyorsuloMozgas {
   private int gyorsulas, sebesseg;
   private String haladasiIrany;
   private Mozgas m ;
   public GyorsuloMozgas(int gyorsulas, int sebesseg, String haladasiIrany) {
      this.gyorsulas = gyorsulas;
      this.sebesseg = sebesseg;
      this.haladasiIrany = haladasiIrany;
      m = new Mozgas(haladasiIrany,sebesseg);
   }
   public String irany() {
       return m.irany().haladasiIrany;
    }
…
}
```

delegálás (továbbítás) – delegált metódus

KOMPOZÍCIÓ vs ÖRÖKLÉS

```
class GyorsuloMozgas extends Mozgas{
   private int gyorsulas;
   public GyorsuloMozgas(String haladasiIrany, int sebesseg) {
      super(haladasiIrany, sebesseg);
   }
…
}
```

A meglévő osztály bővítése helyett adjunk az új osztályhoz egy privát mezőt, amely a meglévő osztály egy példányára hivatkozik. Ez kompozíció, mert a meglévő osztály az új osztály egy komponensévé válik.

KOMPOZÍCIÓ vs ÖRÖKLÉS

Úgy tűnik, hogy a kompozíció helyettesítheti az öröklést, de mégsem az a fő kérdés, hogy

kompozíció vs öröklődés,

hanem az, hogy

egy adattag primitív típus-e, vagy objektum

Pl.:

```
class Jarmu{
    private Motor m;
    public Jarmu(Motor m){
        this.m = m;
    }
}
```

KOMPOZÍCIÓ vs ÖRÖKLÉS

Javaslatok:

Csak akkor használjunk öröklődést, ha egészen biztosak vagyunk benne, hogy az alkalmazás egész életciklusában „is a” (ez egy…) kapcsolat van, és ez meg is marad a két osztály között.

Pl: személy – alkalmazott

az alkalmazott egy szerep, amit a személy játszik;

mi van, ha munkanélkülivé válik?

mi van, ha egyszerre főnök is és alkalmazott is?

MÉG EGY PÉLDA

Vannak hengerek, rudak, csövek – definiáljuk az osztályokat kompozíció segítségével, és teszteljük egy-egy példánnyal.

Emlékeztető: örökléssel

```
public interface AlakzatInterface {
    public double terfogat();
    public double tomeg();
    public double getSugar();
    public double getMagassag();
    public double getSuruseg();
}

public abstract class AbsztraktAlakzat
        implements AlakzatInterface {

public class Henger
        extends AbsztraktAlakzat{
```

```
public class Rud extends Henger {

    private double suruseg;

    public Rud(double sugar, double magassag,
                                double suruseg) {
        super(sugar, magassag);
        this.suruseg = suruseg;
    }

    @Override
    public double tomeg() {
        return this.terfogat() * suruseg;
    }
```

KOMPOZÍCIÓ vs ÖRÖKLÉS

Javaslatok:

Ne használjunk öröklést, ha kizárólag a kódismétlés elkerülése vagy a polimorfizmus a cél, és nincs tényleges „is a” kapcsolat az osztályok között.

1. esetben sima kompozíció

2. esetben interfészekkel megtámogatott kompozíció

De mindez csak a „vájt fülűekre” vonatkozik, azokra, akik már ragyogóan tudják az öröklődést és tényleg különbséget tudnak tenni a kettő között!!

MÉG EGY PÉLDA

Kompozícióval:

```
public interface Henger {

    public double getSugar();
    public double getMagassag();
    public double terfogat();
    public String alakzatNev();
}

public interface Rud
        extends Henger{

    public double getSuruseg();
     public double tomeg();
}
```

```
public class RudImpl implements Rud {

    private Henger henger;
    private double suruseg;

    public RudImpl(double sugar, double magassag,
                                double suruseg) {
        henger = new HengerImpl(sugar, magassag);
        this.suruseg = suruseg;
    }

    @Override
    public double tomeg() {
        return henger.terfogat() * suruseg;
    }
```

ld.: ea_mintapeldak\HengerCsoKompozicioval

NÉHÁNY IRODALOM

http://www.artima.com/designtechniques/compoinh3.html

http://www.javaworld.com/javaworld/jw-11-1998/jw-11-techniques.html

http://www.javaworld.com/javaworld/jw-06-2001/jw-0608-java101.html

http://www.javaworld.com/javaworld/jw-12-1998/jw-12-techniques.html?page=3

Google ☺

NEM IDE TARTOZIK, DE FONTOS

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

// http://stackoverflow.com/questions/16295329/email-address-validation-regex
private static final String EMAIL_PATTERN =
"^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";
private static final Pattern PATTERN = Pattern.compile(EMAIL_PATTERN);

private boolean checkMail() {
    if (this.email == null) {
        return false;
    }
    Matcher matcher = PATTERN.matcher(this.email);
    return matcher.matches();
}
```

NEM IDE TARTOZIK, DE FONTOS

„Klinikás” példa vélemény: a névbe írhatunk számot is.

Kérdés: hogy lehetne kiküszöbölni?

Reguláris kifejezések használatával.

ÚJABB PÉLDA

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);
    }
}
```

FONTOS ÉSZREVÉTEL

~~if(sportag == 1) {~~
~~név bekérése;~~
~~szül_év bekérése;~~
~~szem_szám bekérése;~~
~~az ugró létrehozása;~~
~~}~~
~~if(sportag == 2) {~~
~~név bekérése;~~
~~szül_év bekérése;~~
~~szem_szám bekérése;~~
~~az focista létrehozása;~~
~~}~~

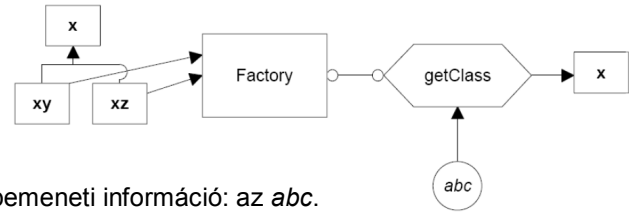
~~if(sportag == 3) {~~
~~név bekérése;~~
~~szül_év bekérése;~~
~~szem_szám bekérése;~~
~~az futó létrehozása~~
~~}~~

Egy jó programban NINCS kódismétlés!!!

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.

ELEGÁNS MEGOLDÁS

Még kevésbé „fapadosan”

Javasolt felépítés:

- interface (esetleg extends Comparable)
- az interface-t implementáló abstract osztály
- az abstract osztályt kiterjesztő további osztályok

TERVEZÉSI MINTÁK CSOPORTOSÍTÁSA

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

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 program felületet nyújtó öröklési hierarchiát kialakítani;

az objektumok összeillesztésének módját meghatározni.

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

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

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

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 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 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

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

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.

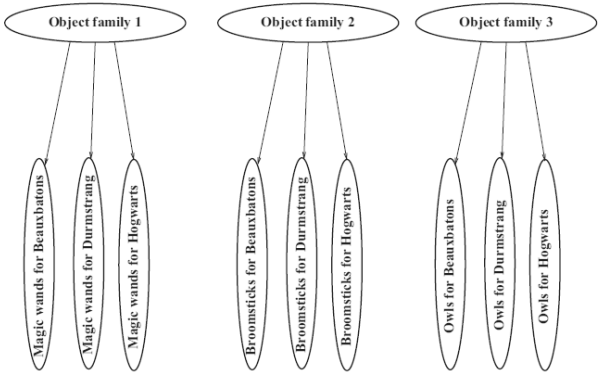
LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

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

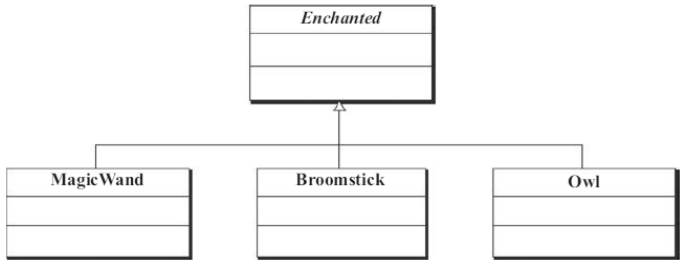
LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

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



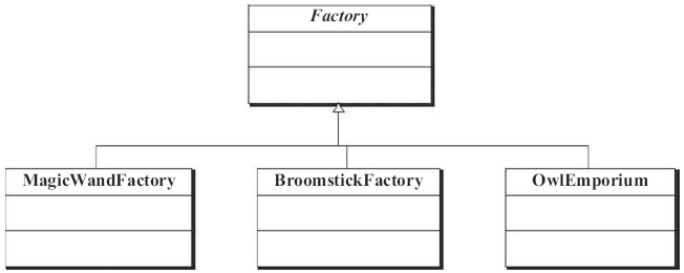
LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

A gyártott példányok mindegyike valamilyen varázslásra (bűbájra) szolgál:



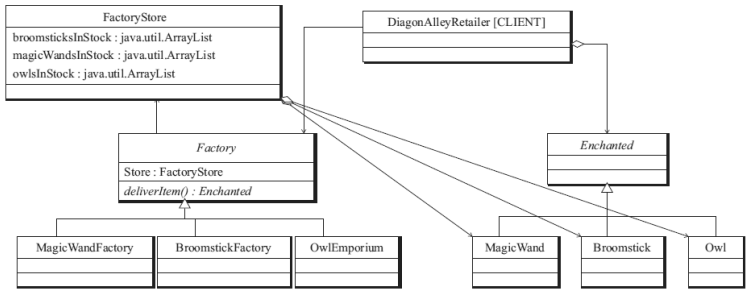
LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

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



LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

A két osztályhierarchiát a Diagon Alley kereskedője fogja össze:

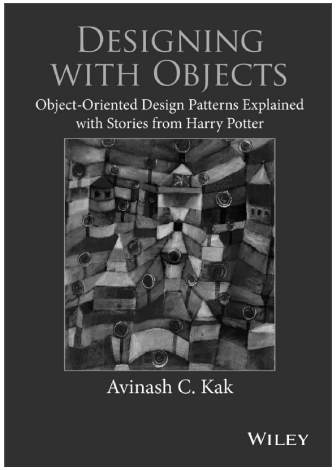


LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

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

Letölthető kódok:

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



LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

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

4. példa – Java standard library:

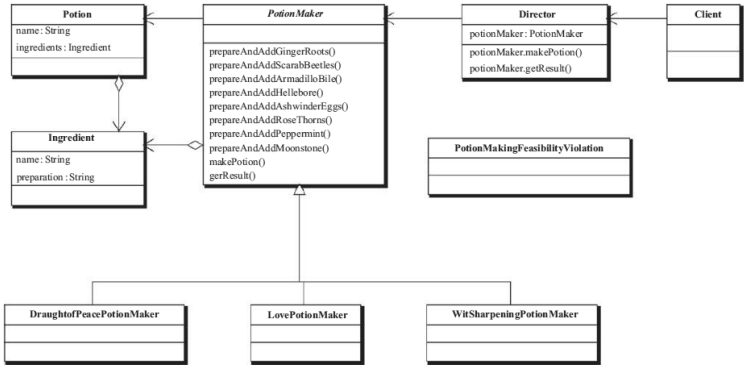
Paraméterezéstől függően a Calendar osztály getInstance() metódusa más-más példányt ad vissza:

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)

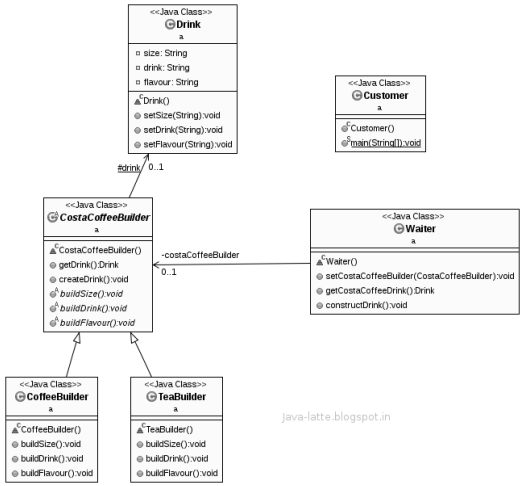
LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

1. példa



LÉTREHOZÁSI MINTÁK – ENYHE RÉSZLETEZÉS

2. példa



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.

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.

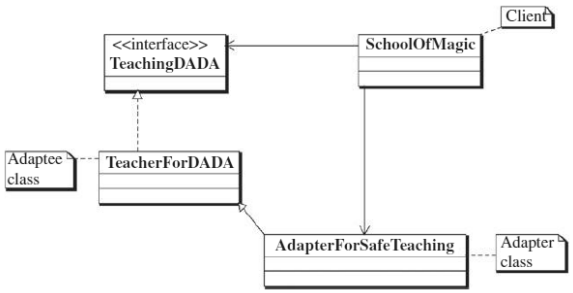
Forrás pl.:
https://myjavalatte.wordpress.com/tag/builder-pattern-vs-factory-pattern/

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

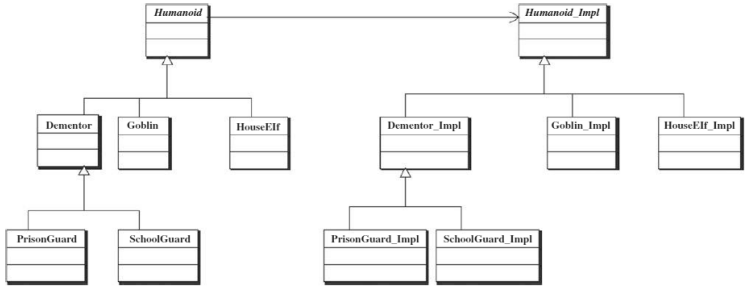
SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS

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



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

SZERKEZETI MINTÁK – ENYHE RÉSZLETEZÉS

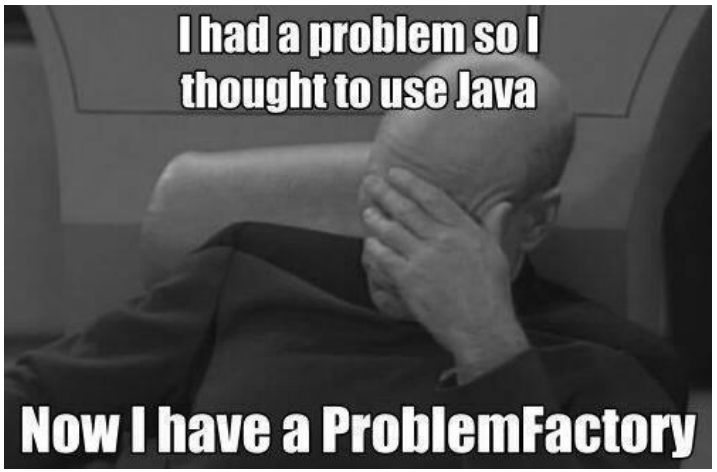
Híd – Bridge
Cél: az elvont ábrázolást elválasztani a megvalósítástól, hogy azok egymástól függetlenül is módosíthatóak legyenek.
Olyan estekben, amikor egy alkalmazást több felületen szeretnénk megvalósítani, az öröklés használata nem mindig szerencsés, mert maradandó kötést hoz létre. Ilyenkor célszerű 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

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 ☺