Tervezési Minták

(Design Patterns)

Mik a tervezési minták?

- Nem kész kód vagy algoritmus
- Általános megoldások gyakran felmerülő szoftvertervezési problémákra objektum-orientált rendszerekben.
- Mindegyik minta specifikálja:
  - a megoldandó problémát,
  - a megoldást,
  - az alkalmazásának körülményeit, valamint
  - az alkalmazásának következményeit.

Kell ez nekem?

- Lehet évekig programozni anélkül, hogy ezekről tanulnál.
- Kétségtelenül megvalósítasz belőlük majd magad is anélkül, hogy tudatosan alkalmaznád őket.
- Ellenben:
  - Ismeretükkel lesz egy eszköztárad kipróbált megoldásokból.
  - Közös nyelved lesz amit a munkatársaiddal használhatsz, amikor szoftvertervezési feladatokat vitattok meg.

Minták csoportjai

- Létrehozási (“creational”), amelyek új objektumok létrehozására alkalmas minták.

- Szerkezeti (“structural”), amelyek minták objektumok összeillesztésére.

- Viselkedési (“behavioral”), amelyek algoritmusokat és vezérlést írnak le, vagy leírják objektumcsoportok együttműködését.

Minták alkalmazása

- A minták különböző bonyolultságúak, és majd látni fogjuk, hogy bizonyos problémák többféleképpen is megoldhatók.

- Forgalmi probléma: biztonságossá kell tennünk egy útkereszteződést.

Minták alkalmazása

- Megoldások:
  - Stop táblák
  - Lámpák
  - Körforgalom
  - Felüljáró
  - Vagy…

Minták alkalmazása

Létrehozási minták

(“Creational patterns”)

Létrehozási minták

- Olyanok, amik új objektumok létrehozására szolgálnak.

- A konstruktor fogalmának különböző fokú általánosításai.

Singleton

Singleton: mi az?

- Biztosítjuk, hogy egy osztályból csak egy példány létezzen.

- Biztosítunk globális elérési pontot ehhez az osztályhoz.

Singleton: miért?

- Miért akarnánk egy osztályból csak egy példányt?
  - Mert olyasmit képvisel, amiből egy van az alkalmazás kontextusában.
  - Az operációs rendszer egésze jellemzően singleton.
  - Ha az alkalmazás valamiből egy valamit használ (pl. pontosan egy adatbázist), az is lehet singleton.

Singleton: hogyan?

- Privát konstruktor megakadályozza további példányok létrehozását.
- Statikus metódus hozzáférést biztosít.

```
public class Database {
   private static final Database instance =
     new Database();

   public static Database getInstance() {
     return instance;
   }

   private Database() {
     …
   }
}
```

Singleton: előnyök

- Egyben tud tartani egy aspektushoz tartozó kódot anélkül, hogy az szét legyen szóródva a rendszerben.

- A globális elérés miatt a kliens kód bárhonnan elérheti, nem kell passzolni a példányt.

-

Singleton: hátrányok

- A globális elérés miatt a kliens kód bárhonnan elérheti (ez az előbb még előny volt, ugye?), ezért az összefüggések kevésbé felfedezhetők.

- Minden alrendszer ugyanannak a példánynak az állapotát módosítja, ez nem várt kölcsönhatásokat válthat ki. Ez gyengébb modularizáltságot eredményez.

- Nehezíti a tesztelést, ha nem lehet behelyettesíteni egy tesztobjektumot a singleton helyére.

Factory Method

Factory Method: mi ez?

- A konstruktor általánosítása egy osztályon belül.
- Van egy alaposztályunk, aminek vannak alosztályai. Az osztály objektumai további objektumokat hoznak létre. Ezen objektumok osztálya viszont függ a létrehozó alosztályától.

Factory Method: hogyan?

```
abstract class Logistics {
  public DeliveryPlan makeDeliveryPlan(…) {
    Transport t = createTransport();
    …
    return new DeliveryPlan(…, t);
  }

  abstract Transport createTransport();
}
```

```
interface Transport {
  public void deliver();
}
```

Factory Method: hogyan?

```
class GroundLogistics extends Logistics {
  Transport createTransport() {
    return new Truck();
  }
}
```

```
class Truck implements Transport {
  public void deliver() {
    …
  }
}
```

Factory Method: hogyan?

```
class AirLogistics extends Logistics {
  Transport createTransport() {
    return new Airplane();
  }
}
```

```
class Airplane implements Transport {
  public void deliver() {
    …
  }
}
```

Factory Method: miért?

- Megkönnyíti platformfüggetlen kód kialakítását.
- A kód zöme az alaposztályban van, és nem tud a konkrét megvalósításokról.
- A platformspecifikus részek a factory method-okban vannak elkülönítve. Ezeket a platformspecifikus alosztályok valósítják meg.
- “Platform” helyett lehet bármilyen más kontextus amitől függetleníteni akarjuk a kód nagy részét.

Factory Method: miért?

- A factory method akár újra is hasznosíthat objektumpéldányokat ha azokat költséges létrehozni.

- *Egy felelősség elve*: a függőségek létrehozása egy helyre van koncentrálva, az alosztályoknak csak ezzel kell foglalkozniuk.

- *Nyílt-Zárt Elv*: jól meghatározott bővítési ponttal új specifikus változatok hozhatók létre az alaposztályból anélkül, hogy akár az alaposztályt, akár annak klienseit változtatni kellene.

Abstract Factory

Abstract Factory: mi ez?

- Általánosítása a Factory Method-nak

- Jellemzően, ha egy osztályban már több Factory Method-ra van szükség, akkor ezeket egy interfészbe gyűjtjük.

Abstract Factory: hogyan?

```
abstract class UI {
  public Dialog makeDialog(String msg, String[] buttonLabels) {
    UIText t = createText(msg);
    UIButton[] bs = buttonLabels.map { l ⇒ createButton(l) }
    Dialog = new Dialog(t, bs)
  }

  abstract UIText createText(String label);
  abstract UIButton createButton(String label);
}
```

Abstract Factory: hogyan?

- Ezt is csinálhatnánk factory method-okkal:

```
class WindowsUI extends UI {
  UIText createText(String label) { return new WinText(label); }
  UIButton createButton(String label); {
    return new WinButton(label);
}
```

```
class MacUI extends UI {
  UIText createText(String label) { return new MacText(label); }
  UIButton createButton(String label); {
    return new MacButton(label);
}
```

Abstract Factory: hogyan?

- De ki is emelhetjük ezeket a metódusokat egy Abstract Factory interfészbe:

```
interface UIToolkit {
  abstract UIText createText(String label);
  abstract UIButton createButton(String label);
}
```

```
class WindowsUIToolkit implements UIToolkit {
  UIText createText(String label) { return new WinText(label); }
  UIButton createButton(String label); {
    return new WinButton(label);
}
```

```
class MacOSUIToolkit implements UIToolkit {
  UIText createText(String label) { return new MacText(label); }
  UIButton createButton(String label); {
    return new MacButton(label);
}
```

Abstract Factory: hogyan?

- Ezután:

```
class UI {
  private UIToolkit toolkit;

  public Dialog makeDialog(String msg, String[] buttonLabels) {
    UIText t = toolkit.createText(msg);
    UIButton[] bs = buttonLabels.map { l ⇒
      toolkit.createButton(l) }
    Dialog = new Dialog(t, bs)
  }
}
```

Abstract Factory: hogyan?

- Sokszor jól megy együtt Singletonnal:

```
class UI {
 public Dialog makeDialog(String msg, String[] buttonLabels) {
    UIToolkit toolkit = UIToolkit.getInstance();
    UIText t = toolkit.createText(msg);
    UIButton[] bs = buttonLabels.map { l ⇒
      toolkit.createButton(l) }
    Dialog = new Dialog(t, bs)
  }
}
```

Abstract Factory: hogyan?

- Sokszor jól megy együtt Singletonnal:

```
abstract class UIToolkit {
  private static final instance;
  static {
    if (getOS().equals(“Windows”)) {
      instance = new WindowsUIToolkit();
    } else if (getOS().equals(“MacOS”)) {
      instance = new MacOSUIToolkit();
    }
  }

  public static UIToolkit getInstance() {
    return instance;
  }

  abstract UIText createText(String label);
  abstract UIButton createButton(String label);
}
```

Abstract Factory: példák?

- JDBC driver
- UI toolkits
- MVC keretrendszerekben különböző View technológiák

Abstract Factory: előnyök

- Kliens kód független a konkrét legyártott példányoktól.
- A factory-t jellemzően magasabb szint (alkalmazás, platform) inicializálja a kliens számára.
- A legyártott példányok egymással kompatibilisek.

Builder

Builder: mi ez?

- Megoldandó problémák:
  - Komplex objektumok lépésenkénti létrehozása, vagy
    - komplex objektumok külső leírásból való létrehozása.
- Az Abstract Factory egymással rokon objektumcsoportok létrehozását teszi lehetővé, jellemzően minden objektumot egyetlen lépéssel. Ezzel szemben a Builder egy objektumot (ami lehet összetett) hoz létre több lépésben.

Builder: hogyan?

```
class Rectangle {
  private Point coordinates;
  private Extent size;
  private Stroke border;
  private Fill fill;

  Rectangle(Point coordinates, Extent size, Stroke border, Fill fill) {
    this.coordinates = coordinates;
    this.size = size;
    this.border = border;
    this.fill = fill;
  }
}
```

Builder: hogyan?

```
class Rectangle {
  private Point coordinates;
  private Extent size;
  private Stroke border;
  private Fill fill;

  Rectangle(Point coordinates, Extent size, Stroke border, Fill fill) {
    this.coordinates = coordinates;
    this.size = size;
    this.border = border;
    this.fill = fill;
  }
}
```

```
Rectangle r = new Rectangle(new Point(12, 55), new Extent(40, 45),
  Stroke.THIN_BLACK, Fill.TRANSPARENT);
```

Builder: hogyan?

```
Rectangle r = new Rectangle(new Coordinates(12, 55), new Extent(40, 45),
  Stroke.THIN_BLACK, Fill.TRANSPARENT);
```

- Mi ezzel a baj?
  - Sok paramétere van a konstruktornak. Nem egyértelmű, hogy melyik mire való.
  - Minden paramétert meg kell adni, még azokat is amikre van valami ésszerű alapértelmezett érték. Pl. lehetne a THIN_BLACK keret és TRANSPARENT kitöltés alap.
  - Ezeket el lehet kerülni azzal, hogy sok konstruktort adunk meg, de az is egy idő után kezelhetetlen.

Builder: hogyan?

```
 Rectangle(Point coordinates, Extent size) {
   this(coordinates, size, Stroke.THIN_BLACK, Fill.TRANSPARENT);
}

Rectangle(Point coordinates, Extent size, Stroke border) {
   this(coordinates, size, border, Fill.TRANSPARENT);
}

Rectangle(Point coordinates, Extent size, Fill fill) {
   this(coordinates, size, Stroke.THIN_BLACK, fill);
}
```

Builder: hogyan?

- Ezzel szemben:

```
public class RectangleBuilder {
    private Point coordinates;
    private Extent size;
    private Stroke border = Stroke.THIN_BLACK;
    private Fill fill = Fill.CLEAR;

    public RectangleBuilder withCoordinates(Point coordinates) {
        this.coordinates = coordinates;
        return this;
    }

    public RectangleBuilder withSize(Extent size) {
        this.size = size;
        return this;
    }

    public RectangleBuilder withBorder(Stroke border) {
        this.border = border;
        return this;
    }

    public RectangleBuilder withFill(Fill fill) {
        this.fill = fill;
        return this;
    }

    public Rectangle build() {
        return new Rectangle(coordinates, size, border, fill);
    }
}
```

```
Rectangle r = new RectangleBuilder()
  .withCoordinates(new Point(12, 55))
  .withSize(new Extent(40, 45))
  .withBorder(Stroke.THIN_RED)
  .withFill(Fill.BLACK)
  .build();
```

```
Rectangle r = new RectangleBuilder()
  .withCoordinates(new Point(12, 55))
  .withSize(new Extent(40, 45))
  .build();
```

Az, hogy a metódusok “this”-t adnak vissza lehetővé teszi, hogy a hívásokat egymásra láncoljuk. Ez is egy mini-pattern, amit “folyékony interfésznek” hívunk (fluent interface), lásd:

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

Builder: hogyan?

- Az is Builder, ha közvetlen metódushívások helyett valamilyen külső specifikációból tudjuk felépíteni az objektumot:

```
<rectangle>
  <coordinates x=“52” y=“40”/>
  <size width=“50” height=“60”/>
  <border width=“0.1” color=“#000000ff”/>
  <fill color=“#ffffffff”/>
</rectangle>
```

Builder: hogyan?

- Ekkor a kód meghívja a builder-t és megadja neki a specifikációt:

```
ShapeBuilder b = new XMLShapeBuilder(new File(shapeSpecPath));
Shape shape = b.build();
```

- Az elv ugyanaz; az egyik esetben a kód ami meghívja a különböző “with” metódusokat a specifikáció, a másik esetben az XML/YAML/JSON/stb. fájl.

Builder: Director

- Van úgy, hogy a builder-t a kliens kód helyett egy másik objektum irányítja, a Director. A Director használja a Builder-t ahhoz, hogy egy meghatározott “recept” szerint állítsa össze az objektumot.
- Az előző példában pl. az “XMLShapeBuilder” tekinthető lenne egy Director-nak is, ami pl. olvassa az XML fájlt és annak tartalma alapján hívja a RectangleBuilder metódusait.

Builder: előnyök

- Elkerüli a “teleszkópos” konstruktorok problémáját.
- Java-ban sajnos nem adhatók nevek sem konstruktoroknak, sem metódus-paramétereknek.
- Lehetővé teszi, hogy az objektumok maguk ne legyenek módosíthatók (minden mező *final*), de a megalkotásuk mégis lépésenként történhet.
- Rekurzív builderekkel komponens objektumfák is felépíthetők.
- Újrahasznosítható konstrukciós kód.
- A konstrukciós kód különválasztható az üzleti logikától.

Prototype

Prototípus: mi az?

- Új objektum létrehozása meglévő objektum (a prototípus) másolásával anélkül, hogy tudnánk a pontos osztályát.

- Ha előre tudjuk az osztályt, akkor akár meg is hívhatjuk a konstruktort és paraméterként átadjuk a meglévő objektum mezőinek értéket. De mi van ha:

  - nem tudjuk pontosan az osztályt? (Az is lehet, hogy deklarált interfészt kaptunk csak az objektumhoz.)

  - nincs elérhető konstruktor, vagy olyan ami minden mezőt kellően inicializálna?

  - nem érhető el kívülről a meglévő objektum minden mezőjének állapota?

Prototípus: megoldás

- Az objektum osztálya deklarál egy klónozási metódust.
- Java-ban a java.lang.Object eleve deklarál egy “clone” metódust, ami binárisan lemásolja az objektum mezőit egy új objektumba.
  - Eközben nem történik konstruktor-hívás, vagyis az új objektum konstruktor nélkül jön létre!
  - A klón un. “sekély” klón: a hivatkozás típusú mezők értéke is átmásolódik, vagyis a klón és az eredeti ugyanazokra az objektumokra hivatkoznak.
  - Ha ezen módosítani kell, akkor felüldefiniálni kell a “clone” metódust.

Prototípus: mikor?

- Ha a kódnak nem kellene függnie a másolandó objektum konkrét osztályától.

- Ha sok különböző objektumot kell létrehozni, de azok közül sok hasonló konfigurációjú. Ekkor az alapkonfigurációk létrehozhatók objektumként, majd ezek klónozásával létrehozható a többi.

Prototípus: előnyök?

- Objektumok klónozhatók anélkül, hogy az osztályuktól függő kódot írnánk.

- Gyakran ismétlődő inicializációs kód (akár builder-rel létrehozott objektumok esetén) is elkerülhető azzal, hogy egy objektumot építünk fel, majd klónozzuk, esetleg tovább konfiguráljuk.

- Kiválthatók vele további alosztályok, ha azok csak a konfiguráció miatt léteznének.

Prototípus: variánsok?

- Prototípus + builder: a buildernek átadható egy prototípus kezdő konfigurációként.

- Paraméteres klónozás: a clone() metódusok paramétereket fogadnak és módosítják egy vagy több mezőjében a klónt. Ezek gyakran ugyanolyan nevű “with” metódusok, mint a builder-ben.

Szerkezeti minták

Adapter

Magyarul: átalakító

Adapter

- Probléma:
  - van egy objektumunk, ami megvalósít egy kívánt funkciót.
  - Van egy részrendszerünk, ami hasznosíthatná az objektumot, de más csatlakozást (interfészt) vár el, mint amit az objektum mutat.
    - Sem a részrendszert, sem az objektumot nem lehetséges/gazdaságos/kívánatos módosítani.
- Megoldás:
  - Létrehozunk egy objektumot, ami az alrendszer felé mutatja az elvárt interfészt, és továbbítja a kéréseket az objektumnak.
- Az adapter lehetővé teszi, hogy egymással nem kompatibilis osztályok együttműködjenek.

Adapter példa

```
interface Stack<T> {
  void push(T t);
  T pop();
  T peek();
}
```

```
interface Deque<T> {
  void addFirst(T t);
  void addLast(T t);
  void removeFirst(T t);
  void removeLast(T t);
  T getFirst();
  T getLast();
}
```

```
class StackAdapter<T> {
  Deque<T> q;

  void push(T t) {
    q.addFirst(t);
  }

  T pop() {
    q.removeFirst();
  }

  T peek() {
    q.getFirst();
  }
}
```

Adapter tulajdonságok

- Az adaptált objektumnak lehet olyan többlet funkcionalitása, amire az adapter nem képez le (pl. az összes “…Last” metódus

- A mi példánkban az adapter lényegében metódus-átnevezésekre szolgál, de a leképezés lehet bonyolultabb is.

- A mi példánkban az adapter egy interfészt (deque) képez le egy másik interfészbe (stack), ezáltal minden olyan objektumot adaptálni tud, amely megvalósítja a deque osztályt. Sok esetben az adapterek konkrét osztályokat is adaptálhatnak.

- Az adapternek saját identitása van, az adaptált objektum identitása kifelé nem látszik többé.

    - Ennek érdekes következménye, hogy az adaptált objektumra nem lehet például szinkronizálni.

Adapterek a Java API-ban

- java.io-ban: stream adapter byte tömbhöz (ByteArrayInputStream), olvasó adapter stream-hez (InputStreamReader)

- java.util: list adapter tömbökhöz, enumeration adapter kollekciókhoz, stb.

Adapter: előnyök

- Nem kell módosítani (vagy a másiktól függővé tenni) sem a kliens osztályt, sem a hasznos osztályt. Az illesztés szeparálva marad az adapterben.

Proxy


0000
0.0
V
00
Hz
RESET
FUNCTION
COST
UP
DOWN
CAT II
Operating voltage: 230V ac
Frequency display: 50Hz
Wide voltage range:230V----250V
Operating current: max 16A
Wattage display (watts):0W----3600W
Current display (amps):0.0A-16.0A
Price display range:0.00€/KWH----99.99€KWH
Total KWh and cost display:
0.000KWH----9999KWH,
0.00€----9999€
Intertek
PC
16
250

Proxy

- Probléma:
    - van egy objektumunk, amelyhez való hozzáférést szabályozni akarjuk.
        - Nem lehetséges/gazdaságos/kívánatos megkeresni *minden* olyan helyet, ahol a rendszer használja az objektumot és ott beiktatni a szabályozást. Az *objektumot magát* sem tudjuk/akarjuk módosítani.
- Megoldás:
    - Létrehozunk egy objektumot, ami a rendszer felé ugyanazt az interfészt mutatja, mint az objektum, és továbbítja a kéréseket az objektumnak.

Proxy vs. Adapter

- Elég hasonlónak tűnnek első látásra, pedig nem azok.
- Mindkettő “becsomagol” egy objektumot, de:
  - Az adapter más interfészt mutat kifelé mint az adaptált objektum. A proxy ugyanazt az interfészt mutatja.
  - Az adapter a kifelé mutató interfész funkcióit képezi le az objektum saját interfészére. A proxy szabályozza a hozzáférést az objektum már meglévő interfészéhez.

Proxy alkalmazások

- Hívásnaplózás
- Hozzáférés engedélyezése vagy megtagadása
- Funkcionalitás korlátozása (“nem zárható objektum” stb.)
- Távoli elérés biztosítása
- Költséges objektum szükség szerinti létrehozása
- Hivatkozás-számlálás

Proxy megvalósítása

- Közvetlenül megvalósítható az interfész, és minden hívás továbbít az eredeti objektum azonos hívásához.

- Használható futásközbeni kódgenerátor (pl. java.lang.reflect.Proxy)

- Olyan nyelvekben, mint a JavaScript, egy kis okoskodással bárki csinálhat általános proxy-generátort.

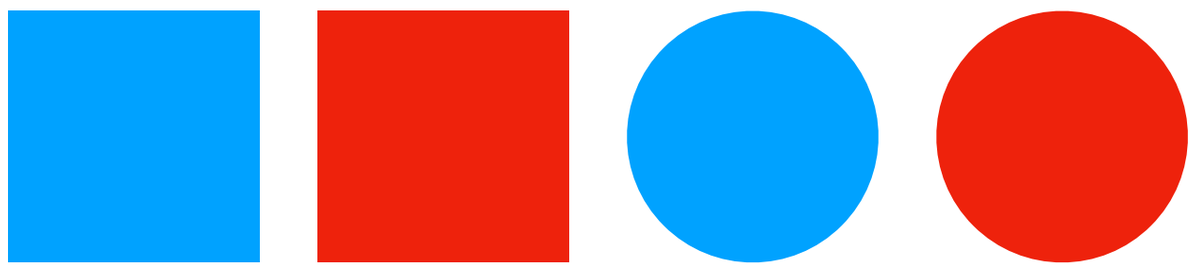
Bridge

Bridge: mi ez?

- Lehetővé teszi, hogy egy bonyolult osztályt (vagy osztály-hierarchiát kettéválaszzunk könyebben karbantartható részekbe.

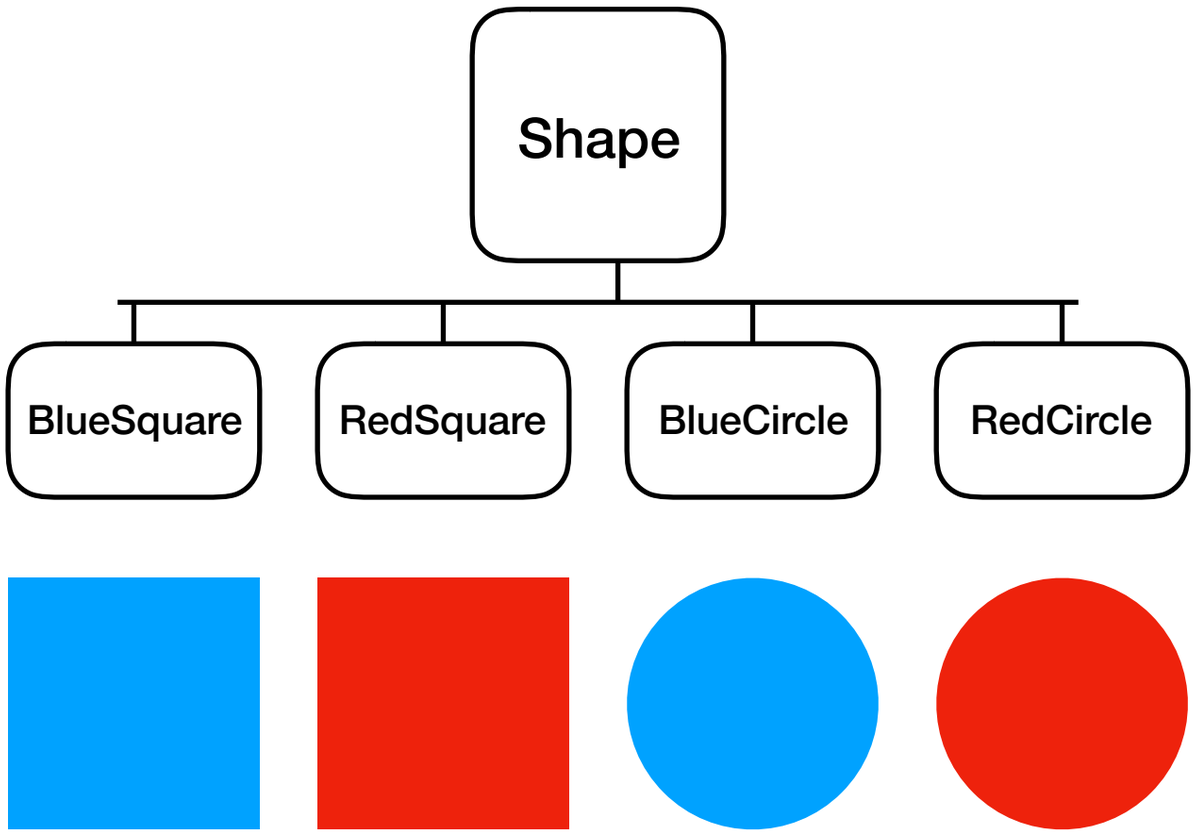
Bridge: mi ez?

- Nagyon mesterkélt példa: vannak alakzataink, amiket a formájuk meg a színük határoz meg:



Bridge: mi ez?

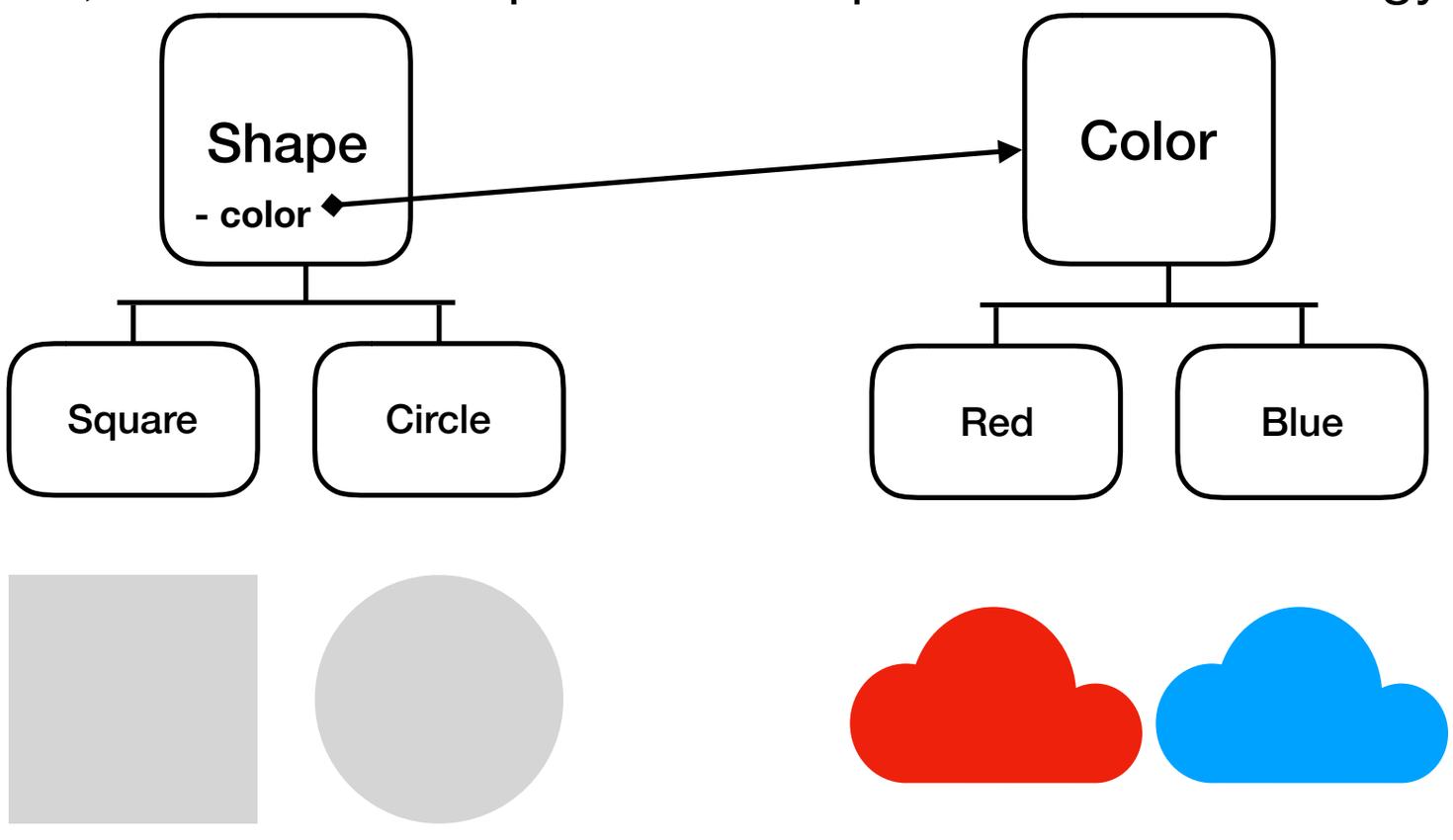
- Ha mind a forma mind a szín szerint csinálunk egy osztályt, akkor négy osztályunk lesz.
- Ha további akár formát, akár színt akarunk hozzáadni, az osztályok száma megsokszorozódik.



Bridge: mi ez?

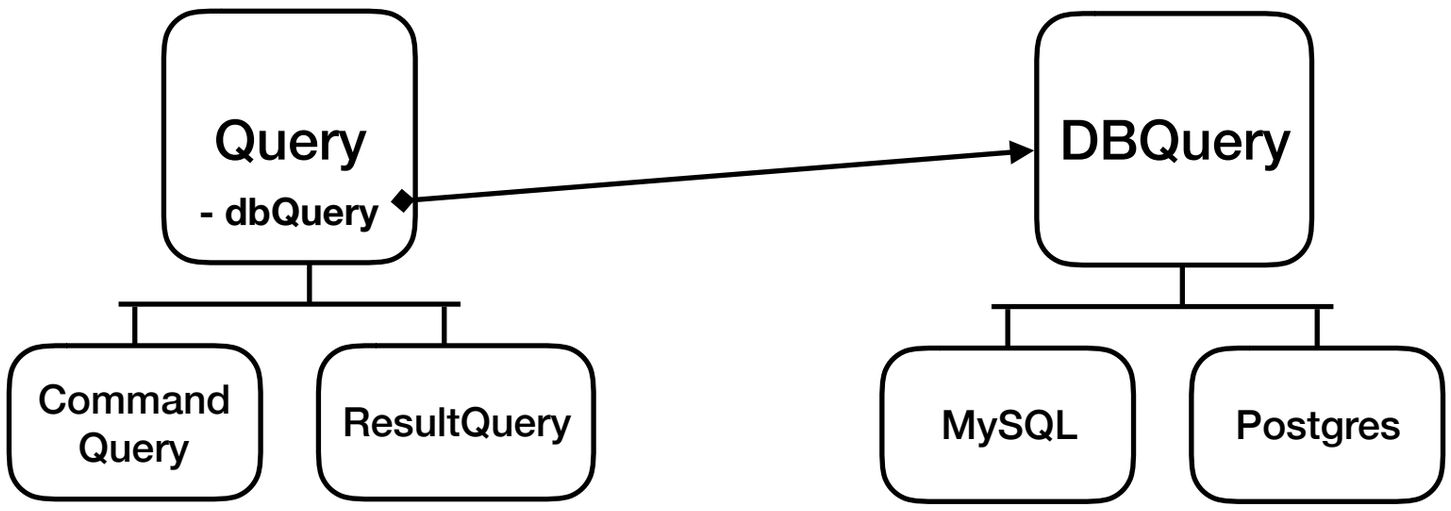
- Ha ehelyett szétbontjuk funkcionális dimenziók szerint, akkor sokkal egyszerűbb bővíteni a funkcionalitást.

- Ezen kívül, a különböző aspektusok szépen elválaszthatók egymástól.



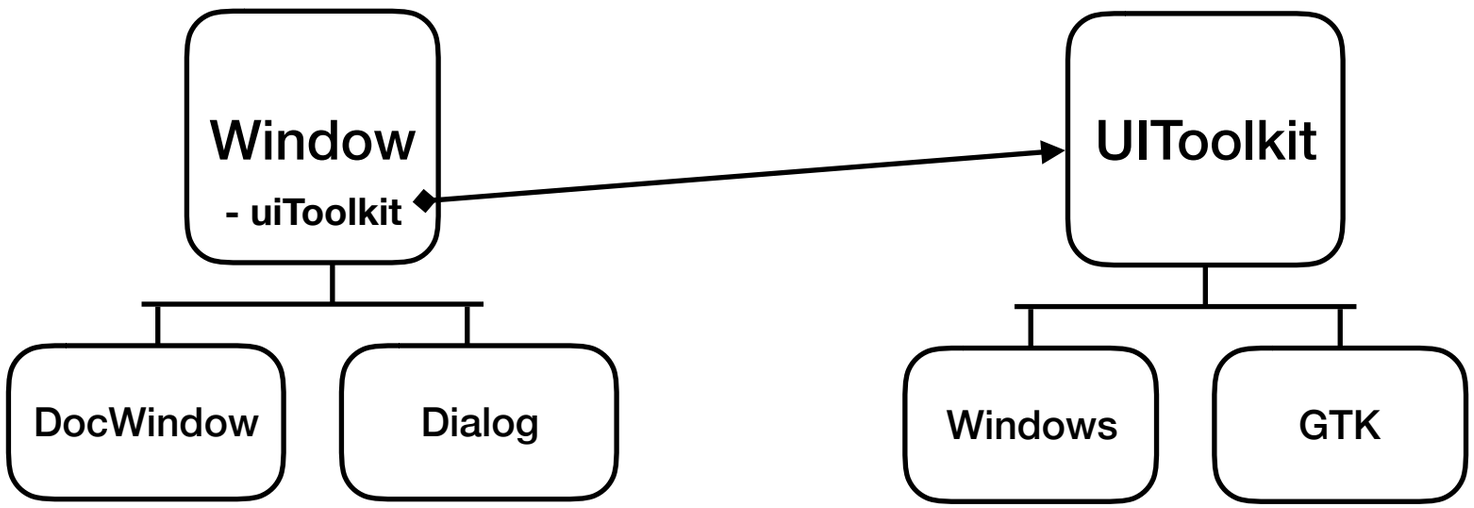
Bridge: mi ez?

- Kevésbé mesterkélt példákkal:



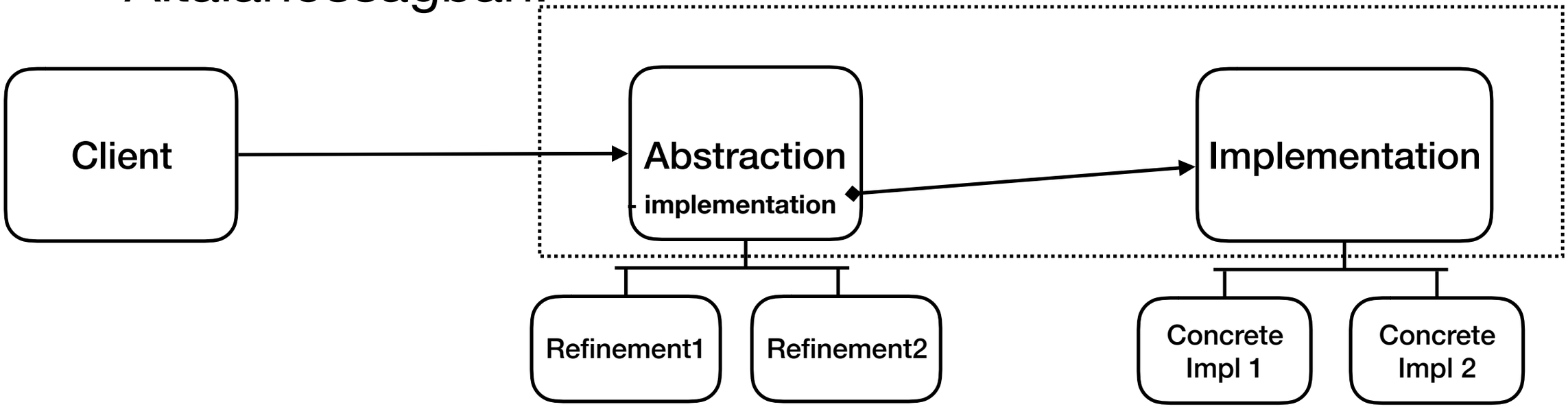
Bridge: mi ez?

- Kevésbé mesterkélt példákkal:



Bridge: mi ez?

- Általánosságban:



Bridge: hasonlóságok

- Az Implementation rész egyfajta általánosítása az Abstract Factory-nak, de nem csak objektum-létrehozást, hanem egyéb műveleteket is megvalósít.
- Akárcsak az Adapter, az Abstraction rész elrejti az Implementation részt a kliens kódtól és a kliens felé egy másfajta interfészt mutat.
    - A különbség az, hogy itt mi irányítjuk a Bridge mindkét oldalát. A cél nem inkompatibilitás megoldása, inkább a bonyolultság csökkentése illetve magasabb szintű műveletek nyújtása a kliensnek.

Bridge: mikor alkalmazzuk?

- Ha van egy monolitikus osztályunk szétválasztható aspektusokkal a kódjában (absztakció-platform, business logic-infrastruktúra, front-back end…)
- Ha az osztály több dimenzióban terjesztendő ki
- Ha több megvalósítással kell működni futás közben (adatbázisok, UI toolkitek, stb.)

Composite

Composite: mi ez?

- Olyan osztályhierarchia, amiben az objektumok csoportosíthatók, és a csoportok szintén tovább csoportosítható objektumokként működnek.
- Faként reprezentálható struktúraknál lehet alkalmazni.
- Két eleme van: a komponens (component), valamint a gyűjtő (container), ami maga is komponens.

Composite: ismert példák

- fájlrendszer: A fájlok a komponensek. A könyvtárak a gyűjtők. A könyvtárak fájlokat tartalmaznak, de a fájlműveletek szempontjából maguk is fájlként viselkednek.

- UI elemek: A gombok, textboxok, stb. komponensek. Ezekből épített panelek gyűjtők, amik maguk is komponensek és további panelekbe ágyazhatók.

- HTML elemek.

- Vállalati és kormányzati hierarchiák.

- Hadseregek.

Composite: delegálás

- A gyűjtő a műveletei zömét azzal valósítja meg, hogy az általa tartalmazott komponenseknek delegálja őket.

- Mivel a beágyazott komponensek is lehetnek gyűjtők, az algoritmusok természetesen rekurzívak.

- Lásd a példáinkban az alakzatok súlypont és területszámítása.

Composite

- Miért jó, ha a gyűjtő egyben komponens is?
- Mert így logikailag csoportosíthatók a komponensek nagyobb komponensekbe.

- Alakzatok példánál maradva: sokszögek lebontása háromszögekre, negatív alakzatokkal komponálás, stb.

Decorator

Decorator

- Úgy ágyaz magába egy objektumot, hogy kibővíti annak a funkcionalitását (“dekorálja új funkcionalitással”).

- A név abból ered, hogy az eredeti alkalmazási területe UI toolkit-ek voltak, ahol pl. egy “ablak” osztályt lehetett dekorálni pl. a “görgethető” vagy “átméretezhető” funkcionalitással.

- Lehetővé teszi, hogy kompozícióval az eredeti objektumra pakoljunk további funkciókat anélkül, hogy minden funkció-kombinációra kellene külön alosztály.

Decorator

- A beágyazott és a dekorátor objektum ugyanazt az interfészt mutatják kifelé.

- Ez a mint szerkezetileg azonos a Proxy-val. Gyakran mindkettőt egy általánosabb “Wrapper” vagy “Handle/Body” minta alesetének tartják.

- A különbség nem a struktúra, hanem a szándék.

  - Proxy jellemzően nem bővíti az eredeti objektum működését, csak megfigyeli, továbbítja, szabályozza.

Facade

Facade

- Több komplex alrendszerhez biztosít egyszerű hozzáférést.
- Az alrendszerek nem tudnak a homlokzat/front létezéséről.
- Valós életbeli példa: utazási iroda. Felénk azt az intefészt mutatja, hogy befizetünk egy útra. Ő ezt összekoordinálja a légitársasággal, a hotellel, az idegenvezetővel, stb.
- Más életbeli példa: online bolt. Mi megveszünk rajta valamit, ő koordinál a raktárral, a csomagkészítéssel, a fizetőszolgáltatással, a szállítással. Meg a beszerzéssel, meg a könyveléssel…

Facade: probléma

- Adott több rendszer.
- Végre kell hajtani egy többműveletes folyamatot ami ezeket a rendszereket magába foglalja.
- A műveleteket helyes sorrendben kell végrehajtani, adott esetben többlépéses koordinálással az alrendszerek között.

Facade: megoldás

- A komplex folyamatokat beskatulyázzuk egy Facade osztályba. A kliens csak a Facade magas szintű műveleteit használja.

- Ugyanaz a Facade mutathat több interfészt is különböző kliensek felé (pl. ügyfél vs. vállalati, felhasználó vs. admin felület).

- Alternatív megoldásként, lehet több külön Facade is ugyanazokhoz az alrendszerekhez.

Facade: hasonlóságok

- A Bridge-ben az Abstraction rész hasonlóan magasabb szintű műveleteket prezentál a kliensnek egy alacsonyabb szintű műveletekkel operáló Implementation-nel, de ott egyszerre egy alrendszer van (implementation). A Facade esetén ezen kívül az alrendszerek jellemzően függetlenül is léteznek a Facade-től.

- Az Adapter is más interfészt mutat kifelé egy kezelt objektumról, de itt jellemzően egy meglévő interfészt adaptál, nem pedig egy komplexebb funkcionalitást mutat kifelé.

- A Facade gyakran lesz Singleton.

Facade

- Maximum hatékonyságért érdemes a rendszert úgy írni, hogy az alrendszerekkel ha lehet mindig a Facade-en keresztül kommunikáljanak a kliensek, az akkor védi őket a változásoktól is.

- Ekkor viszont a Facade hajlamos un. “god object”-té változni - egy olyan objektum, ami a rendszer minden részével kapcsolatban áll és azt irányítja.

- Adott esetben lebontható több kissebb Facade-re.

Flyweight

Flyweight

- Probléma:
  - sok memóriát foglalnak az objektumaink.
    - az objektumok között több mező értékei megegyeznek.
- Megoldás:
  - külön osztályba kiemeljük az egyező értékű mezőket.

Flyweight példa

```
class Location {
    public String city;
    public String region;
    public String countryCode;
    public int metro;
    public List<String> placeIds;
    public double lat;
    public double lon;
    public double confidence;
```

Flyweight példa

```
class SharedLocation {
    public String city;
    public String region;
    public String countryCode;
    public int metro;
    public List<String> placeIds;
class UniqueLocation {
    private SharedLocation sharedLocation;
    public double lat;
    public double lon;
    public double confidence;
```

Flyweight

```
class Location {
    public String city;
    public String region;
    public String countryCode;
    public int metro;
    public List<String> placeIds;
    public double lat;
    public double lon;
    public double confidence;
```

- A nem normalizált alak még súlyosbítható azzal, ha a String-ek sincsenek egyértelműsítve.

Flyweight

- Rengeteg felhasználási terület:
    - Szövegszerkesztő, grafikai progam, böngésző: elemek stílusai
    - Játékok: egységek közös jellemzői, sprite-ok/ részecskék közös jellemzői
- Rokon technika az adatbáziselméletben ismert második normálforma (laikusan: az egybetartozó adatokat csak egyszer fejezzük ki).

Flyweight

- Rengeteg felhasználási terület:
    - Szövegszerkesztő, grafikai progam, böngésző: elemek stílusai
    - Játékok: egységek közös jellemzői, sprite-ok/ részecskék közös jellemzői
- Rokon technika az adatbáziselméletben ismert második normálforma (laikusan: az egybetartozó adatokat csak egyszer fejezzük ki).

Flyweight megvalósítás

- Kell egy Flyweight osztály, amely példányai a közös állapotrészeket tartalmazzák. A flyweight példányai nem módosíthatók létrehozásuk után.

- Kell egy Kontextus osztály, amely példányai kombinálják a flyweight-et az egyedi(bb) állapottal. Kontextus objektumokból sok van.

- Kell egy FlyweightFactory, ami igazából egy cache, ami lehetővé teszi, hogy megtaláljuk vagy létrehozzuk a megfelelő flyweight példányt az újonnan létrehozandó kontextus példányokhoz.

Viselkedési minták

Iterator

Iterator: mi ez

- Nagyon elterjedt példa, ma már annyira nyilvánvaló, hogy sokszor fel sem tűnik.

- Egy csoportobjektum bejárását függetlenítjük annak struktúrájától.

Iterator: mi ez

```
package java.lang

public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

```
package java.util

public interface Iterator<T> {
    public T next();
    public boolean hasNext();
}
```

```
for (Object e: someIterable) {
    doSomething(e);
}
```

Iterator

- Vannak objektumok, amelyeknek lehet több alkalmazástól függő bejárása is.
- Tipikus példa a fák, amelyeknek bejárhatók széltében is meg mélységben is. Mélységi bejárás is lehet többféle attól függően, hogy magát a csomópontot mikor járjuk be.

Iterator: megvalósítás

- Kell egy “Iterable” interfész, amin keresztül a csoportobjektumok tudnak iterátort létrehozni magukra.

- Kell egy “Iterator” interfész, ami jellemzően két metódust ismer: “hasNext” meg “next”. (Vannak más dizájnok is.)

- A konkrét csoportobjektum-osztályok megvalósítják az “Iterable” interfészt, és jellemzően belső osztályként van iterátor-osztályuk.

- Minden “Iterable.iterator()” hívás új iterátort kell, hogy létrehozzon, amivel a többitől függetlenül járható be a csoport.

Iterator: alkalmazhatóság

- Akkor használjuk az Iterator-t, ha:
  - bonyolultabb adatszerkezetünk van, és el akarjuk rejteni a szerkezetét a bejáráshoz.
  - bejárási kód duplikálását elkerülendő
  - ha a kliens kódnak többféle szerkezetet is tudnia kell bejárni, és ezek esetleg ismeretlenek előre.

Visitor

Visitor

- Probléma:
  - több osztály példányaiból álló objektum-struktúránk van. Meg kell valósítanunk valami funkcionalitást a bejárásukkal.
    - A funkcionalitás nem vág bele az osztályok alapcéljaival, ezért nem akarjuk oda belekeverni őket. Több fajta ilyen funkcionalitás is elképzelhető.
- Megoldás:
  - szétválasztjuk a bejárás logikáját a bejárással megvalósítható funkciótól.

Visitor

- Az iteratorral szemben, ami elrejti a bejárás logikáját, itt kimondottan érdekel minket a struktúra.

- Jellemzően kombináljuk a **Composite** mintával, kompozitok bejárására nagyon alkalmas.

Visitor

```
public interface ShapeVisitor {
    public void visitSquare(Square s);
    public void visitCircle(Circle c);
    public void visitTriangle(Triangle t);
    public void visitCompositeShape(CompositeShape c);
}
```

```
public interface Shape {
    …
    public void visit(ShapeVisitor v);
}
```

```
public class Square implements Shape {
     …

   public void accept(ShapeVisitor v) {
        v.visitSquare(this);
    }
}
```

Visitor

- Ha van egy osztálystruktúránk, ami fogad látogatókat, akkor azokkal mindenfajta új számításokat végrehajthatunk. Pl.
  - lekódolhatjuk XML-be, JSON-ba, vagy valami másba.
  - Kereshetünk vagy összegezhetünk benne (“hány kör van összesen az alakzatokban”).
  - Építhetünk egy másik struktúrát, módosításokkal. Például végigmehetünk egy HTML DOM struktúrán és kereshetünk kulcsszavakat, amiket linkké alakíthatunk.

Visitor

- Ezt a megoldást “double dispatch”-nak is hívják: az objektummal elfogadtatjuk a látogatót (első hívás), majd az objektum meghívja a látogatón a megfelelő metódust és átadja magát paraméterként (második hívás).

- Igényel módosítást a megcélzott osztályokon (accept metódust meg kell valósítani), de ez kicsi módosítás és általános kiterjesztési pont onnantól.

- A célosztályoknak nem kell ismerniük csak a visitor interfészt, de a visitor megvalósításoknak ismerniük kell az összes konkrét osztályt.

Visitor: hátrányok

- A fogadó osztályoknak olvasásra elérhetővé kell tenniük az állapotukat.

- Ha új látogatható osztályokat adunk hozzá a rendszerhez, az összes látogató kódját frissíteni kell.

Observer

Observer

- Lehetővé teszi, hogy objektumok feliratkozzanak arra, hogy értesítéseket kapjanak, amikor egy másik objektum állapota változik.

- Egy megoldása annak a problémának, hogyan propagálhatók változások a programban (pl. model és view között akár.)

Observer

- Ha nincs observer, akkor is propagálhatók a változások, de ezek a módszerek nem annyira hatékonyak:
  - aki megfigyelni akar, időszakosan lekérdezheti az objektumot, de akkor veszteget CPU-t ha nincs változás, ha pedig van, akkor késleltetve veszi észre.
  - az objektum maga is megpróbálhatja értesíteni az összes objektumot, ami érdekelt lehet a változásban, de akkor sok olyat is értesít, aki nem az.

Observer

- Terminológiák:
  - observer - observable
  - observer - subject
  - subscriber - publisher
  - (change) listener - ?

Observer

- Java tele van példákkal.
  - Van két általánosan használható java.util.Observable és java.util.Observer osztály.
  - AWT event listener-ek és azokkal való munkát egyszerűsítő java.awt.AWTEventMulticaster
  - Swing event listener-ek és azokkal val munkát egyszerűsítő javax.swing.event.EventListenerList

Observer

- Jellemzően van egy “Observable” osztály, aminek vannak metódusai amikbe regisztrálhatunk megfigyelőket (“addObserver”/“addListener”) meg deregisztrálhatjuk őket (“removeObserver”/“removeListener”). A megfigyelt objektumnak magának kell kezelnie a figyelői listáját.

- Van egy “Observer” interfész egy értesítési metódussal (“update”, “onChange”, stb.), amit a megfigyelt objektumnak kell meghívnia a regisztrált figyelőin amikor megváltozik az állapota.

Observer

- A megfigyelt objektum a megfigyelőinek csak az interfészét ismeri, megvalósításukat nem kell ismernie.

- Lehet úgy is csinálni, hogy a megfigyelt objektummal több különböző megfigyelő is regisztrálható különböző eseményekhez.

Observer: alkalmazhatóság

- Ha bizonyos objektumokban bekövetkező változások további változásokat kell okozzanak más objektumokban.

- Az, hogy mely objektum hatnak ki mely másokra nem feltétlenül tudott előre.

- Akkor is alkalmazható, ha a változásokat csak egy ideig kell érzékelni (előtte/utána regisztrálható/deregisztrálható a figyelő).

Command

Command

- A Command (“parancs”) általánosítása a metódushívásnak.

- Ha pl. van egy szerkesztőnk, ami tudja elmenteni a tartalmát, akkor azt kódból meghívhatjuk, közvetlenül úgy, hogy “editor.save()”.

- Ha sok helyről kell tudni meghívni (gombnyomásra, menüből, billentyű-kombinációra), akkor viszont bonyolódik a helyzet.

Command

- Létre kell hozni egy interfészt, aminek van “execute” vagy hasonló metódusa. Pl. Swing-ben az Action is egy példája a Command pattern-nek.

- Azokba az elemekbe, amik végre tudják hajtani a parancsot, regisztrálni kell a parancsokat. Ilyen elemek a menüpontok, gombok, billentyű-kombináció kezelők, stb.

- Amikor az elem aktiválva van (gomb lenyomva, menü elem kiválasztva, stb.), akkor azok meghívják a parancs “execute” metódusát.

- Ezzel szétválasztjuk a parancs megfogalmazását/konfigurálását (az a kód csinálja, ami építi az UI-t) a végrehajtásától (amit később végez az UI).

Command

- Nagyon sok más példája is van a Command-nak, ami nem az UI paradigmában van:
  - SQL query-k végrehajtása
  - Távoli objektumoknak kérések küldése.
  - Ha van egy sor parancsunk, akkor lehetőségünk van arra, hogy továbbítás/végrehajtás előtt beléjük avatkozzunk (csoportosítás, átrendezés, összevonás, stb.)

Command

- További lehetőség a Command patternnel (nem mindig) az, hogy létre tudunk hozni undo funkcionalitást.
- Ha pl. egy szerkesztőben minden műveletet kifejezünk egy Command objektummal, és megjegyezzük őket egy listában, valamint a Command-nak van egy “undo” metódusa is ami visszafelé tudja megvalósítani, akkor a lista segítségével tudunk undo funkcionalitást biztosítani.

Memento

Memento

- Lehetővé teszi, hogy egy objektum állapotát biztonságosan el lehessen menteni majd visszaállítani.

- Kiválóan alkalmas ezért undo funkcionalitás megvalósítására.

- “Biztonságosan” azt jelenti, hogy az objektum kiadhatja az állapotát de úgy, hogy az kívülálló csak megőrizheti, de nem tudja sem olvasni sem módosítani.

Memento

- A mementó osztálya olyan kell legyen, hogy az őt kiadó osztály ismeri csak (privát belső osztály Java-ban).

- A mementó az objektum állapotának másolatát tartalmazza (nem feltétlenül azonos formátumban).

- Kifelé a kiadó osztály csak “Object”-ként adja ki és fogadja be. Ezért a kliens nem tud vele mást csinálni, csak visszaadni az objektumnak.

- Arra is alkalmas lehet, hogy különböző példányok között másoljunk állapotot.

Memento

- Akár csak a Command, ez is alkalmas undo funkció megvalósítására. A Memento az egész objektum állapotát menti, ezért több memóriát foglal. A Command-undo viszont több hibalehetőséget rejt a megvalósításban.

- Akár csak a Prototype, ez is alkalmas arra, hogy különböző objektumokat konfiguráljunk egy prototípusból, ha kombináljuk a klónozással.

Chain of Responsibility

Chain of Responsibility

- Munkafolyamatok szervezésére szolgál lépésenként.
- “Kezelők” kezelnek “kéréseket”. A kezelők láncolva vannak egymás után. Minden kezelő végrehajt valamilyen részfeladatot a kérésen, majd továbbítja a következő kezelőnek.

Chain of Responsibility

- Példák:
    - Billentyű-lenyomás kezelése UI-ban. Mindig van egy fókuszált elemünk (gomb, szövegmező, stb.). Az kapja először a lehetőséget, hogy feldolgozza az eseményt. Ha nem kezeli, akkor az továbbmegy az őt tartalmazó panelhez, majd ablakhoz (hátha ablakzáró parancs), majd menühöz/alkalmazáshoz, végül az operációs rendszer fő UI-hoz.

Chain of Responsibility

- Példák:
  - Web alkalmazásokban HTTP kérés kezelése. Általában “csővezetéket” (pipeline) konfigurálunk, ami kezeli az azonosítást, cache-elést, titkosítást, stb. mielőtt a végső kezelőhöz (servlet) kerülne a végrehajtás.

Chain of Responsibility

- Egy lehetséges megvalósítás: kezelő maga tárolja, hogy ki a következő kezelő a láncban.

```
abstract class Handler {
    private Handler next;

    public void setNext(Handler next) {
      this.next = next;
    }

    public void handle(Task task) {
      if (handleImpl(task) && next ≠ null) {
        next.handle(task);
      }
    }

    abstract boolean handleImpl(Task task);
}
```

Chain of Responsibility

- Másik lehetséges megvalósítás: a kezelőnek minden híváskor átadunk egy objektumot, ami képviseli a lánc hátralévő részét.

```
interface Handler {
  public void handle(Task task, HandlerChain rest);
}
```

```
interface HandlerChain {
  public void handle(Task task);
}
```

```
abstract class HandlerBase implements Handler {

   public void handle(Task task, HandlerChain rest) {
      if (handleImpl(task)) {
        rest.handle(task);
      }
    }

    abstract boolean handleImpl(Task task);
}
```

Chain of Responsibility

- Végül modern Java-ban a HandlerChain el is hagyható:

```
interface Handler<T> {
  public void handle(T task, Consumer<T> rest);
}
```

```
abstract class HandlerBase<T> implements Handler<T> {

   public void handle(T task, Consumer<T> rest) {
      if (handleImpl(task)) {
        rest.accept(task);
      }
    }

    abstract boolean handleImpl(Task task);
}
```

Chain of Responsibility

- De ha nagyon akarjuk, a Handler is :

```
abstract class HandlerBase<T> implements BiConsumer<T, Consumer<T>> {

   public void accept(T task, Consumer<T> rest) {
      if (handleImpl(task)) {
        rest.accept(task);
      }
    }

    abstract boolean handleImpl(Task task);
}
```

Chain of Responsibility

- Mikor használjuk:
    - ha különböző feladatokat/kéréseket kell feldolgozni különbözőképpen, és a lépések konfigurálhatók.
    - ha meghatározott sorrendben kell végrehajtani műveleteket, amik külön-külön osztályban vannak definiálva.
    - ha ez a sorrend változhat futás közben.

Chain of Responsibility

- A kezelők a hatásukat mellékhatásként fejtik ki (mivel a lánc maga void műveletekkel működik). Ezek lehetnek akár külső mellékhatások, vagy a feladat-objektum állapotának módosítása.

- Minden kezelőnek el kell döntenie, hogy tesz-e valamit a feladattal, valamint, hogy továbbküldi-e a következő elemnek.

- A kezelő akár létre is hozhat új feladat objektumot (ami legtöbbször dekorálja azt, amit ő kapott), és azt küldheti tovább a láncban. Így a dekorátoron keresztül befolyásolhatja a további láncelemek működését.

Chain of Responsibility

- A kliens vagy maga épít láncot, vagy azt kívülről kapja.
- A kliens nem tudja, hány eleme van a láncnak, ő csak belekküldi a feladat-objektumot végrehajtásra.
- A lánc megkövetelheti, hogy az utolsó elem már ne küldje tovább a feladat-objektumot (pl. exception-t dob ilyen esetben).

Chain of Responsibility

- Ez a minta gyakran fordul elő a Component-tel, mivel a komponesek természetes láncokat alkotnak a levéltől a gyökérig.

- A “feladat” gyakran Command, ami addig terjed a láncban, amíg valamelyik kezelő nem tudja végrehajtani.

- A kezelő is lehet Command. Ebben az esetben ugyanaz a parancsobjektum végrehajtható különböző láncok kontextusában.

- Az első megvalósítása (setNext) hasonlít a Dekorátorra, de ez csak szerkezeti hasonlóság, más szándéka van a kettőnek.

Strategy

Strategy

- Az objektum viselkedésének valamely részét kiszervezzűk egy interfészbe (ami a stratégia), majd ennek különböző megvalósításait biztosítjuk.

Strategy

- Példák:
  - A factory method példában a Transport az lehet Strategy, ami a DeliveryPlan-hez tartozik.
  - Webshopnál ami különböző országokba küld termékeket az ÁFA és egyéb adók számolása lehet országonkénti Strategy, amit beteszünk a “megrendelés” objektumba.

Strategy

```
abstract class Logistics {
  public DeliveryPlan makeDeliveryPlan(…) {
    Transport t = createTransport();
    …
    return new DeliveryPlan(…, t);
  }

  abstract Transport createTransport();
}
```

```
interface Transport {
  public void deliver();
}
```

Strategy

- Akkor használjuk ha:
  - több osztály is alkalmazhatja ugyanazt a stratégiát, ezért belekomponálhatjuk objektumként
  - az osztály fő logikájától elválasztjuk a megvalósításának kevésbé fontos részleteit
  - ha sok helyen van if/else-if/else-if logikánk a kódban

State

State

- Lehetővé teszi, hogy az objektum módosítsa a viselkedését amikor megváltozik a belső állapota.

- Olyan osztályoknál van értelme alkalmazni, ahol az objektumok viselkedése jelentősen eltérhet annak függvényében, hogy milyen állapotban vannak.

State

- Példák:
  - ügykövető rendszerekben egy ügy állapota
  - objektumok, amik több inicializálási fázison mennek keresztül
  - megválasztott vezetővel rendelkező hálózati rendszerekben minden szereplő lehet vezető, követő, vagy jelölt.

State

- Miben más ez mint a Strategy?
    - A Strategy-t kívülről konfigurálja a kliens kód. A State belső állapotátmeneteket jelenít meg. Az objektum bizonyos feltételek teljesülése esetén magától vált közöttük.
    - Különböző State megvalósítások összefüggnek. A Strategy-k egymástól függetlenek.

Template Method

Template Method

- Emlékszünk a Factory Method és Abstract Factory-ra?
- Azok a metódusok, amik használták őket (vagyis a közös elemek az alaposztályban) azokat nevezzük Template Methodnak.

Template Method

```
abstract class Logistics {
  public DeliveryPlan makeDeliveryPlan(…) {
    Transport t = createTransport();
    …
    return new DeliveryPlan(…, t);
  }

  abstract Transport createTransport();
}
```

```
interface Transport {
  public void deliver();
}
```

Template Method

```
abstract class UI {
  public Dialog makeDialog(String msg, String[] buttonLabels) {
    UIText t = createText(msg);
    UIButton[] bs = buttonLabels.map { l ⇒ createButton(l) }
    Dialog = new Dialog(t, bs)
  }

  abstract UIText createText(String label);
  abstract UIButton createButton(String label);
}
```

Template Method

- Template Method célja ugyanaz, mint a Strategy-nek: az objektum viselkedésének egy részének kiemelése.
- A Template Method esetén a viselkedés öröklés lévén kerül meghatározásra, osztály szintjén. A Strategy ezzel szemben objektumonként teszi lehetővé az objektum és a viselkedés összeállítását.

További erőforrások

- Nagyon jó, ingyen elérhető mintakatalógus: https://refactoring.guru/design-patterns