Szoftvertechnológia

Tervminták

Dr. Szendrei Rudolf
ELTE Informatikai Kar
2020.

Tervminták

Emlékeztető: Mi is a tervminta?

- A tervezési minták tipikus megoldást jelentenek a szoftverek tervezésében gyakran felmerülő problémákra. Olyanok, mint a tervrajzok, melyeket testreszabhatunk a részben különböző, de ismétlődő tervezési problémák megoldására.
- Nem elég megtalálni egy mintát, és bemásolni azt a programunkba, ahogy azt a polcról levehető kész komponensekkel vagy program könyvtárakkal szokás tenni.
- A minta nem egy konkrét kódrészlet, hanem egy általános koncepció egy adott probléma megoldására.
- A mintát követve készíthetünk egy olyan megoldást, amely megfelel a saját programunk sajátosságainak.

Ajánlott irodalom:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design patterns

Tervminták

Emlékeztető: Mi is a tervminta?

- A mintákat gyakran összekeverik az algoritmusokkal, mivel mindkét koncepció ismert problémák tipikus megoldásait írja le.
- Míg az algoritmus mindig egyértelműen meghatározza azokat a műveleteket, amelyek valamilyen célt elérhetnek, így analógiaként a főzési recept juthat eszünkbe.
- A minta a megoldás magasabb szintű leírása, így egy mintának a különböző programokban történő alkalmazása adhat eltérő kódot. Ezért ezt inkább egy fajta tervrajzként foghatjuk fel, ahol láthatjuk, hogy mi lesz az eredmény és annak jellemzői, de a végrehajtás pontos sorrendje rajtunk áll.

Tervminták

Történet

- Négy szerző kapta fel az ötletet:
  Erich Gamma, John Vlissides, Ralph Johnson és Richard Helm.
- 1994-ben publikálták:
  Design Patterns: Elements of Reusable Object-Oriented Software, melyben tervezési minták koncepcióját alkalmazták a programozásra.
- 23 mintát mutattak be, amelyek megoldják az objektum-orientált tervezés különböző problémáit, és nagyon gyorsan bestsellerré váltak.
- Azóta tucatnyi más, objektumorientált mintát fedeztek fel.
- A „minta megközelítés” nagyon népszerűvé vált más programozási területeken is, így sok más minta létezik az objektum-orientált tervezésen kívül is.

Tervminták

Miből áll a tervminta?

- A legtöbb minta leírása kellően formális ahhoz, hogy többféle esetben is megvalósítható legyen. A **mintaleírás szakaszai** általában az alábbiak szoktak lenni:
  - A **minta szándéka** röviden leírja mind a problémát, mind a megoldást.
  - A **motiváció** elmagyarázza a problémát és megoldását, amelyet a minta lehetővé tesz.
  - Az **osztályok felépítése** megmutatja a minta mindegyik részét és annak összefüggését.
  - **Példa kód** valamely népszerű programozási nyelven, mely megkönnyíti a minta ötletének felfogását.
- Néhány mintakatalógus felsorol más hasznos részleteket is, például a minta alkalmazhatóságát, a megvalósítási lépéseket és az egyéb mintákkal való kapcsolatokat.

Tervminták

Előnyök

- Évekig programozhatunk úgy, hogy egyetlen mintáról is tudnánk, de ekkor is megvalósíthatunk bizonyos mintákat anélkül, hogy tudnánk róla. Miért tanuljuk meg akkor őket?
  - Mert a szoftvertervezés általános problémáinak kipróbált és bevált megoldásainak eszközkészlete. A minták megismerése akkor is hasznos, ha később nem használjuk őket, mert megtanít, a különböző problémák megoldásra az objektum-orientált tervezés elveinek felhasználásával.
  - Meghatározhatják azt a közös nyelvet, amellyel a csapat tagjai hatékonyabban kommunikálhatnak.
    Pl.: „Ó, erre használj Singletont”.
    Nem szükséges elmagyarázni, ha mindenki ismeri a mintát.

Tervminták

Kritikák

- **Mentőöv a gyenge programozási nyelvekhez**
  - A minták általában akkor válnak szükségessé, amikor egy programozási nyelv vagy technológia nem rendelkezik az absztrakció szükséges szintjével.
    - Pl.: a stratégiai minta leírható lambda kifejezéssel
- **Nem hatékony megoldások**
  - A minták megkísérlik a széles körben alkalmazott megközelítéseket rendszerezni. Ezt az egységesítést sokan dogmának tekintik, és „a lényegre” vezetik be a mintákat anélkül, hogy a projekthez igazítanák őket.
- **Indokolatlan felhasználás**
  - „*Akinek csak kalapácsa van, mindent szögnek néz...*”
  - Ez a probléma sok kezdőt megkísért, mert megpróbálják azokat olyan helyzetekben is alkalmazni, amikor egy egyszerű kód is megfelelne.

Tervminták

A tervminták osztályai

- A legalapvetőbb és az alacsony szintű mintákat gyakran idiómáknak nevezik.
  - Általában egyetlen programozási nyelvre vonatkoznak
- A legsokoldalúbb és legmagasabb szintű minták az architekturális minták.
  - Szinte bármilyen nyelven megvalósíthatók.
  - Más mintáktól eltérően, felhasználhatók egy teljes alkalmazás architektúrájának megtervezésére.

Tervminták

A tervminták osztályai

- A mintákat kategorizálhatjuk szándékuk vagy céljuk alapján, melyből mi most hármat különböztetünk meg:
- **Létrehozási minták**
  - Objektum-létrehozási mechanizmusokat biztosítanak, amelyek növelik a meglévő kód rugalmasságát és újrafelhasználását.
- **Szerkezeti minták**
  - Elmagyarázzák, hogyan lehet az objektumokat és osztályokat nagyobb szerkezetekbe összeállítani, miközben a szerkezeteket rugalmasan és hatékonyan tartják.
- **Viselkedési minták**
  - Gondoskodnak az objektumok közötti hatékony kommunikációról és a felelősség megosztásáról.

Tervminták

Létrehozási minták

- Gyártó művelet (Factory method) lásd Sz.T. 4. ea. ✓
- Absztakt gyártó (Abstract factory) lásd Sz.T. 4. ea. ✓
- Egyke (Singleton) lásd P.T. 2. ea. ✓
- Építő (Builder)
- Prototípus (Prototype)

Tervminták

Építő (Builder)

- Célja, hogy alternatívát találjon a „teleszkópos” konstruktor anti-mintára, mely akkor jelenik meg, amikor az osztály konstruktor paraméter-kombinációk számának növekedése a konstruktorok exponenciális listáját eredményezi.
- A rengeteg konstruktor helyett egy építő objektumot alkalmaz, mely minden egyes inicializációs paramétert lépésről lépésre kap meg, majd egyben adja vissza az elkészült objektumot.
- A minta lehetővé teszi egy objektum különböző típusainak és ábrázolásainak elkészítését ugyanazon építési kóddal.

- Java-ban ezt a tervmintát használja pl. a StringBuilder…

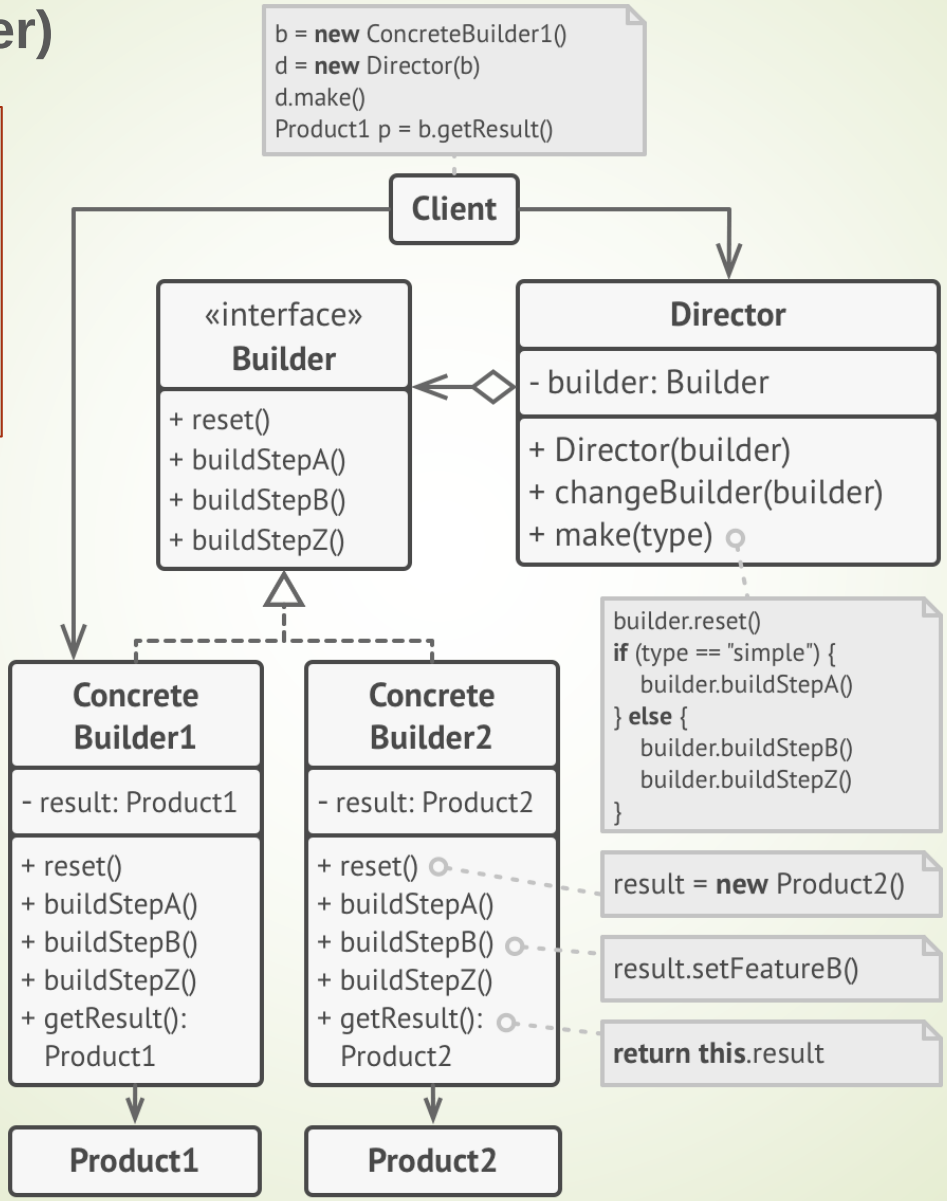
Tervminták

Építő (Builder)

Az Építő interfész deklarálja a termék létrehozásának lépéseit, melyek közösek minden építő típusnál.

A konkrét építők a létrehozás különböző módjait implementálják.

Az építők által létrehozott objektumok nem feltétlenül tartoznak ugyanahhoz az osztályhierarchiá hoz, vagy interfész hez.



A Director osztály meghatározza az építési lépések meghívásának sorrendjét, így létrehozhatjuk / újra felhasználhatjuk a termékek meghatározott konfigurációit.

Tervminták

Építő (Builder)

```
class Car {
  private int wheels;
  private String color;
  public void setWheels(final int wheels) {
    this.wheels = wheels;
  }
  public void setColor(final String color) {
    this.color  = color;
  }
  ...
}

interface CarBuilder {
  CarBuilder setWheels(final int wheels);
  CarBuilder setColor(final String color);
  Car        build();
}
```

Tervminták

Építő (Builder)

```
class CarBuilderImpl implements CarBuilder {
  private Car car;

  public CarBuilderImpl() {
    car = new Car();
  }

  @Override
  public CarBuilder setWheels(final int wheels) {

    car.setWheels(wheels);
    return this;

  }


  @Override
  public CarBuilder setColor(final String color){

    car.setColor(color);
    return this;

  }

  @Override public Car build() { return car; }
}
```

Tervminták

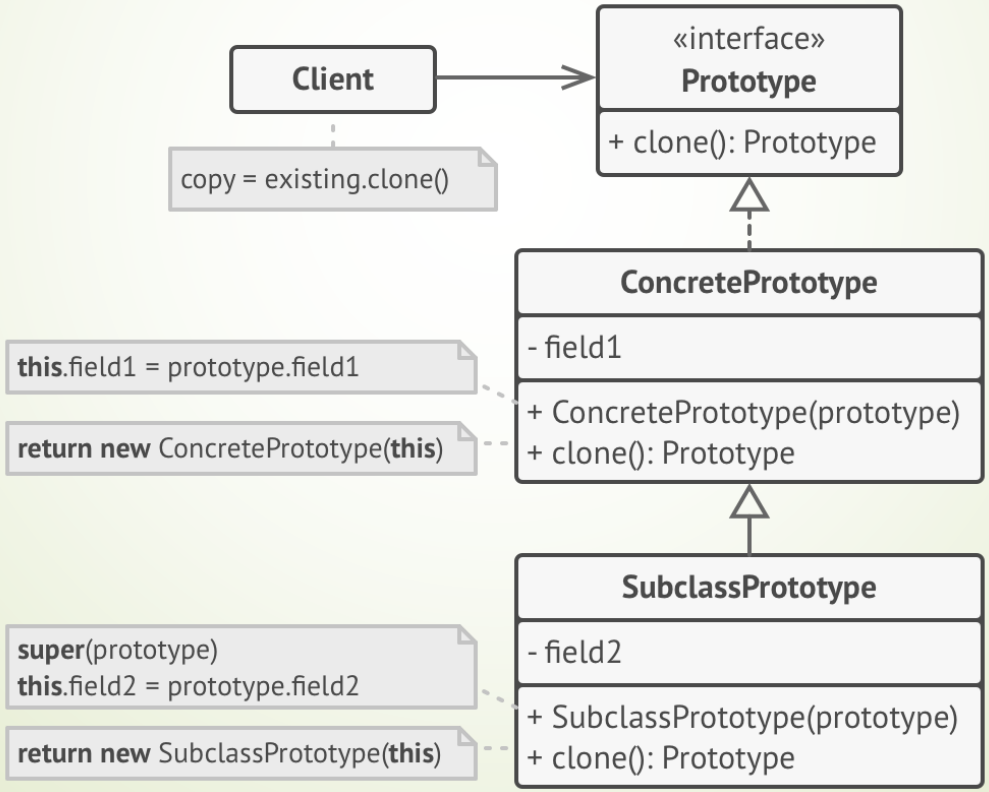
Építő (Builder)

```
public class CarBuildDirector {
  private CarBuilder builder;
  public CarBuildDirector(final CarBuilder builder){
    this.builder = builder;
  }
  public Car construct() {
    return builder.setWheels(4).setColor("Red").build();
  }
  public static void main(final String[] arguments){
    CarBuilder builder = new CarBuilderImpl();
    CarBuildDirector carBuildDirector =
                                  new CarBuildDirector(builder);
    System.out.println(carBuildDirector.construct());
  }
}
```

Tervminták

Prototípus (Prototype)

- A minta lényege a klónozás, azaz az eredeti objektummal megegyező új példány létrehozása
- A konkrét prototípus osztály kezeli a mélymásolás eseteit.



Tervminták

Prototípus (Prototype)

```
public abstract class Prototype {
  public abstract Prototype clone();
}

public class ConcretePrototype1 extends Prototype {
  @Override
  public Prototype clone() {
    return super.clone();
  }
}

public class ConcretePrototype2 extends Prototype {
  @Override
  public Prototype clone() {
    return super.clone();
  }
}
```

Tervminták

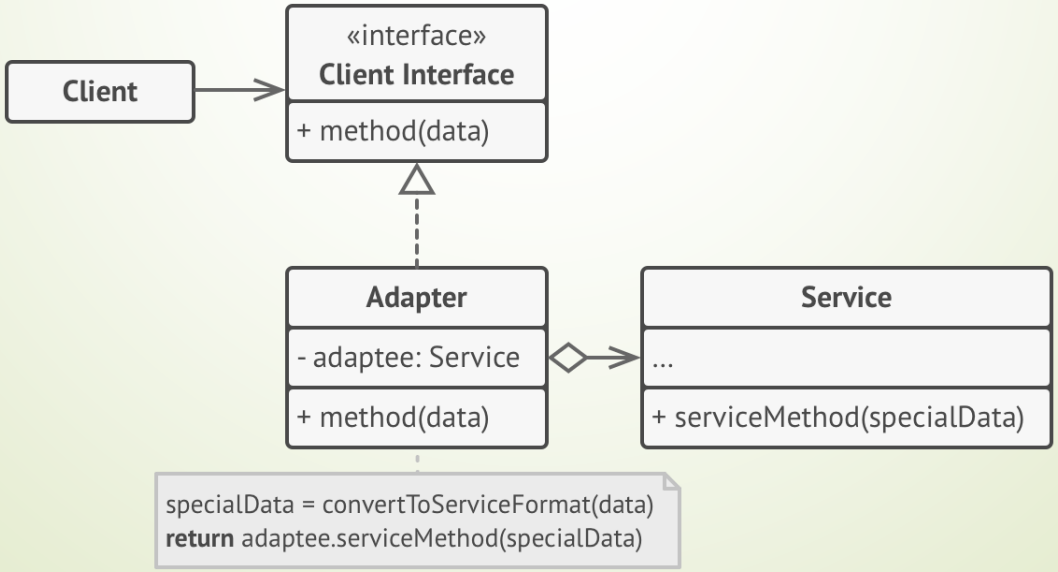
Szerkezeti minták

- Díszítő (Decorator) lásd P.T. 2. ea. ✓
- Illesztő (Adapter)
- Híd (Bridge)
- Összetétel (Composite)
- Homlokzat (Facade)
- Pehelysúlyú (Flyweight)
- Helyettes (Proxy)

Tervminták

Illesztő (Adapter)

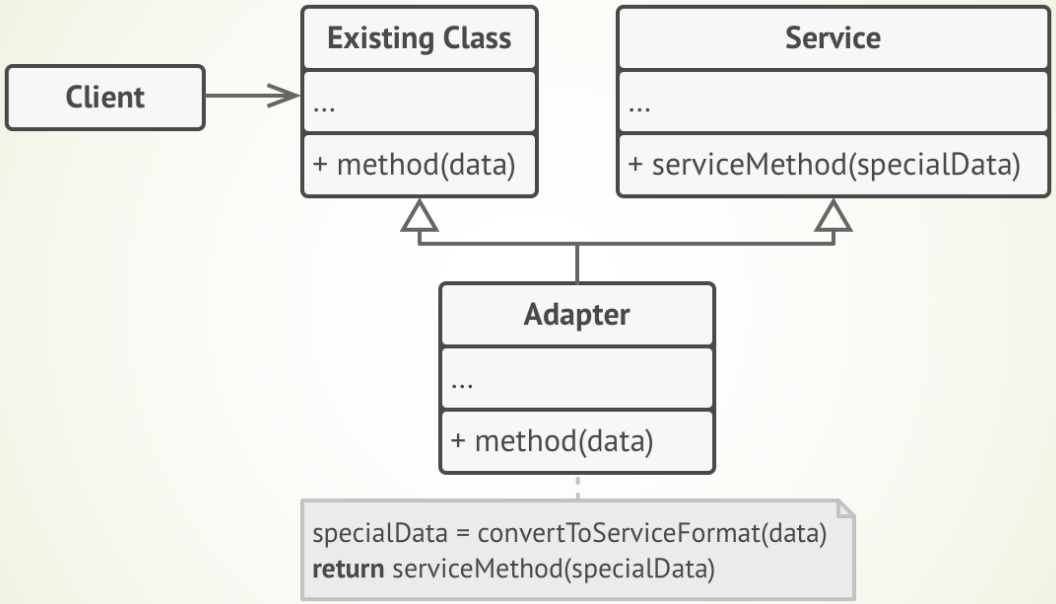
- Lefordítja egy osztály interfészét egy kompatibilis másik interfészre.
- Egy *illesztő* lehetővé teszi olyan osztályok együttműködését, amelyek az inkompatibilis interfészeik miatt normálisan nem tudnának együttműködni, mindezt úgy, hogy interfészt nyújt a kliensek számára, míg ő maga az eredeti interfészt használja
- Objektum adapter esetében:



Tervminták

Illesztő (Adapter)

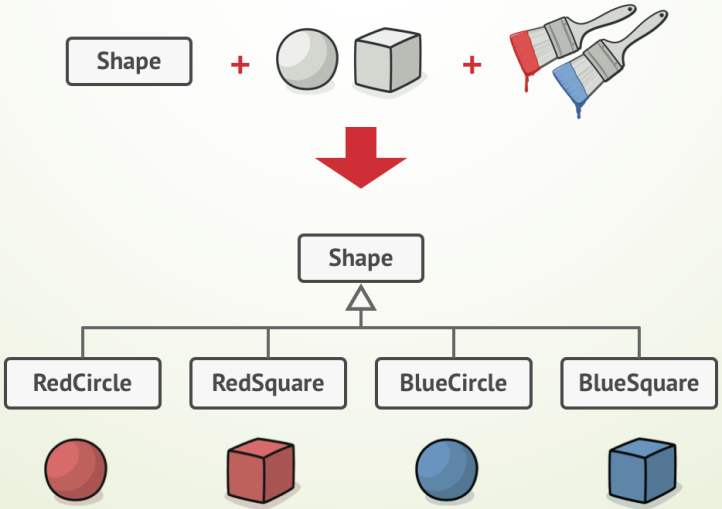
- Osztály adapter esetében:



Tervminták

Híd (Bridge)

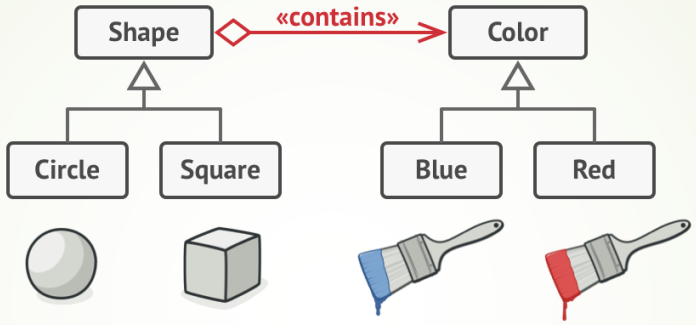
- Lehetővé teszi egy nagy osztály vagy szorosan kapcsolódó osztályok halmazának két külön hierarchiára - absztrakcióra és megvalósításra - felosztását, amelyeket egymástól függetlenül lehet lefejleszteni.
- Tipikus használati helye, amikor az osztály származtatását a különböző tulajdonságai alapján végezzük, ahol a tulajdonságok egymással párhuzamosan léteznek.

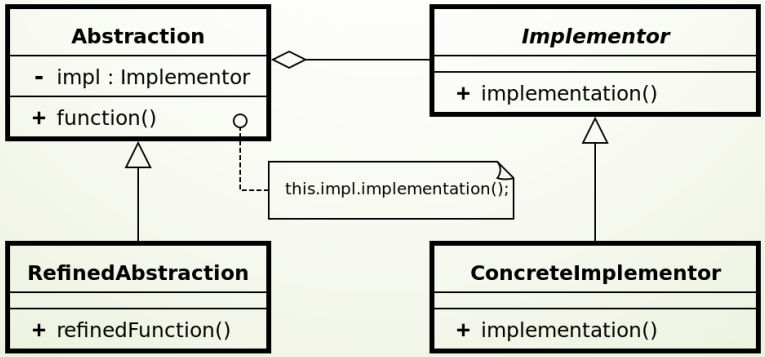


Tervminták

Híd (Bridge)

- A megoldás a származtatás lecserélése kompozícióra.

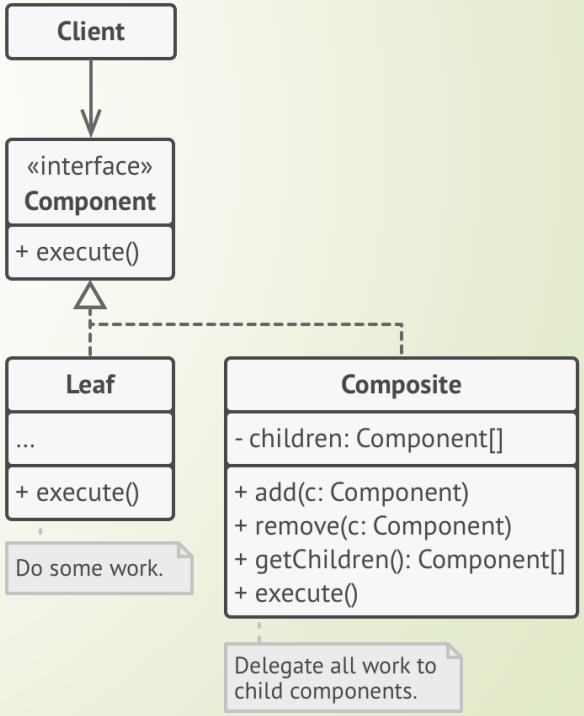




Tervminták

Összetétel

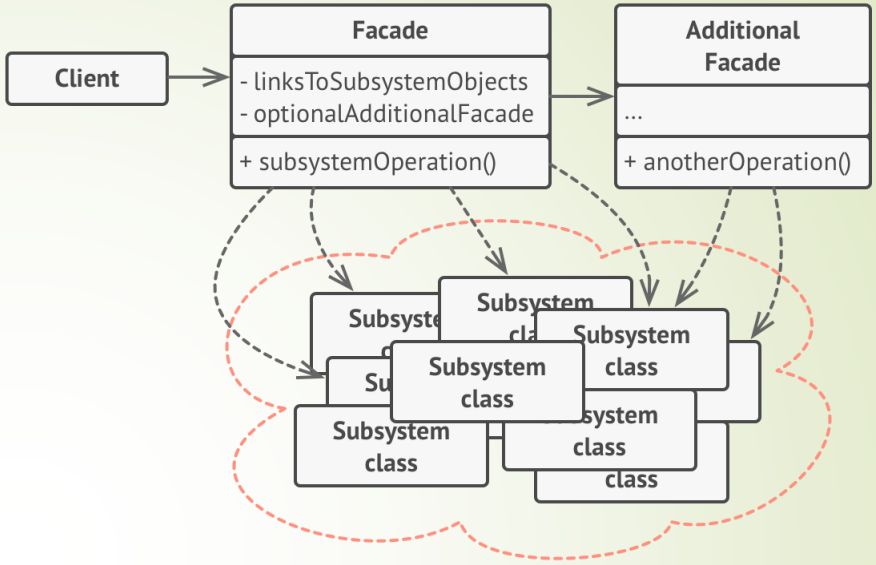
- A minta azt írja le, hogy az objektumok egy csoportját ugyanúgy kell kezelni, mint egy adott objektum példányait külön-külön.
- Az összetétel itt arra utal, hogy fa struktúrába szervezünk objektumokat így reprezentálva a rész-egész hierarchiákat.
- Az összetétel minta lehetővé teszi, hogy a kliensek az önálló objektumokat és összetételeket egységes módon kezeljék.
- Tipikus példája a bináris keresőfa, ahol minden egyes csúcs a gyerekeivel együtt szintén egy keresőfát alkot.



Tervminták

Homlokzat (Facade)

- Egyszerűsített felületet biztosít egy könyvtárhoz, keretrendszerhez vagy bármely más komplex osztálycsoporthoz.



- Alap probléma
  - Képzeljük el, hogy a programnak sok olyan objektummal kell dolgoznia, amelyek egy bonyolult könyvtárhoz tartoznak. Rendszerint az összes objektumot inicializálni kell, nyomon követni a függőségeket, a módszereket a megfelelő sorrendben kell végrehajtani stb.
  - Az osztályok üzleti logikája ezért szorosan összekapcsolódik a mások által írt osztályok végrehajtási részleteivel, megnehezítve ezzel a megértést és karbantartást.

Tervminták

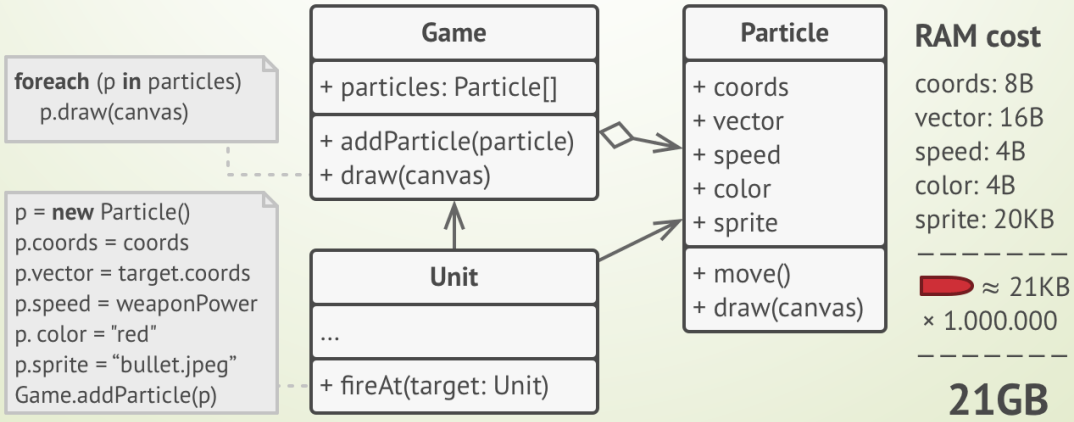
Homlokzat (Facade)

- Megoldás
  - A homlokzat korlátozott funkcionalitást biztosít az alrendszerrel való közvetlen munkához képest, vagyis csak azokat a funkciókat tartalmazza, amelyekkel az ügyfelek valóban törődnek.
- A homlokzat használata akkor hasznos, ha integrálni kell az alkalmazást egy bonyolult könyvtárba, amely több tucat funkcióval rendelkezik, melyből nekünk csak egy apróbb funkcióra van szükségünk.
- Példa
  - Adott egy alkalmazás, amely rövid videókat tölt fel a közösségi médiába, melyhez felhasznál egy videókonvertert.
    Amire valóban szükségünk van, az egy osztály az egyetlen kódolási módszerrel (fájlnév, formátum).
    Ezt az új osztályt összekapcsolva a videókonverterrel, megkapjuk az első homlokzatunkat.

Tervminták

Pehelysúlyú (Flyweight)

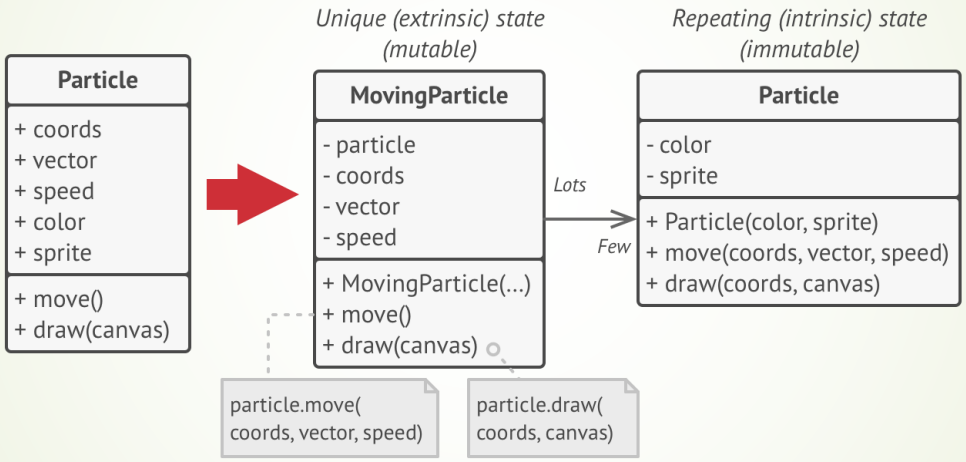
- Segítségével több objektumot helyezhetünk el a memóriában úgy, hogy megosztja az állapot közös részeit több objektum között egy pehelysúlyú objektumban ahelyett, hogy az eredeti objektumokban minden adatot megtartana. Példa:
  - Egy játékban az egyes részecskéket, például egy golyót, rakétát stb. egy-egy különálló objektum képvisel, amely rengeteg adatot tartalmaz. Amikor egy játékos képernyőjén a részecskék száma eléri a csúcspontját, az újonnan létrehozott részecskék már nem férnek el a fennmaradó RAM-ban, így a program összeomlik.



Tervminták

Pehelysúlyú (Flyweight)

- A Particle osztályban észrevehető, hogy a szín és a sprite mezők sokkal több memóriát fogyasztanak, mint más mezők, ráadásul ez a két mező szinte azonos adatokat tárol az összes részecskén.

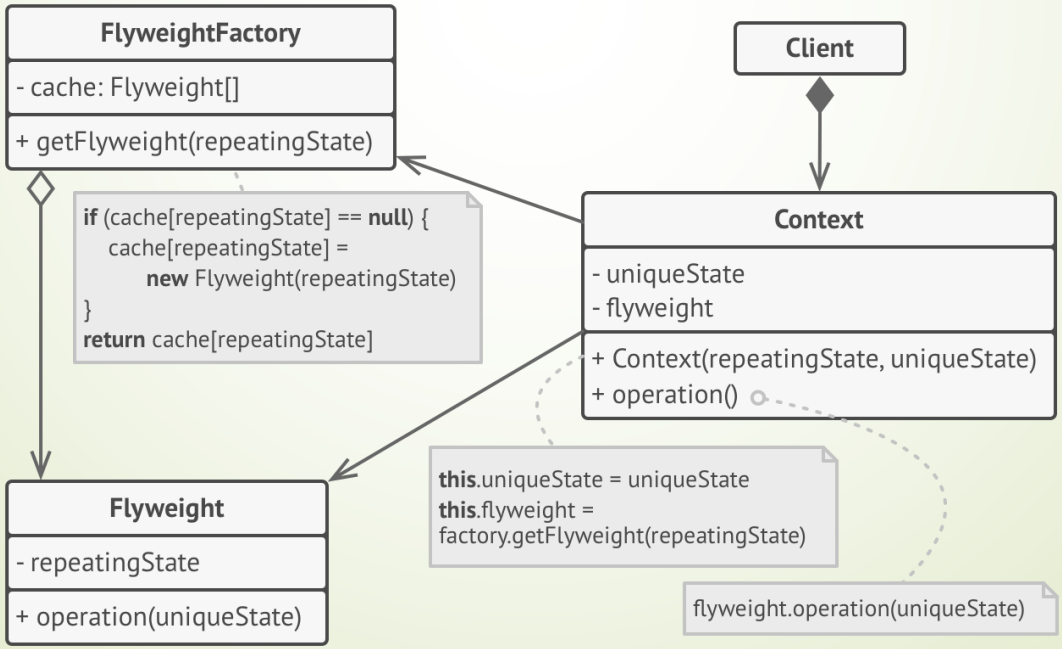


- Az objektum ezen állandó adatait általában belső állapotnak nevezik (a tárgyon belül él, más objektumok csak olvasni tudják, nem változtathatják meg).
- Az objektum többi állapotát, amelyet más tárgyak gyakran "kívülről" megváltoztatnak, külső állapotnak nevezzük.

Tervminták

Pehelysúlyú (Flyweight)

- Ne tároljunk külső állapotot az objektumon belül, szervezzük azt ki olyan objektumba, amelyre a metódusok támaszkodnak.
- A megmaradó pehelysúlyú objektum különböző összefüggésekben használható, meg kell győződni arról, hogy állapota nem módosítható (csak egyszer inicializálhatjuk, a konstruktorával).



Tervminták

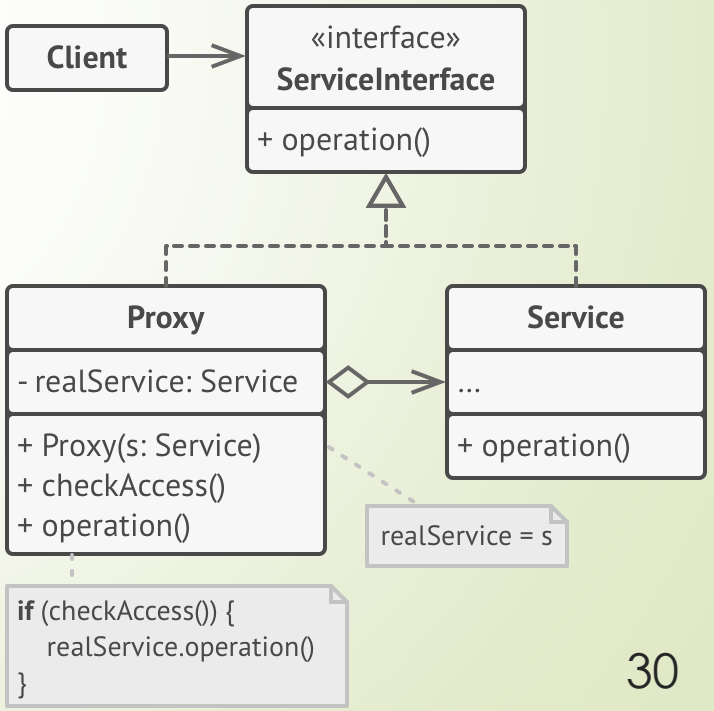
Helyettes (Proxy)

- Lehetővé teszi helyettesítő vagy helyőrző megadását egy másik objektumhoz.
- A proxy vezérli az eredeti objektumhoz való hozzáférést, lehetővé téve, hogy végrehajtsunk valamit a kérés előtt vagy után, hogy az az eredeti objektumhoz elérne.
- Példa
  - Adott egy objektum, amely hatalmas mennyiségű rendszer erőforrást fogyaszt. Időről időre szükség van rá, de nem mindig. (Pl.: egy adatbázis)
  - Lusta inicializálással akkor hozzuk létre az objektumot, amikor ténylegesen szüksége van rá. Az objektum minden „ügyfelének” végre kell hajtania egy ilyen inicializálást, ez azonban sok kódmásolatot okozna.

Tervminták

Helyettes (Proxy)

- Ideális világban az inicializáló kódot közvetlenül a tárgyunk osztályába tennénk, de például harmadik fél könyvtárában ezt nem tehetjük meg.
- Hozzunk létre egy új proxy-osztályt, ugyanolyan felülettel, mint ami az eredeti szolgáltatás objektumnak van.
- Frissítsük az alkalmazást úgy, hogy az átadja a proxy objektumot az eredeti objektum összes ügyfelének.
- Miután kérést kap egy ügyféltől, a proxy létrehoz egy valós szolgáltatási objektumot, és az összes munkát rá delegálja.



Tervminták

Viselkedési minták

- Parancs (Command) lásd Sz.T. 4. ea. ✓
- Megfigyelő (Observer) lásd Sz.T. 4. ea. ✓
- Látogató (Visitor) lásd OAF tárgy ✓
- Felelősséglánc (Chain of Responsibility)
- Bejáró (Iterator)
- Közvetítő (Mediator)
- Emlékeztető (Memento)
- Állapot (State)
- Stratégia (Strategy)
- Sablonfüggvény (Template method)

Tervminták

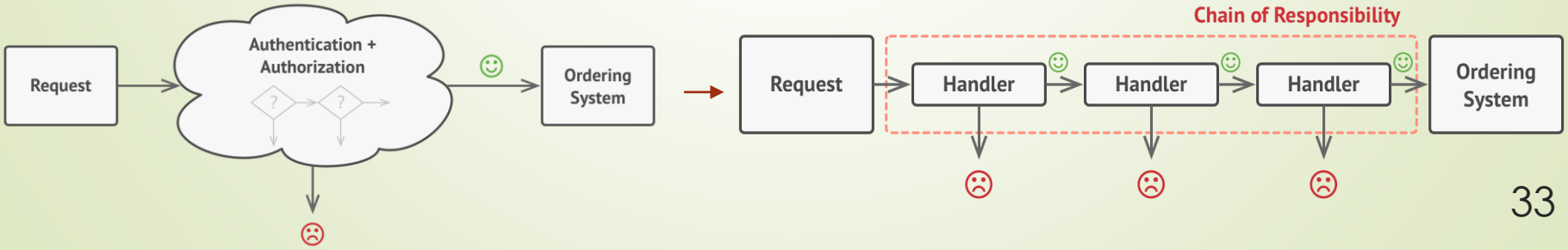
Felelősséglánc (Chain of Responsibility)

- Lehetővé teszi a kérések továbbítását a kezelők lánca mentén. Kérés kézhezvétele után minden kezelő úgy dönt, hogy feldolgozza a kérelmet, vagy átadja a lánc következő kezelőjének.
- Probléma
  - Egy webáruházban a vásárláshoz be kell jelentkezni, hogy utána kéréseket küldhessünk a szervernek. Mindig megvizsgáljuk, hogy a felhasználó bejelentkezett-e.
  - Később újabb és újabb ellenőrzéseket vezetünk be, amiket egymás után kell elvégezni, ugyanakkor egy függvénybe kerülnek, amitől a kód menedzselhetetlenné válik.

Tervminták

Felelősséglánc (Chain of Responsibility)

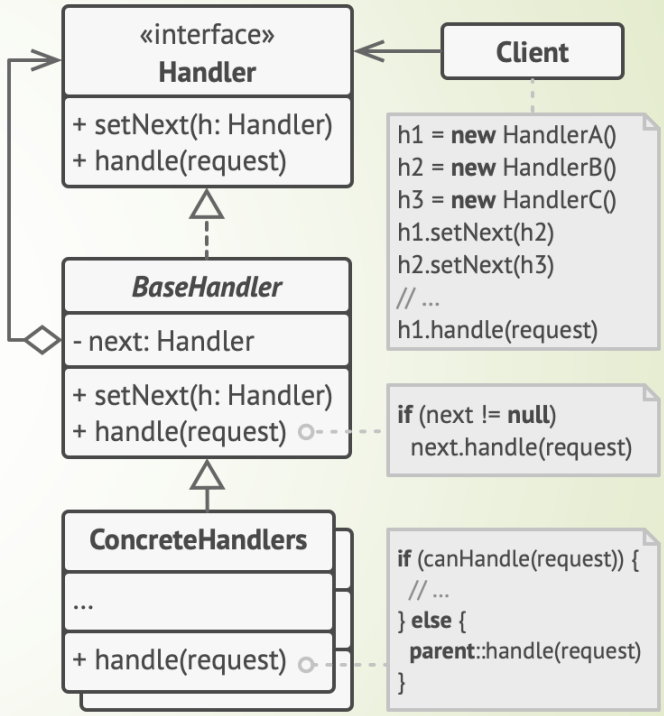
- Megoldás
  - Minden ellenőrzést tegyünk külön osztályba, egyetlen metódussal, amely elvégzi az ellenőrzést. A kérést az adatokkal együtt továbbítjuk ehhez a metódushoz.
  - A minta ezeket a kezelőket láncba kapcsolja, vagyis minden kezelőnek van egy mezője, amely a lánc következő kezelőjére hivatkozik.
  - A kérelem feldolgozása mellett a kezelők továbbítják a kérelmet a lánc mentén, ami addig halad a láncon, amíg minden kezelőnek esélye van feldolgozni.
- A kezelő dönthet úgy, hogy nem továbbítja a kérelmet.



Tervminták

Felelősséglánc (Chain of Responsibility)

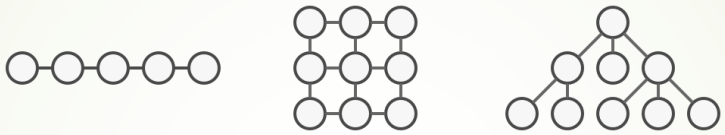
- A konkrét kezelőket ugyanabból a kezelőből származtatjuk, azonban más-más köztes lépéseket végeznek.
- Példányosításkor minden kezelőnek beállítjuk a rákövetkezőjét.
- Használatkor csak a „legkülső” kezelő metódusát kell meghívnunk, amely elindítja a végrehajtási láncot.



Tervminták

Bejáró (Iterator)

- Segítségével anélkül járhatjuk be egy gyűjtemény elemeit, hogy kitérnének a gyűjtemény vagy elemeinek ábrázolásra (lista, verem, fa stb.).

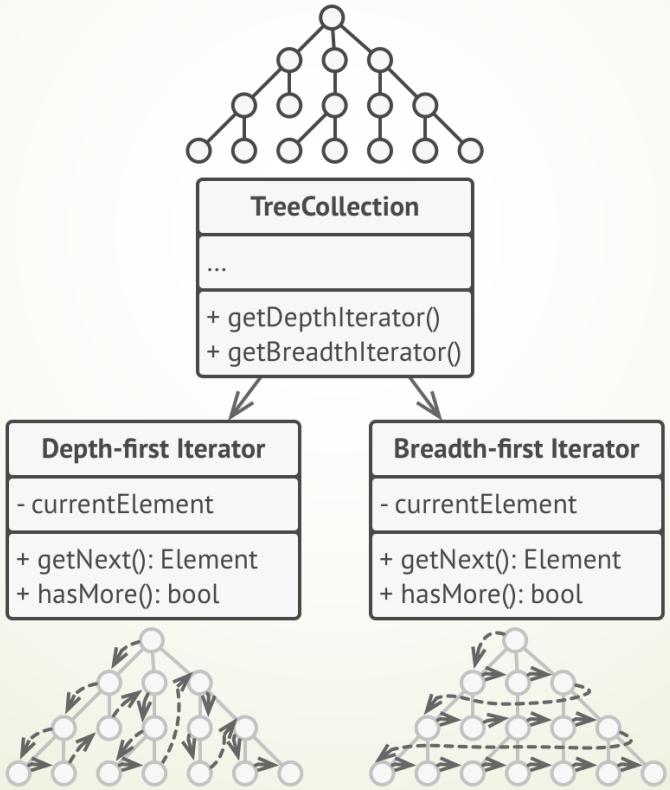


- Alap probléma
  - Milyen bejárást alkalmazzunk, ha minden adatszerkezetnek meg van a maga sajátos bejárási módja?
  - Az algoritmusok esetlegesen más-más bejárást részesítenek előnyben egy adott adatszerkezetnél.
  - Célszerűtlen akár az adatszerkezetbe, akár egy algoritmusba bedrótozni ezért a bejárást.

Tervminták

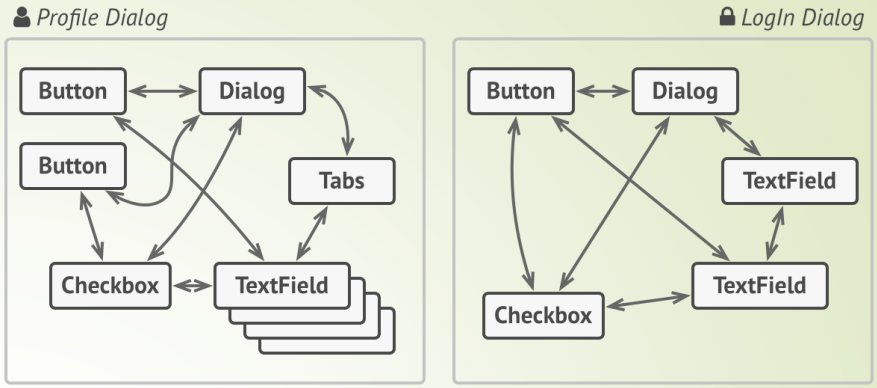
Bejáró (Iterator)

- Készítsünk egy bejáró osztályt az adatszerkezethez, ahol a minden leszármazott egy konkrét bejárási módot ad meg.



Tervminták

Közvetítő (Mediator)

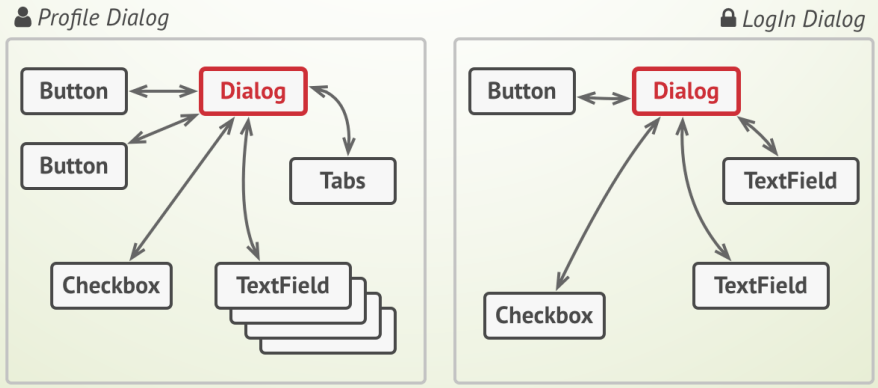


- A közvetítő az objektumok közötti függőségeket csökkenti.
- Korlátozza az objektumok közvetlen kommunikációját, és közvetítő objektumon keresztül kényszeríti őket együttműködésre.
- Probléma
  - Tegyük fel, egy párbeszédablak különböző szövegmezőkből, jelölőnégyzetekből stb. áll. Ahogy az alkalmazás fejlődik, ezen elemek kapcsolata kaotikussá válhat.
  - Egyes elemek kölcsönhatásba léphetnek másokkal, pl. bejelöljük, hogy „Van kutyám”, mire egy rejtett szövegmező jelenik meg a kutya nevének beviteléhez.
  - Az UI elemeinek változásai akár több más elemet is befolyásolhatnak.
  - Ha ez a logika a vezérlőelemek kódjában valósul meg, az megnehezíti ezen elemek osztályainak újra felhasználását.

Tervminták

Közvetítő (Mediator)

- Megoldás
  - Szüntessük meg a közvetlen kommunikációt az összetevők között, amelyeket függetleníteni szeretnénk.
  - Az összetevők közvetetten működjenek együtt, bevonva egy speciális közvetítő objektumot, amely átirányítja a hívásokat a megfelelő összetevőkre.
  - Végeredményként az alkotóelemek így csak egyetlen közvetítői osztálytól fognak függni ahelyett, hogy tucatnyi kollégájukhoz kapcsolódnának.



Tervminták

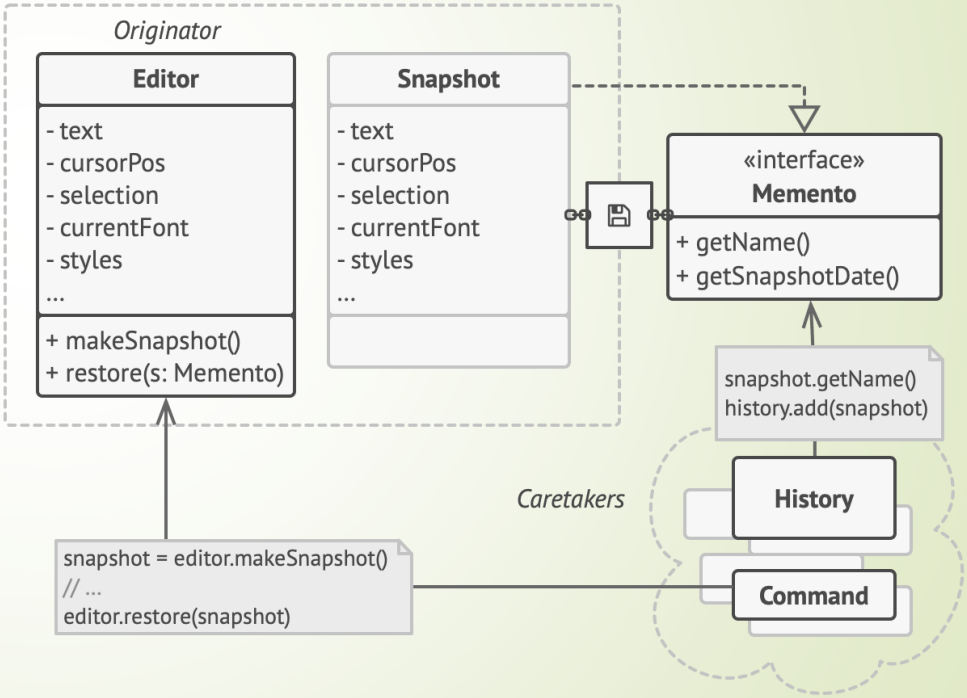
Emlékeztető (Memento)

- Lehetővé teszi az objektum előző állapotának mentését és visszaállítását anélkül, hogy közben feltárná az objektum megvalósításának részleteit.
- Probléma
  - Pl. egy szövegszerkesztő esetében több osztály objektumainak az állapotát szeretnénk rögzíteni, hogy visszavonási pontokat készíthessünk.
  - A szövegszerkesztő minden változtatás előtt pillanatképet készít, azonban ehhez minden objektumnak az állapotához hozzá kell férnie.
  - A hozzáférés miatt a privát adattagok vagy publikussá válnak, vagy azzal egyenértékű módon setter/getter metódusok jönnek létre, ami sérti az adatelrejtést.

Tervminták

Emlékeztető (Memento)

- Megoldásként hozzunk létre az osztályunkhoz egy Snapshot osztályt, amely implementálja a Memento interfészt.
- Az interfész csak meta adatokat szolgáltat, mint pl. művelet neve, készítésének időpontja.
- Az osztály objektumai a pillanatképeik adataihoz hozzáférhetnek, mások viszont csak a képek interfészét fogják látni.
- Pillanatkép készítését / betöltését az osztálynál tudjuk végezni.



Tervminták

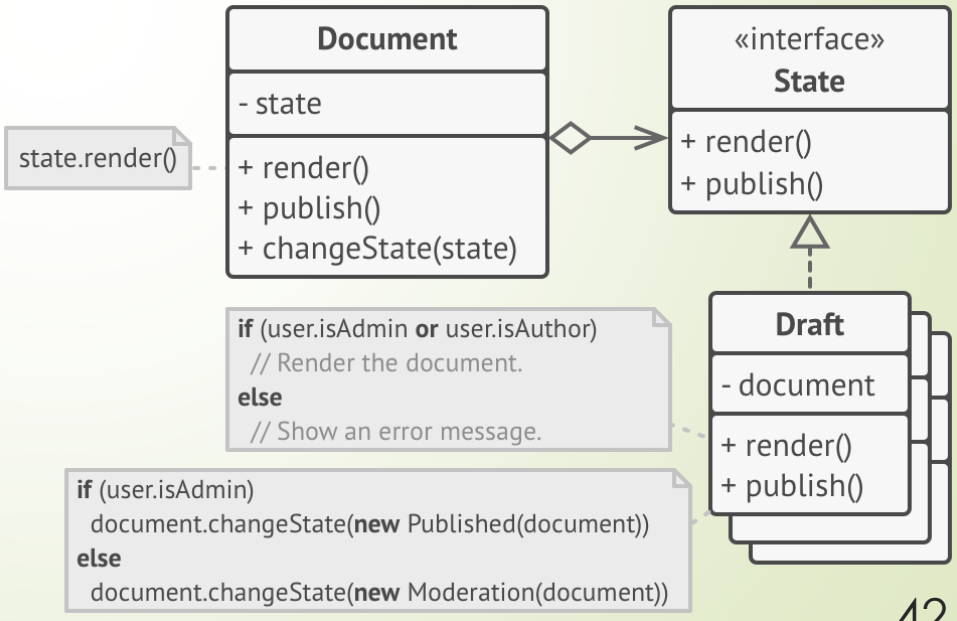
Állapot (State)

- Segítségével az objektum úgy változtathatja meg viselkedését, belső állapota függvényében, mintha az osztályát változtatná meg.
- A megoldandó probléma egy véges állapotú automatán keresztül érthető meg igazán. Adott egy Dokumentum osztály, melynek objektuma a következő három állapot egyikében lehet: Piszkozat, Moderálás vagy Közzétett.
- A közzététel módszere eltérő minden állapotban:
  - Piszkozat: a dokumentum moderálásra kerül.
  - Moderálás: a dokumentumot nyilvánosságra hozza, de csak akkor, ha az aktuális felhasználó rendszergazda.
  - Közzétett: nem tesz semmit.
- Új állapot bevezetése sérti a SOLID elvben az SRP-t és OCP-t, valamint monolitikus és nehezen karbantartható kódot ad.

Tervminták

Állapot (State)

- Megoldásként hozzunk létre új osztályokat az objektum minden lehetséges állapotához, és az összes állapot-specifikus viselkedést tegyük át ezekbe az osztályokba.
- Ahelyett, hogy minden viselkedést önmagában hajtana végre az eredeti objektum, az úgynevezett kontextus az aktuális állapotát ábrázoló állapotobjektumok egyikére való hivatkozást tárolja, és az állapottal kapcsolatos összes munkát az adott objektumra delegálja.
- A kontextus állapot váltásához kicseréljük az aktív állapotobjektumot egy másik objektumra, amely az új állapotot képviseli.



Tervminták

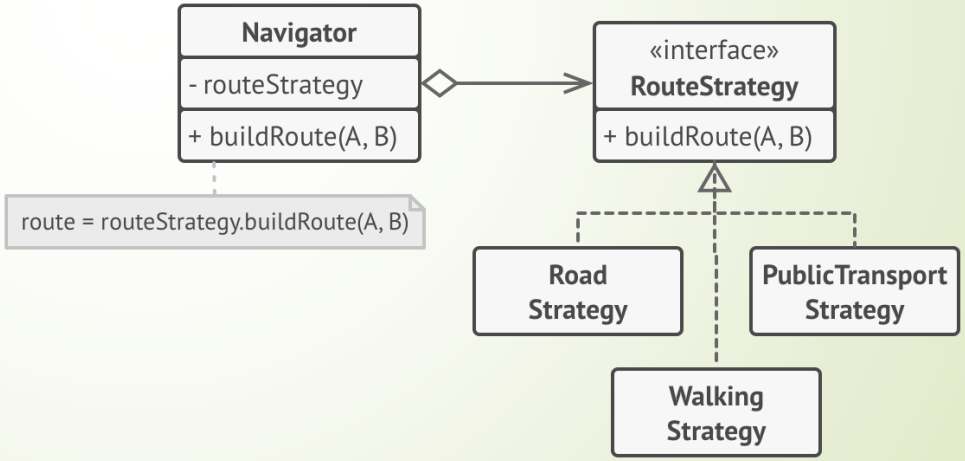
Stratégia (Strategy)

- Lehetővé teszi az algoritmusok családjának meghatározását, mindegyik külön osztályba sorolását, és objektumaik kicserélhetőségét.
- Probléma
  - Adott egy útvonaltervező alkalmazás, mely kezdetben csak autósoknak készül, melynek algoritmusát a Navigator osztályban helyezik el.
  - Később új algoritmusokkal bővítik ki gyalogosokra, majd kerékpárosokra, és akár turistákra is.
  - A Navigator osztály mérete egyre csak duplázódik, merge conflict-ok jelentkeznek a csapatmunka miatt, és az osztály karbantartása rémálommá válik, mert valahányszor egy algoritmushoz hozzányúlnak, a többit is érinti.

Tervminták

Stratégia (Strategy)

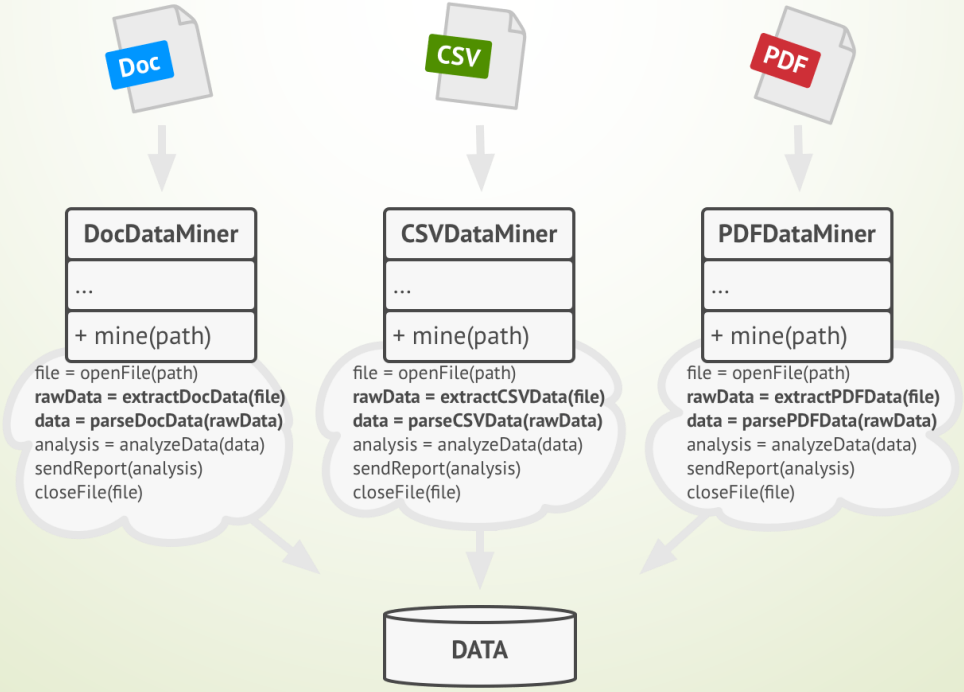
- Vegyük azt az osztályt (kontextus), mely különféle módokon végez valamit, és az algoritmusait tegyük külön osztályokba (stratégiákba). Tároljunk el egy stratégiára való hivatkozást, és a munkát ennek a stratégia objektumnak delegáljuk majd.
- A kontextus nem felelős a megfelelő algoritmus kiválasztásáért, az ügyfél adja át neki a kívánt stratégiát.
- A kontextus alig tud a stratégiákról, velük ugyanazon általános felületen keresztül működik, ami csak egyetlen módszert fed fel.



Tervminták

Sablonfüggvény (Template method)

- Meghatározza egy algoritmus vázát a szuperosztályban, de lehetővé teszi, hogy az alosztályok felülbírálják az algoritmus egyes lépéseit anélkül, hogy szerkezetét megváltoztatnák.
- Probléma (kódismétlés)
  - Több metódusban / osztályban is definiálunk egy algoritmust úgy, hogy csak bizonyos részei térnek el.



Tervminták

Sablonfüggvény (Template method)

- Megoldás:
  - bontsuk az algoritmust lépések sorozatára,
  - a lépéseket alakítsuk metódusokká,
  - az algoritmust „sablon metódus”-ként írjuk meg, azaz a lépései lehetnek absztraktak, vagy végrehajthatnak bizonyos alapértelmezett lépéseket.
- Az algoritmus használatához a kliensnek elő kell állítania saját alosztályát, implementálni az absztrakt lépéseket, és szükség esetén felülbírálni az opcionális lépéseket (de nem a sablon módszerrel!)