Návrhové vzory (design patterns)

- při řešení problémů se často dá vycházet z ověřeného způsobu řešení
- řeší, **jak** se problém řeší v objektově orientovaném prostředí -> UML modely návrhu

- **GoF patterns** - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. (český překlad: Návrh programů pomocí vzorů. Grada 2003.)

- **J2EE patterns**
  - http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

- **.NET patterns (většinou GoF patterns pro C#)**
  - http://www.dofactory.com/

Další literatura

- *B. Eckel: Thinking in Patterns.* Výklad o návrhových vzorech založený na jazyce Java. Předběžnou verzi lze získat ve formátu PDF na adrese http://www.mindview.net/Books

- *J. W. Cooper: Java Design Patterns.* Addison-Wesley 2000. ISBN 0-201-48539-7. 330 stran. Návrhové vzory založené na jazyce Java.

- *J. W. Cooper: The design patterns JAVA companion* (el.verze zdarma)

- *J. W. Cooper: Design patterns and Object oriented programming in Visual Basic 6 and VB.NET* (el. verze zdarma)

- *I. Kraval: Design Patterns v OOP se zaměřením na Javu, C# a Delphi*, http://www.objects.cz (el. verze po žádosti zdarma)

Návrhový vzor – obecně

- Popisuje zobecnění konkrétních situací (přirovnání k matematickému vzorci) -> problém s úrovní abstrakce…a pochopením…vzor se vysvětluje na konkrétních použitích
- proměnné vzorce - účastníci vzoru
- motiv – známé použití vzoru -> srovnání motivu a problému naší aplikace -> schopnost vzor použít
- smysl – identifikace vzoru – velmi stručné vystižení problému a řešení
- kombinace vzorů

Katalog vzorů

- Vyhledávání podle smyslu vzoru
- Vyhledávání podle kritérií – klasifikace vzoru:
    - Podle účelu (purpose)
    - Rozsahu platnosti (scope)
    - Důležitější rozdělení dle účelu

Klasifikace vzorů podle účelu

- Pro tvorbu objektů – tvořivé vzory - creational patterns – řeší tvorbu objektů s tím související problematiku (např. odstínění tříd)
- Pro struktury – strukturální vzory - structural patterns – řeší opakující se struktury objektů
- Řešící chování – vzory chování - behavioral patterns

Klasifikace vzorů podle rozsahu platnosti

- S rozsahem platnosti tříd – class patterns – popisují interakce mezi třídami (dědění, agregace,…)
- S rozsahem platnosti objektů – object patterns – popisují vztahy mezi objekty, většinou problémy typu změna „něčeho v runtime“, flexibilnější než třídní vzory

Skladba vzoru

- Název a klasifikace
- Alias – jiné názvy pro vzor
- Smysl – většinou jedna věta
- Motiv – demonstrace problému
- Aplikovatelnost – podmínky vedoucí k použití vzoru
- Struktura vzoru – popisuje vzor jako takový
- Účastníci vzoru – role vzoru s účastníky vzoru, popis jejich postavení a účasti ve vzoru
- Spolupráce – interakce mezi účastníky vzoru, tj. zasílání zpráv, scénáře,…
- Důsledky – výhody a nevýhody použití vzoru
- Implementace – „historicky“ Smalltalk, C++, dnes Java, .NET, Delphi
- Příklad vzoru – často spojeno s částí implementace
- Známé použití – v ČR není známo
- Související vzory – řešení často kombinace vzorů, zástupné vzory

Myšlenkový základ vzorů

- Dva pohledy: klient-rozhraní a rozhraní-implementace, pro návrhový vzor důležitý první pohled -> změnu implementace klient nepozná -> nebudeme se zaměřovat na implementaci
- Polymorfismus
- Posílání zpráv a vyvolávání metod
- Terminologie – zpráva, metoda, operace
- Abstraktní třída
- Abstraktní operace
- Čistá abstraktní třída
- Rozhraní

Iterátor

- jeden z nejjednodušších a nejčastěji používaných návrhových vzorů

- smysl: jednoduchý a sekvenční přístup ke složité struktuře objektů

- motiv: agregovaný objekt (jako je např. seznam) by měl umožňovat procházení, aniž by k tomu uživatel musel znát vnitřní strukturu agregátu (která se může změnit) – uživatel = zjednodušený pohled na složitou strukturu a její snadné ovládání

- iterátor udržuje informaci o aktuálním objektu v agregátu, umí se posunout na následující prvek agregátu a prvek poskytnout uživateli (anebo také vrátit první prvek seznamu, poslední prvek seznamu apod.)

- moderní OO programovací jazyky obsahují podporu iterátorů
- př. Java - rozhraní Iterator:

```
public interface Iterator
 {
    boolean hasNext(); // Vrací "true" pokud jsou další prvky.
    Object next();     // Vrací další prvek agregátu.
 }
```

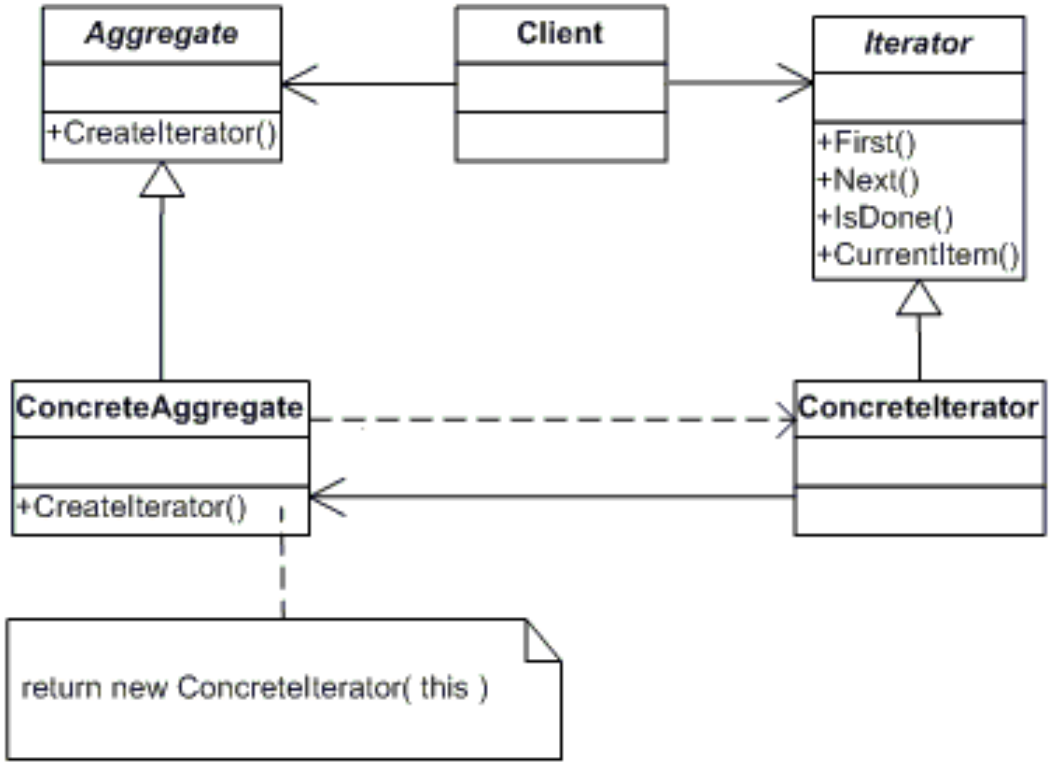
- v Javě poskytují iterátor třídy Set, List, Map, atd.; lze psát např.

```
for (Iterator ii = c.iterator(); ii.hasNext();) {
    System.out.println(ii.next());
}
```

- ale i jednoduché nabízení vlastních složitých struktur = předřazení objektu s rozhraním vzoru Iterator (možné použít rozhraní Iterator daného prostředí)

- tvorba iterátoru – nejlépe od dané struktury… např. operace CreateIterator()

Struktura vzoru



Použití vzoru:

- skrytí složité struktury složeného objektu
- polymorfní průchod různými typy struktur
- násobný přístup ke složené struktuře a další

Související vzory:

- vzor Composite se často doplňuje o vzor Iterator
- tvorba Iteratoru = vzor Factory Metod (Prototype)
- robustní Iterator – vzor Observer

Reification (Zkonkrétnění)

- obecný termín s významem "zkonkrétnit něco",
- v objektovém návrhu znamená "udělat objekt z něčeho, co na první pohled jako objekt nevypadá"
- není návrhový vzor, ale spíše meta-vzor, ze kterého vznikají konkrétní návrhové vzory

- když o nějaké aktivitě nebo chování začínáme uvažovat jako o metodě, ale časem zjistíme, že má spoustu komplikovaných variant - pak je aktivita vhodným kandidátem na třídu, varianty chování jsou kandidáty na podtřídy

- např. vyhledávání: na první pohled vypadá jako úloha pro metodu, časem zjistíme, že vyhledávání lze specifikovat mnoha způsoby, vytvoříme třídu Search, jejíž instanční proměnné udržují informace o kritériích vyhledávání

- objekt třídy Search se nám hodí, pokud budeme chtít vyhledávat podle trochu odlišných kritérií
- pokud bychom chtěli prohledávat různá data, můžeme vytvořit několik podtříd třídy Search
- nejdůležitější metoda se bude jmenovat např. "execute" nebo "doIt" - bude reimplementována polymorficky v podtřídách

- třída může vzniknout i z něčeho většího, než je metoda, např. Jacobson říká, že vhodným kandidátem na objekt je celý případ užití

- pokud provedeme reifikaci aktivity, říkáme vzniklému objektu "řídící objekt" (control object)

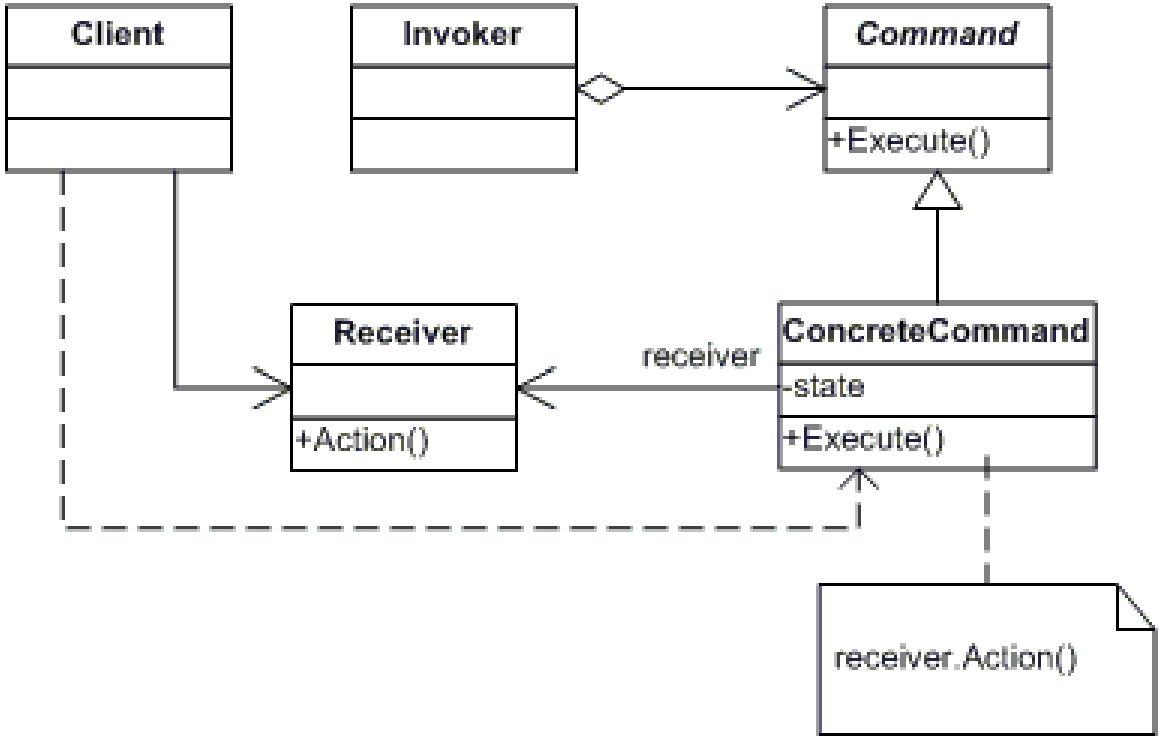
Příkaz (Command)

- smysl: zapouzdřuje požadavek do podoby objektu -> požadavek je proměnná (seznamy požadavků, parametrizace požadavků apod.)

- motiv:
- dynamické skládání prvků GUI
- flexibilní zpracování návratové hodnoty
- a další

- klient volá operaci, ta vrací hodnotu, máme se rozhodnout, jak na ni budeme reagovat (řešení – switch - funkční, ne flexibilní), vytvoříme abstraktní třídu "Command" s operací "execute"
- pro každé zpracování návratové hodnoty vytvoříme konkrétní podtřídu, která bude operaci implementuje, výběr převedeme na výběr instancí z těchto tříd
- klient je odstíněn od konkrétního volání operace objektů reagujících na návratovou hodnotu

Struktura vzoru



Použití vzoru

- flexibilní přidávání podtříd Command, s objekty se pracuje jako s proměnnými
- oddělení klienta od realizace požadavku, klientovi můžeme vyměnit dynamicky „jeho command“
- Composite + Command = rekurzivní MacroCommand
- v procedurálním programování – volání funkce přes pointer
- samotný Command nemusí jenom delegovat, ale může i sám něco vykonávat
- kopie vykonaných objektů Command v seznamu + operace Undo() – historie

Související vzory

- Composite – MacroCommand
- Memento – Undo()
- Prototyp – seznam historie

Memento

- smysl: klient získá stav objektu, aby bylo možné vrátit objekt do původního stavu, není porušeno zapouzdření

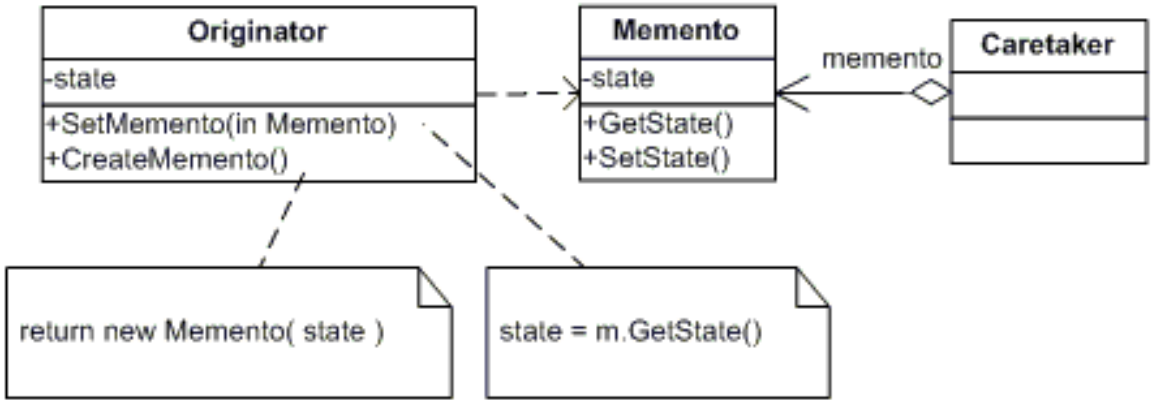
- motiv: zavoláme operaci objektu, změní se jeho stav a my chceme vrátit stav do stavu původního, nelze se spoléhat na inverzní operace, budeme si pamatovat výchozí stav

- klient používá objekt třídy Element (Originator v obrázku), potřebuje uchovat současný stav, zavolá operaci createMemento(), vytvoří se objekt Memento, překopíruje se do něho stav objektu Element, klient si uchová objekt Memento, při návratu klient zavolá operaci setMemento(Memento memento) a předá mu objekt memento, který obdržel při předchozím volání operace createMemento()

- klient ví, kdy má memento použít (vytvořit, vrátit zpět, odstranit), ale neví, co v něm je

- element ví, jak se memento tvoří a jak se podle něj rekonstruuje stav, ale neví, kdo a kdy bude o memento žádat, nabízí pouze vytvoření a vydání objektu mementa

Struktura vzoru



Použití vzoru

- rozdělování rolí, zjednodušení problému
- kombinace vzorů Command a Memento – Command podporuje operaci Undo(), seznam vykonaných Commandů, mělké kopie (každá kopie stejné instance Commandu vlastní memento, ale referenci na objekt)

Související vzory

- Command viz výše
- Iterator (průchod historií Commandů a Mement)

Abstraktní továrna (Abstract factory)

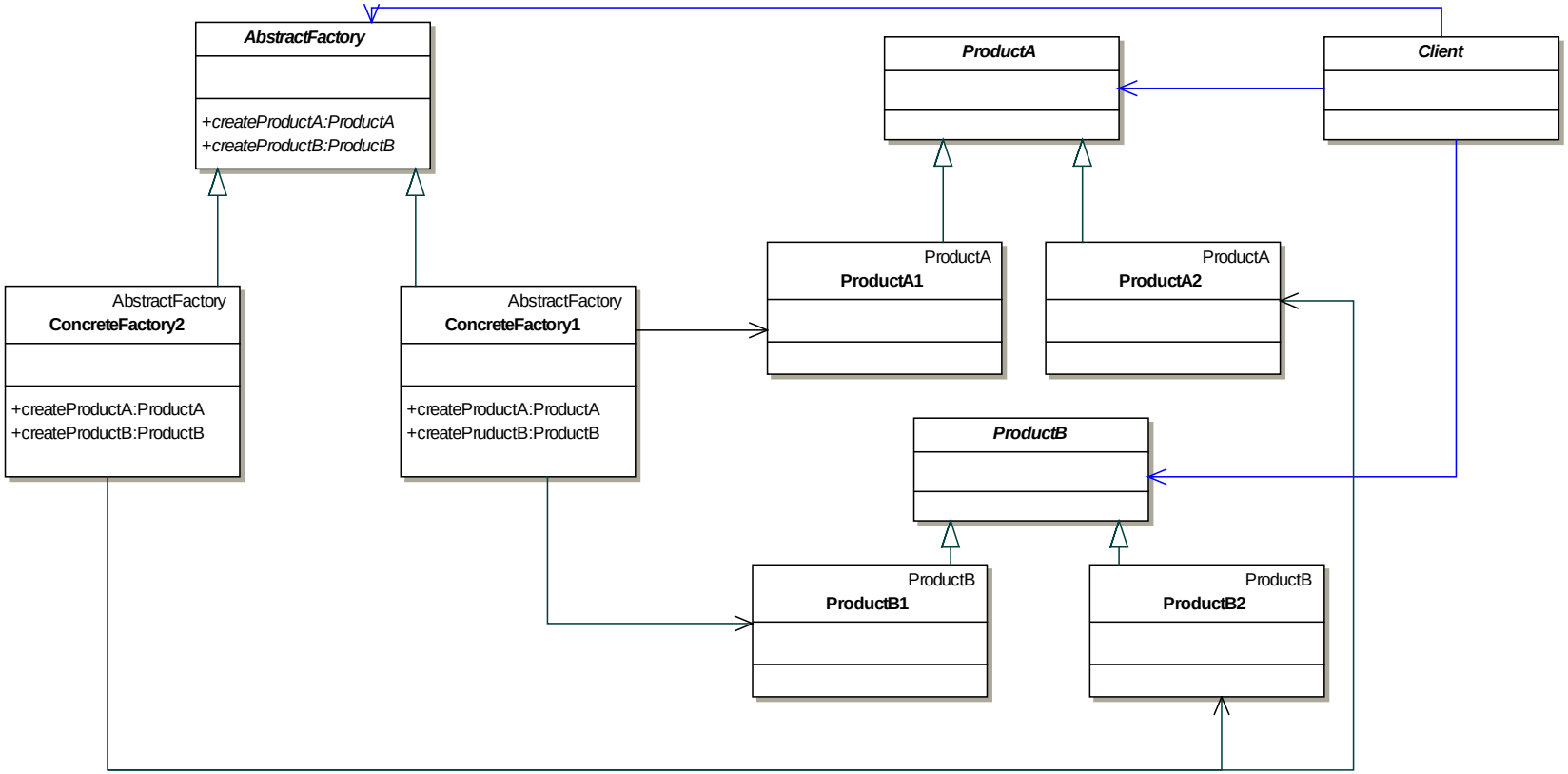
- klasifikace: Tvořivý, Objekt
- smysl: Zavedení rozhraní pro tvorbu rodin (řad) příbuzných nebo závislých objektů, aniž by se musely specifikovat konkrétní třídy, klient je izolován od volání těchto tříd
- alias: Kit (Souprava)

- motiv:
  - dva (více) standardů vzhledu grafického uživatelského rozhraní, označme St1 a St2,… lze zavést buď St1, nebo St2, nebo…
  - jiný vzhled definuje odlišná zobrazení a chování prvků GUI - oken, posuvníků, atd.
  - aplikace může přepínat mezi standardy
  - definována abstraktní třída FactorySt – rozhraní pro tvorbu prvků GUI, konkrétní třídy FactorySt1 a FactorySt2

- definování abstraktní třídy pro prvky GUI - pro okna, posuvníky,…tj. např. pro okna WindowSt1 a WindowSt2 abstraktní třída WindowSt, pro posuvníky ScrollBarSt1, ScrollBarSt2, abstraktní třída ScrollBarSt
- FactorySt – abstraktní operace pro vytvoření prvku GUI např. CreateWindow() a CreateScrollBar()

- FactorySt1 a FactorySt2 - konkrétní implementace rozhraní FactorySt, vracejí odpovídající typy prvku GUI, tj. např. ScrollbarSt1 nebo ScrollBarSt2
- Volba prvků GUI je převedena na volbu objektu továrny FactorySt1, nebo FactorySt2,…
- klient vytváří prvky GUI pomocí rozhraní FactorySt, zavazuje se pouze k rozhraní definovanému abstraktní třídou, nevolá konstruktory objektů, pouze metody rozhraní
- objekty FactorySt, tj. např. objekt FactorySt1 a objekt FactorySt2 umístíme do kolekce, nazveme např. ManagerFactory, tyto objekty se přidávají spolu s klíčem

Struktura vzoru



Created by Borland® Together® Designer Community Edition

- využití rolí – nahrazení prvků GUI pojmem Product a továrny do stromu s předkem AbstractFactory a rolemi potomků ConcreteFactory
- AbstractFactory vytváří rozhraní, klient volá metody rozhraní, žádné konstruktory

Použití vzoru

- odstínění volání konstruktoru, tj. je zabezpečena flexibilita výměny produktů
- je možné vyměnit „řadu produktů“ změnou výběru továrny
- produkty typu framework, knihovny – klient nevytváří přímo objekty, ale volá metody rozhraní různých továren
- izoluje klienta od implementace konkrétních tříd, zapouzdřuje odpovědnost a proces tvorby produktových objektů
- usnadňuje výměnu produktových řad a zabezpečuje používání produktů jedné řady

Nevýhody vzoru

- přidání produktu znamená rozšíření rozhraní AbstractFactory a implementaci tohoto rozhraní v konkrétních továrnách
    - i při malém rozdílu skupiny produktů je nutné přidat novou konkrétní továrnu – řeší vzor Prototype

tzv. extenzivní flexibilní továrna

- rozhraní = 1 metoda se vstupním parametrem identifikujícím produkt – klíč produktu
- návratová hodnota metody rozhraní abstraktní továrny musí být objekt implementující stejné rozhraní – všechny produkty musí patřit do jedné rodiny s jedním předkem

Továrna (klon abstraktní továrny)

- nevyžaduje se přepínat rodiny (existuje jen jedna) + vzor se vstupním parametrem = návrhový vzor Továrna (Factory)
- slouží k odstínění konstruktoru, volá se metoda rozhraní s parametrem ID produktu, návratová hodnota typu vrcholu stromu dědičnosti
- pomocný objekt Factory ke stromu produktů (v tomto případě strom generalizace-specializace)
- převede volání konstruktoru na volání metody rozhraní s parametrem – flexibilní
- typické použití dynamické menu – strom položek (s kořenem CPolozka), číselník položek (id, název položky), rozhraní továrna (MojeTovarna) s jedinou metodou s parametrem(vytvor(int id)), pak vždy stejný kód např.

CPolozka MojePolozka = MojeTovarna.vytvor(id);

Související vzory

- abstraktní továrna realizována vzorem tovární metoda (Factory method)
- konkurenční vzor prototype
- většinou se zavádí jako Jedináček (Singleton)

Tovární metoda (factory method)

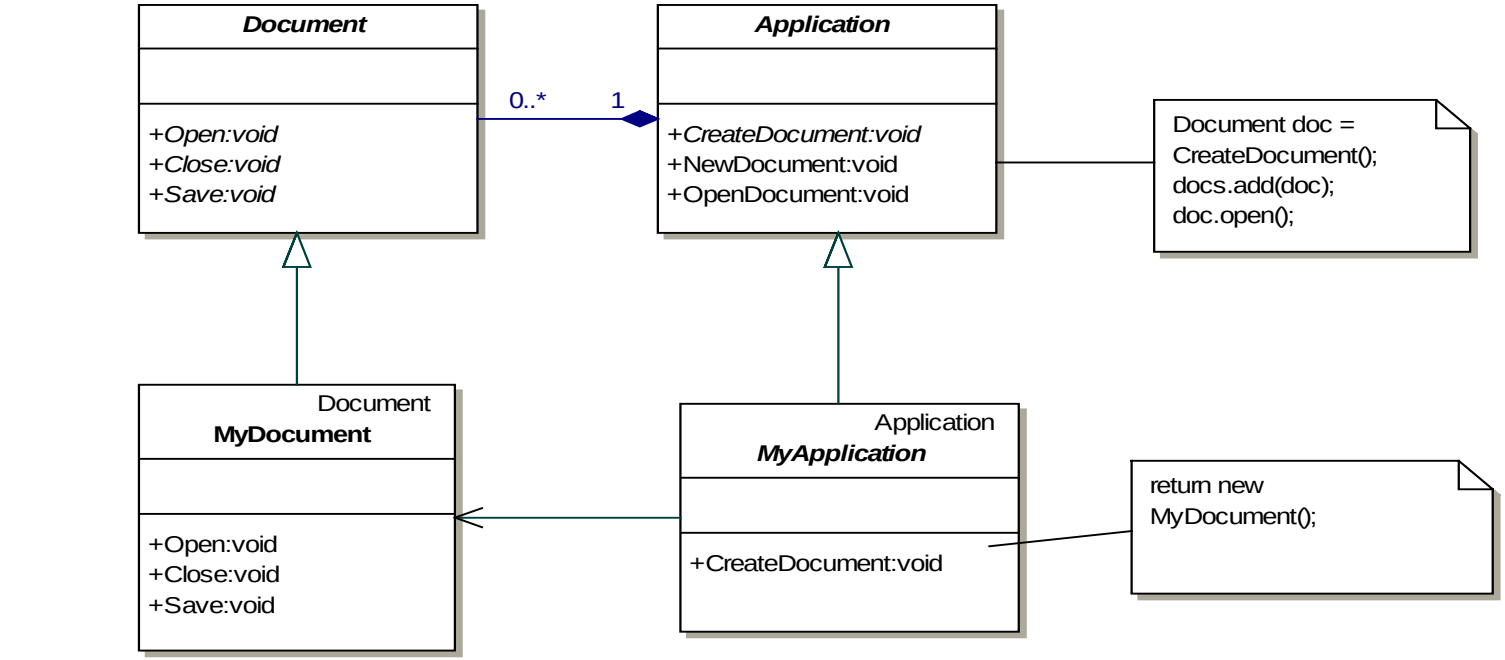
- klasifikace: Tvořivý, Třída
- smysl: Definuje metodu rozhraní pro zrod objektu na úrovni předka, ale ponechává potomkovi k rozhodnutí, jakého konkrétního typu tento objekt bude
- alias: Virtuální konstruktor

- motiv: použití ve vzoru Abstraktní továrna, rozhraní obsahuje metody CreateProduct(), návratové typy metod jsou vrcholy stromu daných produktů, každá konkrétní třída abstraktní továrny dosazuje do návratové hodnoty konkrétní typ dědice produktu

- jiný motiv:

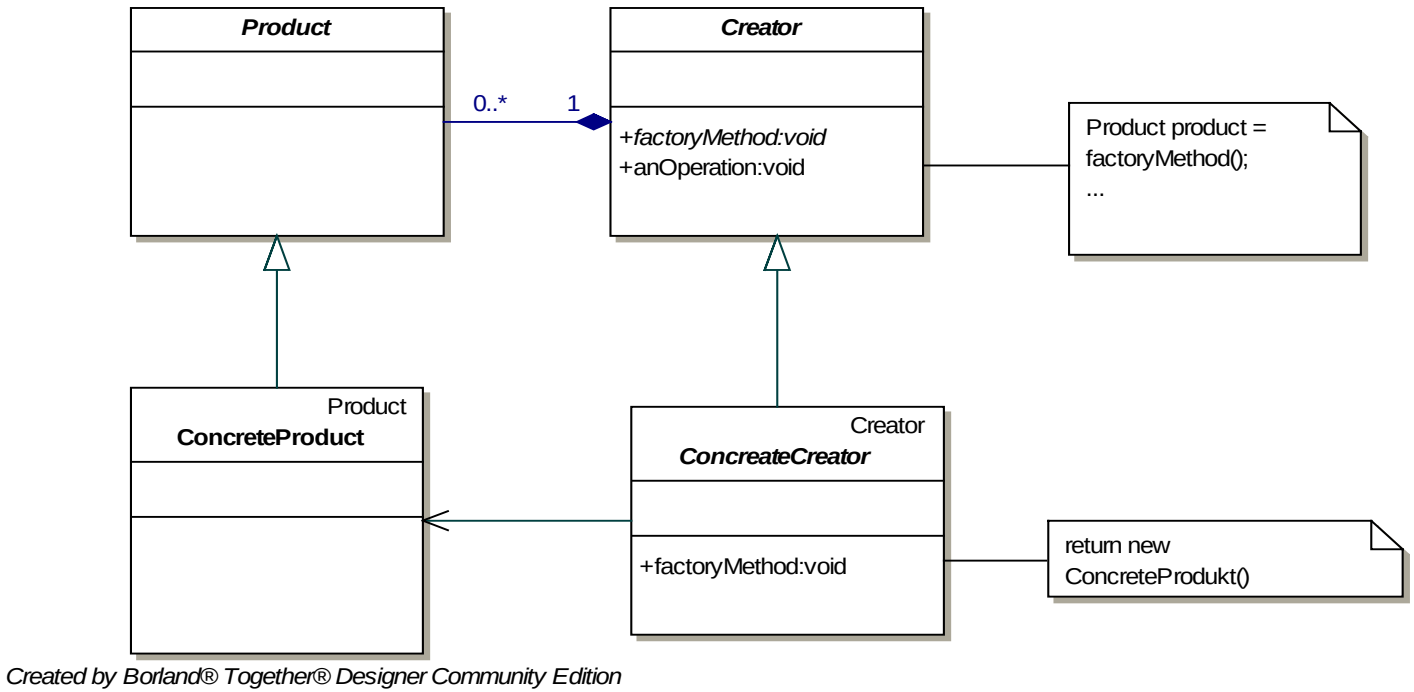
  - vývoj knihovny pro tvorbu editorů, několik možných editorů podle typů dokumentů,
  - daný typ aplikace pracuje s daným typem dokumentu, abstraktní třídy Application a Document, podtřída třídy Document je specifická dané oblasti,
  - třída Application nemůže předvídat konkrétní třídu dokumentu, pouze ví, kdy by měl být dokument vytvořen, ale neví, jaký druh dokumentu vytvořit
  - řešení: abstraktní metoda CreateDocument() předefinovaná konkrétními aplikačními podtřídami, které vytvářejí konkrétní dokumenty

Příklad s editory



Created by Borland® Together® Designer Community Edition

Struktura vzoru



Struktura

- Product – definice rozhraní objektů vytvářených tovární metodou
- ConcreteProduct – implementuje rozhraní produktu
- Creator – deklaruje tovární metodu, může definovat výchozí implementaci
- ConcreateCreator – překrývá tovární metodu, aby vrátila instanci ConcreteProduct

Použití vzoru

- scénář typu předek ví, kdy je třeba nechat vzniknout objekt, ale neví, jaký objekt, potomek ví a rozhodne, který objekt se má vytvořit
- propojení paralelních třídních hierarchií – dva paralelní stromy propojené vazbou viz motiv Aplikace a Dokument

Pozn

- třída Creator abstraktní? – může a nemusí, jedna z aplikací se svým typem dokumentu může být implicitní

  Související vzory

  – abstraktní továrna, šablona (template method), prototyp

Další příklad

- metoda openWindow() ve třídách Calculator a PhoneBook má téměř stejnou podobu:

```
-    x = new nějakéOkno(); // typ okna závisí na třídě
-    x.openWidget();

-    abstract class DesktopObject {
-
-         abstract Window getWindowInstance();
-
-         void openWindow() {
-              Window w = this.getWindowInstance();
-              w.openWidget();
-         }
-    }
```

```
class Calculator extends DesktopObject {

    Window getWindowInstance() {
        .... // vrací instanci správné třídy
    }

}
```

- getWindowInstance() je tovární metoda, protože (většinou) vyrábí instanci související třídy - nemusí ale vytvářet instanci vždy - někdy může jenom vrátit odkaz na existující instanci

Singleton (Jedináček)

- klasifikace: tvořivý, objekt
- smysl: třída má jen jednu jedinou instanci globálně viditelnou
- motiv:
    - často chceme, aby nějaký objekt byl globálně viditelný a jedinečný (objekt faktory u vzoru Abstract factory)
    - instance se umístí jako privátní statický člen, zpřístupňuje se přes statickou veřejnou metodu

Použití

- unikátnost instance
- nutnost skrytého konstruktoru

- řešení v jazyku Java:
-
```
-       public class Singleton {
-           private static final Singleton instance = new Singleton();
-
-           private Singleton() {}  /* soukromý konstruktor */
-
-           public static Singleton getInstance() /* pro získání instance */
-           {
-               return instance;
-           }
-       }
```
-
- klienti získají instanci pomocí statické tovární metody:
-
```
-     x = Singleton.getInstance();
```

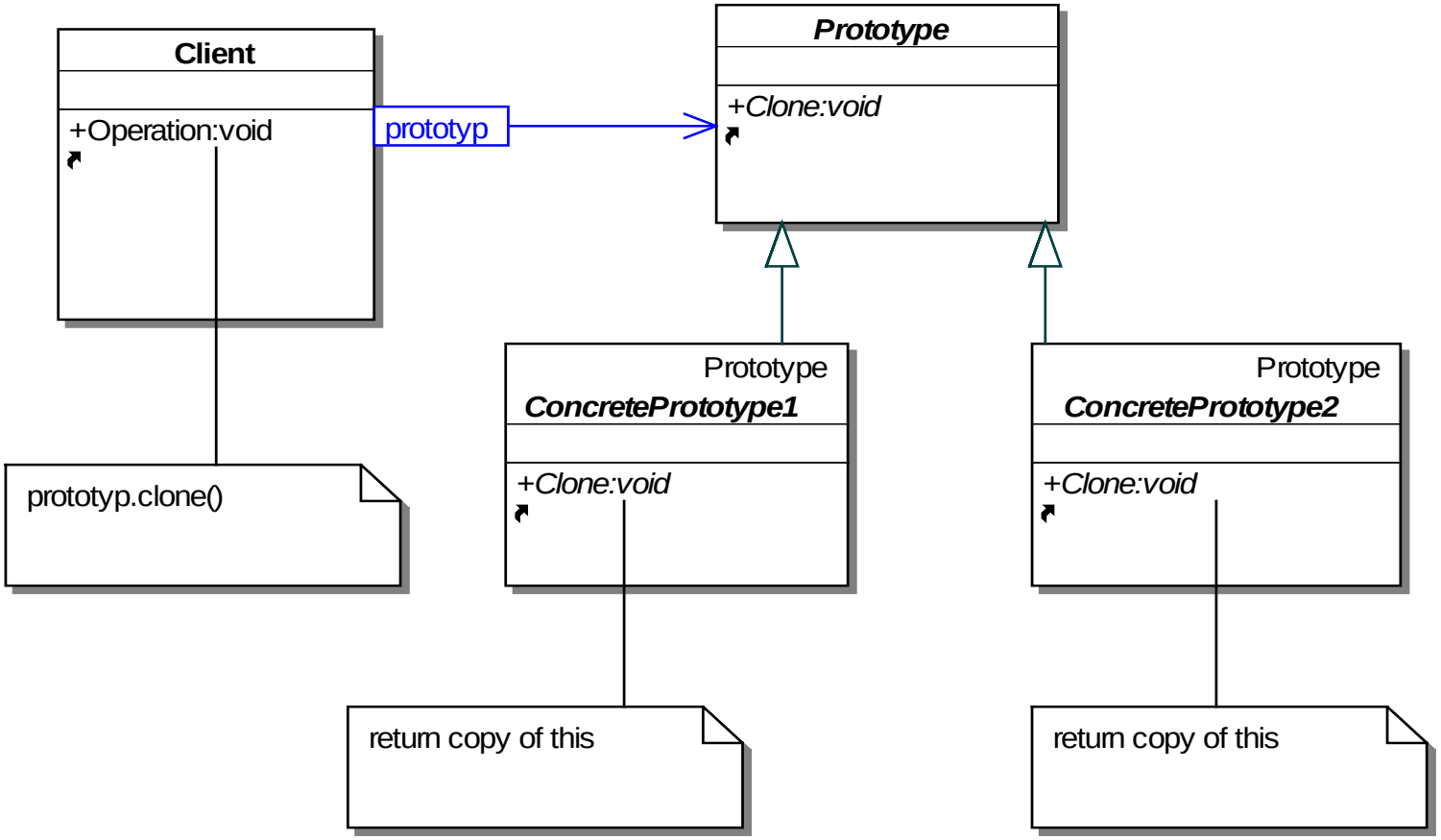
Prototyp (Prototype)

- klasifikace: Tvořivý, objekt
- smysl: Zavádí se specifické druhy objektů vznikající klonováním prototypových instancí pomocí jednotného rozhraní

- motiv: otázka vhodné parametrizace práce se třídami – výběr třídy produktu parametrem
  - částečné řešení vzor Továrna se vstupním parametrem (metoda CreateProduct() s parametrem) – nutnost přepisovat metodu i při malých změnách, seznam produktů nelze měnit dynamicky
  - výběr obrazce v aplikaci obsluhou
  - flexibilní řešení prototypem: každý obrazec ze stromu dědičnosti implementuje metodu rozhraní Clone() – po zavolání vytvoření kopie objektu
  - instance (prototyp) je spolu s klíčem umístěna do kolekce, kolekci prototypů nazveme manažer prototypů

- potřebujeme nový objekt obrazce, nevoláme konstruktor, ale žádáme kolekci přes klíč o novou instanci obrazce voláním metody Clone(), kopie instance je návratovou hodnotou
- v rámci inicializace metoda pro registraci daného prototypu v kolekci
- klient může dynamicky upravovat seznam prototypů

Struktura vzoru



Created by Borland® Together® Designer Community Edition

- Prototype – deklaruje rozhraní pro vlastní kopii
- ConcretePrototype – implementuje operaci pro vlastní kopírování
- Client – vytváří nový objekt požádáním prototypu o naklonování

Použití

- třídy, jejichž instance se mají vytvořit, jsou specifikovány za běhu
- nechceme budovat třídní hierarchie továrny souběžnou s hierarchií produktů
- instance třídy mají několik málo různých kombinací stavu, může být výhodnější vytvořit prototypy a ty klonovat

Ostatní

- přes manažera se dobře přidávají a odstraňují seznamy tříd v runtime
- vhodné použít např. při konfiguraci systému

Další příklad - grafický editor hudby

- vybírání not a jiných prvků notace graficky znázorněných a manipulace s nimi
- dva stromy dědičnosti Tool – manipulace s notacemi a Graphic – samotné notace
- sloučení dvou hierarchií dohromady – ne dědění (velký počet tříd), ale skládání objektů, tj. např. zápis noty: třída MoveTool a nový objekt typu GraphicNota jako atribut s rolí theGraphic
- volba třídy GraphicNota – vzor prototype, tj. ve třídě Graphic abstraktní metoda Clone() aneb výběr konkrétní třídy rozhraní Graphic převeden na výběr prototypické instance

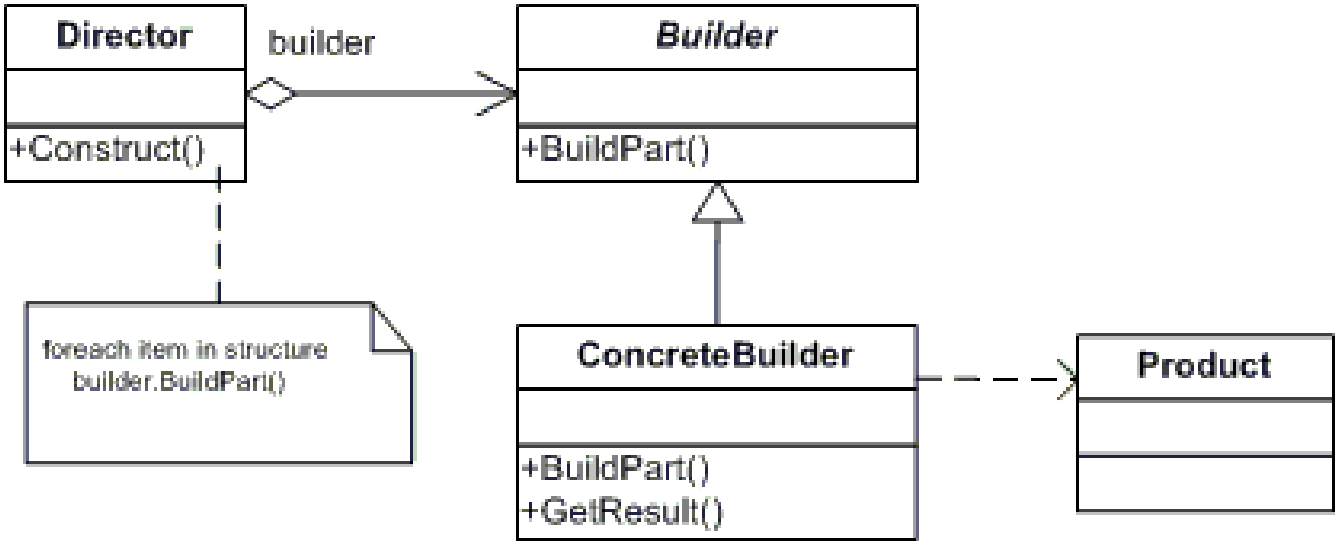
Související vzory

- Abstraktní továrna a Prototyp - - konkurenční vzory, lze je však používat i společně – abstraktní továrna může ukládat sadu prototypů
- Skladba, Dekorátor

Stavitel (Builder)

- klasifikace: tvořivý, objekt
- smysl: Odděluje konstrukci složitého objektu od jeho reprezentace, lze pak vyrábět různé reprezentace
- motiv:
  - co má/nemá klient vidět při skládání složitých objektů pomocí jeho částí
  - konverze dokumentu z jednoho typu (třeba RTF) do několika jiných typů, počet konverzí je otevřený
  - parser čte RTF dokument, klient zvaný Director převezme výsledek od parseru a posílá požadavky, co se má stavět (jiný dokument) rozhraní Builder, Builder převezme požadavky a provede se samotná konstrukce nového dokumentu
  - Director neví, jaká je reprezentace objektu za rozhraním Builder
  - Po ukončení konstrukce objektu se požádá Builder o vydání poskládaného objektu

Struktura vzoru



Builder – specifikuje rozhraní pro stavbu částí výsledného objektu
ConcreteBuilder – implementuje rozhraní Builder, kostruuje konkrétní objekt daného Produktu, zavádí operaci pro návrat výsledku
Director – zkonstruuje objekt použitím rozhraní Builder
Product – reprezentuje skládaný konstruovaný objekt

Použití

- abstract faktory – operace CreateProduct() – návratová hodnota = nový produkt, u Builderu také žádost o nový produkt, ale dělán se „per partes“
- výsledné produkty nemusí patřit do stejného stromu dědičnosti
- nový složený objekt potřebuje nový builder
- možné žádosti klienta o mezivýsledky

Související vzory

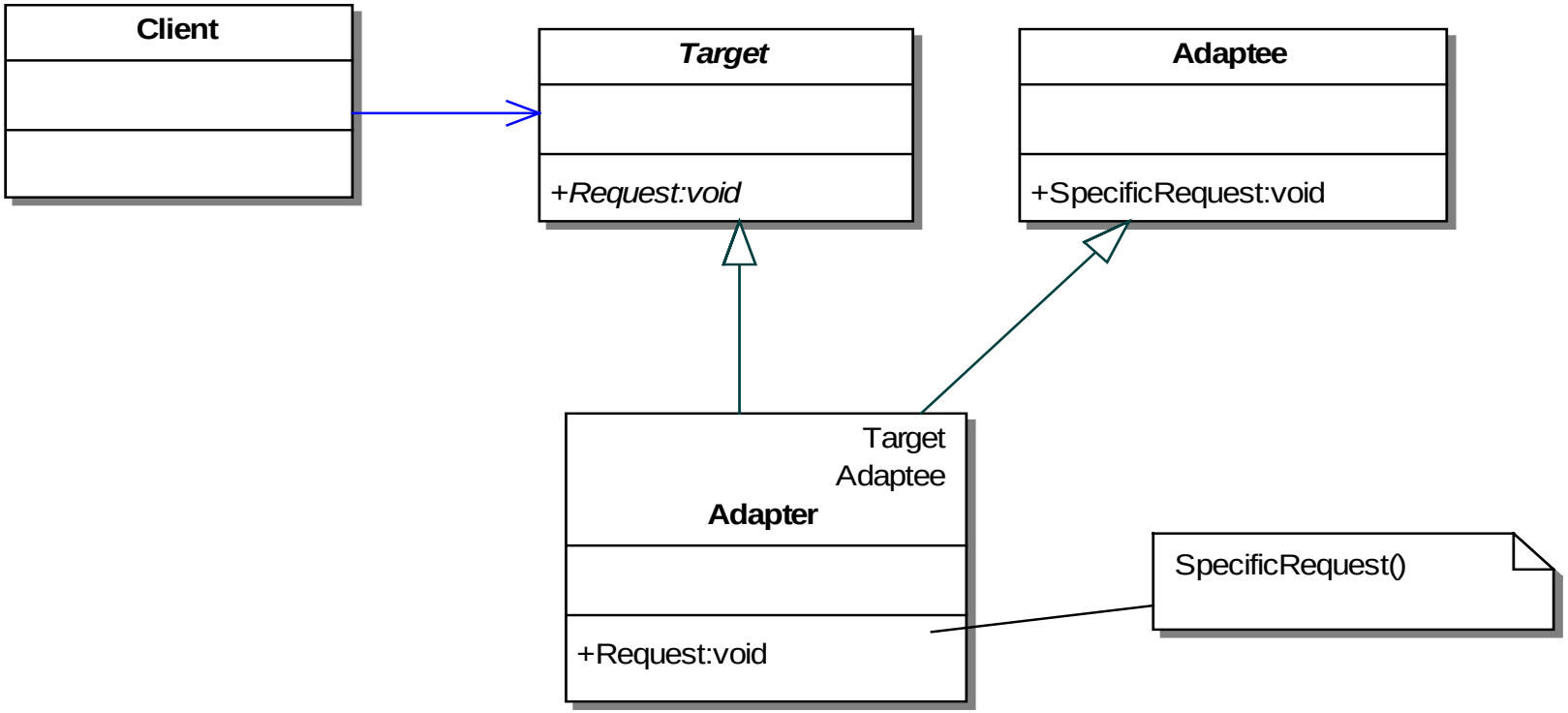
- Abstraktní továrna – také konstruuje komplexní objekty, ale najednou, vrací produkt okamžitě
- x Stavitel – postupně konstruuje objekty, vrací produkt na konci postupu
- Skladba (Composite) – často výsledek konstrukce Stavitele

Adaptér (Adapter)

- klasifikace: strukturální, objekt nebo třída
- alias: Obal (Wrapper)
- smysl:
  - objekt – objekt jako spojka mezi klientem očekávajícím určité rozhraní a neznámým rozhraním, aneb klient může používat neznámé rozhraní
  - třída – zavádí třídu spojující neznámé a očekávané rozhraní, aneb klient může používat neznámé rozhraní

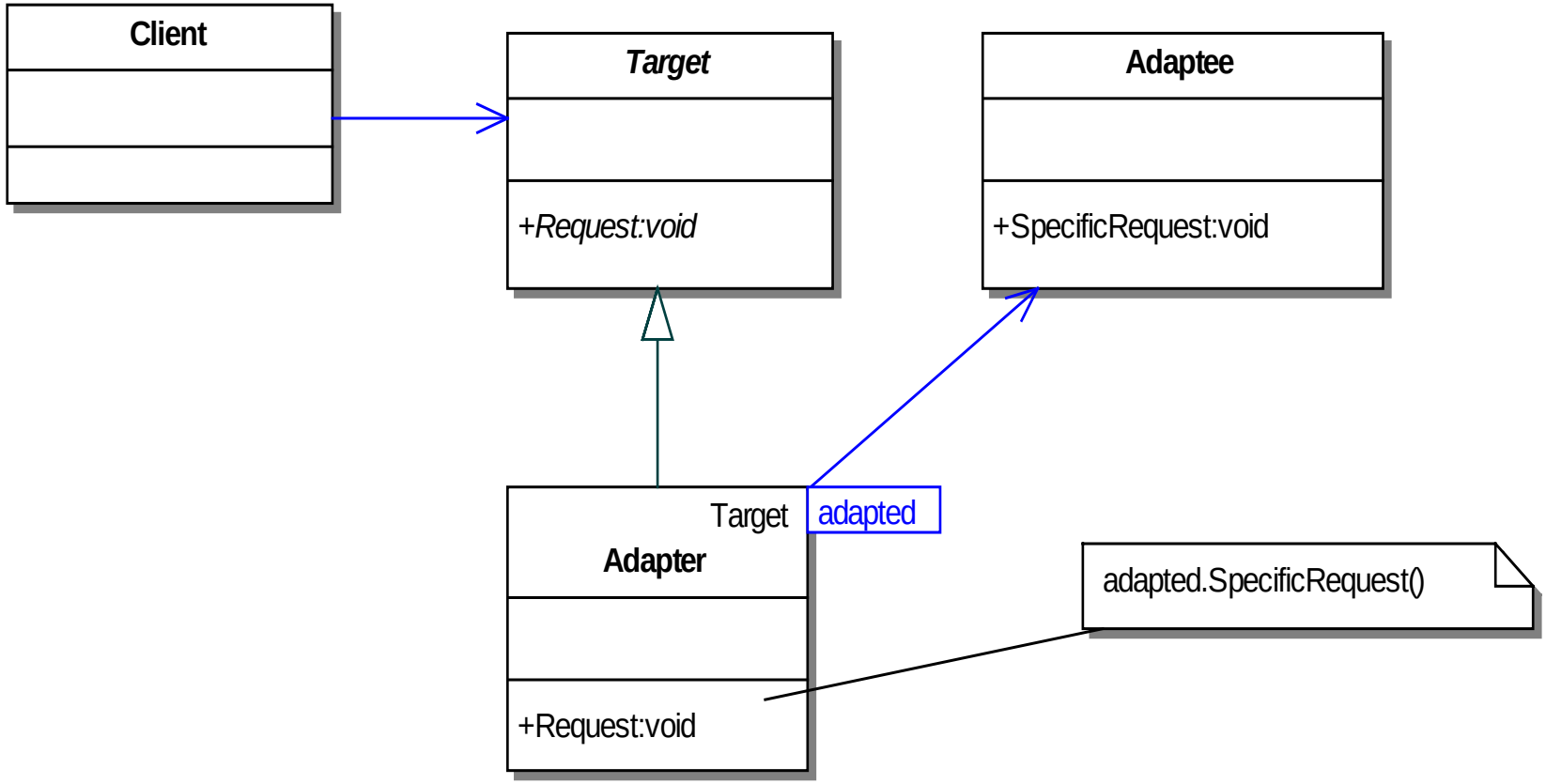
- motiv:
  - situace, kdy máme k dispozici třídu, která se nehodí do již existujících struktur, tj. např. podporuje rozhraní, které v existující struktuře vypadá jinak
  - jak zapojit cizí prvky do existující aplikace
  - adapter přizpůsobuje nekompatibilní rozhraní ke klientovi

Struktura vzoru třídní adaptér



Created by Borland® Together® Designer Community Edition

Struktura vzoru objektový adaptér



Created by Borland® Together® Designer Community Edition

Třída a objekt varianta, častěji užívaná údajně objekt varianta

- Target – definuje rozhraní používané Clientem
- Client – spolupracuje s objekty vyhovující cílovému rozhraní Target
- Adaptee – existující rozhraní, které potřebuje přizpůsobení
- Adapter – přizpůsobuje rozhraní Adaptee na rozhraní Target

- třídní varianta – používá vícenásobnou dědičnost, Adapter umí SpecificRequest(), protože je dědicem Adaptee
- objektová varianta – používá objektovou skladbu – zavolání požadavku Request() rozhraní Target se přesměruje na vložený objekt s rolí adapted

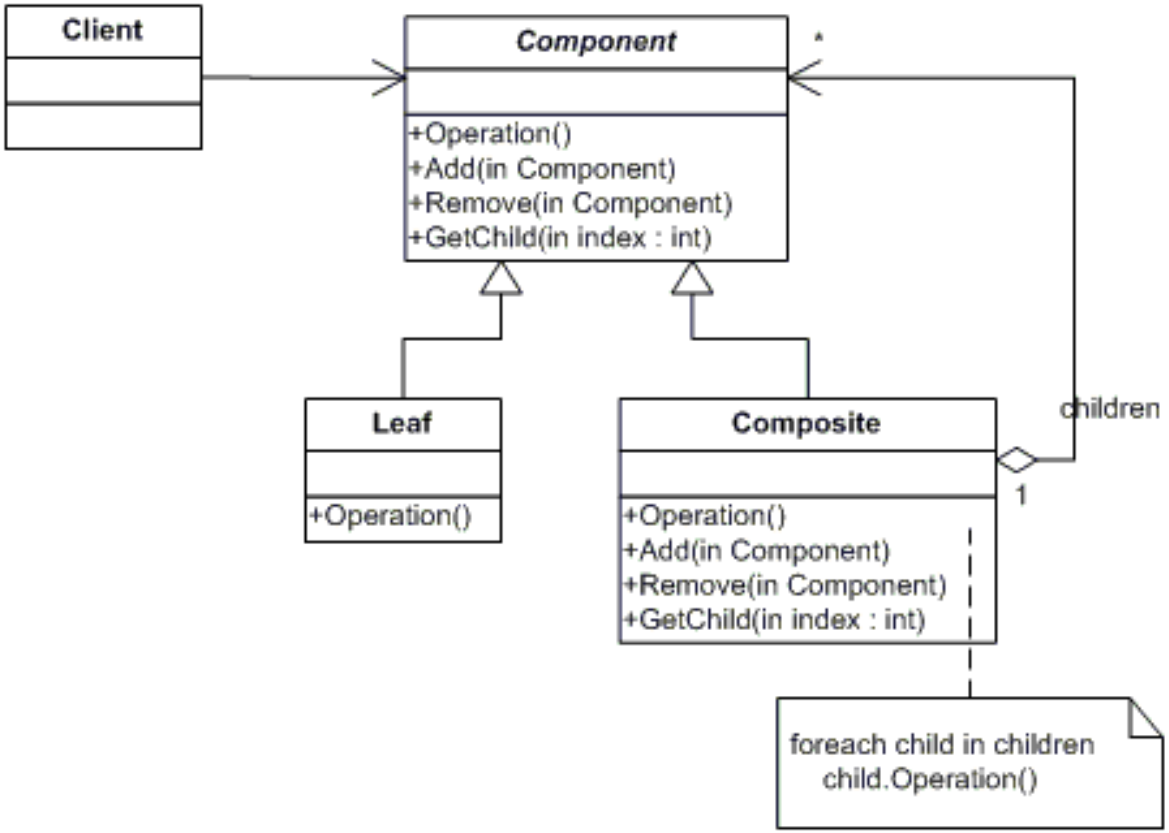
Použití

- objekt adaptér pracuje s objektem ze třídy Adaptee a s jeho dědici (stejné rozhraní), ale neumožňuje jednoduše přepsat jeho metody
- není určeno, co všechno má adaptér provádět, může operace pouze přejmenovat, může také přidat další funkcionalitu
- podoba se vzory Most (Bridge) a Zástupce (Proxy)

Skladba (Composite)

- klasifikace: strukturální, objekt
- smysl: skládá prvky do stromové struktury, k prvkům stromu přistupuje jednotným způsobem
- motiv:
    - obrazce se společnou abstraktní třídou Obrazec (metoda spoctiObsah()) viz PT
    - složený obrazec začleněn do stromu (člen rodiny obrazců, implementuje rozhraní Obrazec, rekurzivní volání) – prvky strom nejsou unifikované (listy a vnitřní uzly)
    - transparentní řešení – všechny možné operace do abstraktní třídy CObrazec (i operace pro správu dětí – pridej(), odeber(), zjistiDeti()) – unifikace prvků

Struktura vzoru



Použití

- maximalizace rozhraní Component – výhody a nevýhody (není třeba downcasting vs. nutnost ošetřit nesmyslnost operací u Leafu) (celkově výhodnější)
- parent reference

Související vzory
- Iterator pro průchod stromem
- Decorator někdy prvek Kompositu
- Chain of responsibility a odkaz na rodiče
- Visitor na prvku stromu

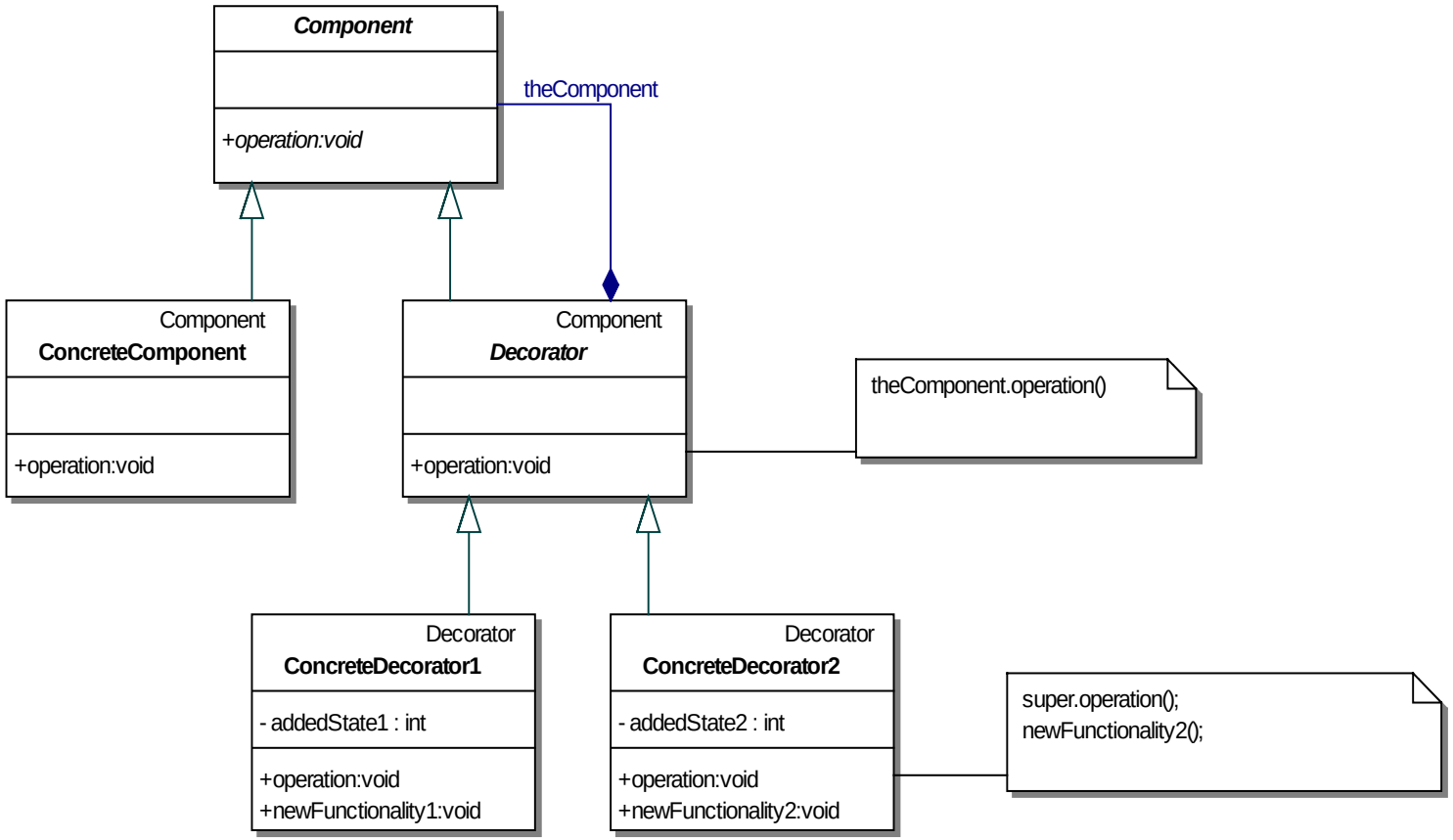
Dekoratér (Decorator)

- klasifikace: Strukturální, Objekt
- smysl: Dynamické přidání další funkcionality k objektu, zavedení flexibilní alternativy k dědění
- alias: Obal (Wrapper)

- motiv:

  - chceme přidat funkcionalitu k nějaké třídě – možné řešení je dědičnost – nevýhodou statická vazba a omezená možnost kombinací
  - jiné řešení – uzavření komponenty do jiného objektu, tzv. dekorátoru, využití skladby, výsledná struktura připomíná zřetězený seznam, nutné stejné rozhraní pro propojení mezi prvky reprezentované jedinou metodou operation()

Struktura vzoru



Created by Borland® Together® Designer Community Edition

- všechny prvky seznamu jsou chápány jako Component
- Component – rozhraní pro objekty, k nimž lze dynamicky přidávat funkcionalitu
- ConcreteComponent – definuje objekt, k němuž lze přidat funkcionalitu, koncový prvek seznamu
- Decorator – odkaz na objekt Component a definice rozhraní vyhovující rozhraní Component, mohou mít za sebou další jeden prvek, Component nebo další Decorator
- ConcreteDecorator – přidává funkcionalitu a stav

Důsledky

- větší tvárnost než statická dědičnost
- postupné přidávání funkcionality tj. není nutné v horní části hierarchie dědičnosti předvídat všechna možná chování
- dekorátor a komponenta nejsou identické – dekorátor se chová jako průhledný obal
- mnoho malých objektů – obtížná laditelnost

Použití

- dynamické přidávání funkcionality jednotlivým objektům
- když je rozšíření pomocí tvorby podtříd nepraktické – masivní nárůst podtříd
- např. streamy v Javě

Související vzory

- podobný Adaptéru a Zástupci, protože za ním také kompatibilní rozhraní
- z matematického hlediska degenerovaný Composite
- podobný vzor Strategie

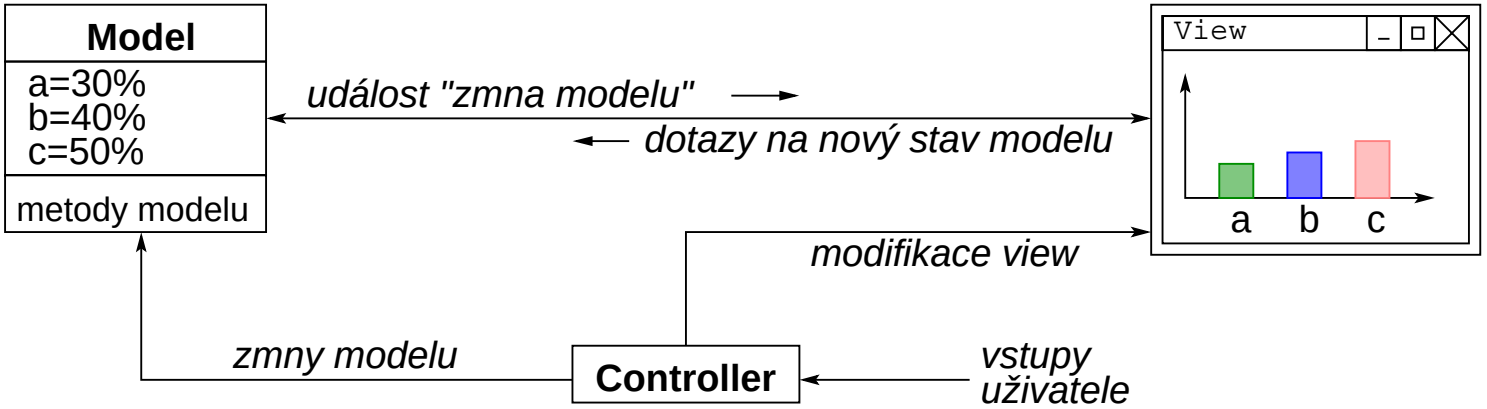
Vztah vydavatel-předplatitel

- angl. publisher-subscriber, subject-observer, broadcasting, model-view
- příklad architektury Model/View/Controller (MVC)

- Model/View/Controller

- funkčnost aplikace rozdělíme na 3 typy objektů, mezi objekty je vztah vydavatel-předplatitel
- "model" je objekt aplikace, zapouzdřuje data, která mají být zobrazena
- "view" (česky "náhled") je prezentace modelu na obrazovce
    - ve chvíli, kdy dojde ke změně stavu modelu, oznámí to model všem "view", které na něm závisejí; view zjistí od modelu nové hodnoty a znázorní je
    - způsobů prezentace může být více, je možné ho změnit např. z tabulky na koláčový graf, aniž by to ovlivnilo model nebo controller

- "controller" definuje způsob, jak uživatelské rozhraní reaguje na vstup
    - způsob reakce na uživatelský vstup je možné změnit, aniž by to ovlivnilo model nebo view; například místo klávesových zkratek použijeme výběr z menu



-

- příklad MVC = použití návrhového vzoru vydavatel-předplatitel
  - pokud je UI dobře navržené, view zná přímo model, ale obráceně to neplatí
  - pokud se model změní, pošle vyvolá metodu "update()" všech view
  - view se pak zeptá modelu na konkrétní věci, které ho zajímají a zpracuje je
- tento způsob komunikace se OO systémech vyskytuje dostatečně často = návrhový vzor - nazveme "vydavatel-předplatitel"
  - vydavatel = instance, v níž může událost vzniknout, tj. v našem případě model
  - předplatitel = instance, která má zájem reagovat na určitou událost
    - předplatitel si u vydavatele zaregistruje metodu, která má být v důsledku události vyvolána, v případě MVC se nazývá "update()"
    - předplatitelů může být i více
- nastane-li událost, vydavatel vyvolá registrované metody předplatitelů

- vztah závislosti je možné implementovat různým způsobem, např. model bude mít v sobě instanční proměnnou ukazující na závislé view, nebo systém může obsahovat sdílený slovník závislostí apod.

- hlavní výhoda návrhového vzoru - model nezávisí na počtu a druhu view - model se může zabývat pouze svými objekty bez vztahu k uživatelskému rozhraní
- view se naopak zabývají znázorněním informací, k tomu musejí znát svůj model a způsob, jak z něj získat informace

Použití

- návrhový vzor vydavatel-předplatitel lze použít, kdykoli potřebujeme oznámit změnu stavu předem neznámému počtu objektů

Právník (lawyer)

- občas dva objekty pracují společně, ale ani jednoho nechceme komplikovat tím, že by měl přímou znalost druhého (podobné Mediatoru)
- např. ikona a model, se kterým ikona souvisí
- chceme oddělit "model" od "view", není dobré dát modelu přímý odkaz na jeho ikonu
- ikona je tak jednoduchá, že od ní neočekáváme, že by znala objekt, který reprezentuje, tj. chceme je od sebe oddělit, ale na druhou stranu chceme model informovat o tom, co se děje s ikonou

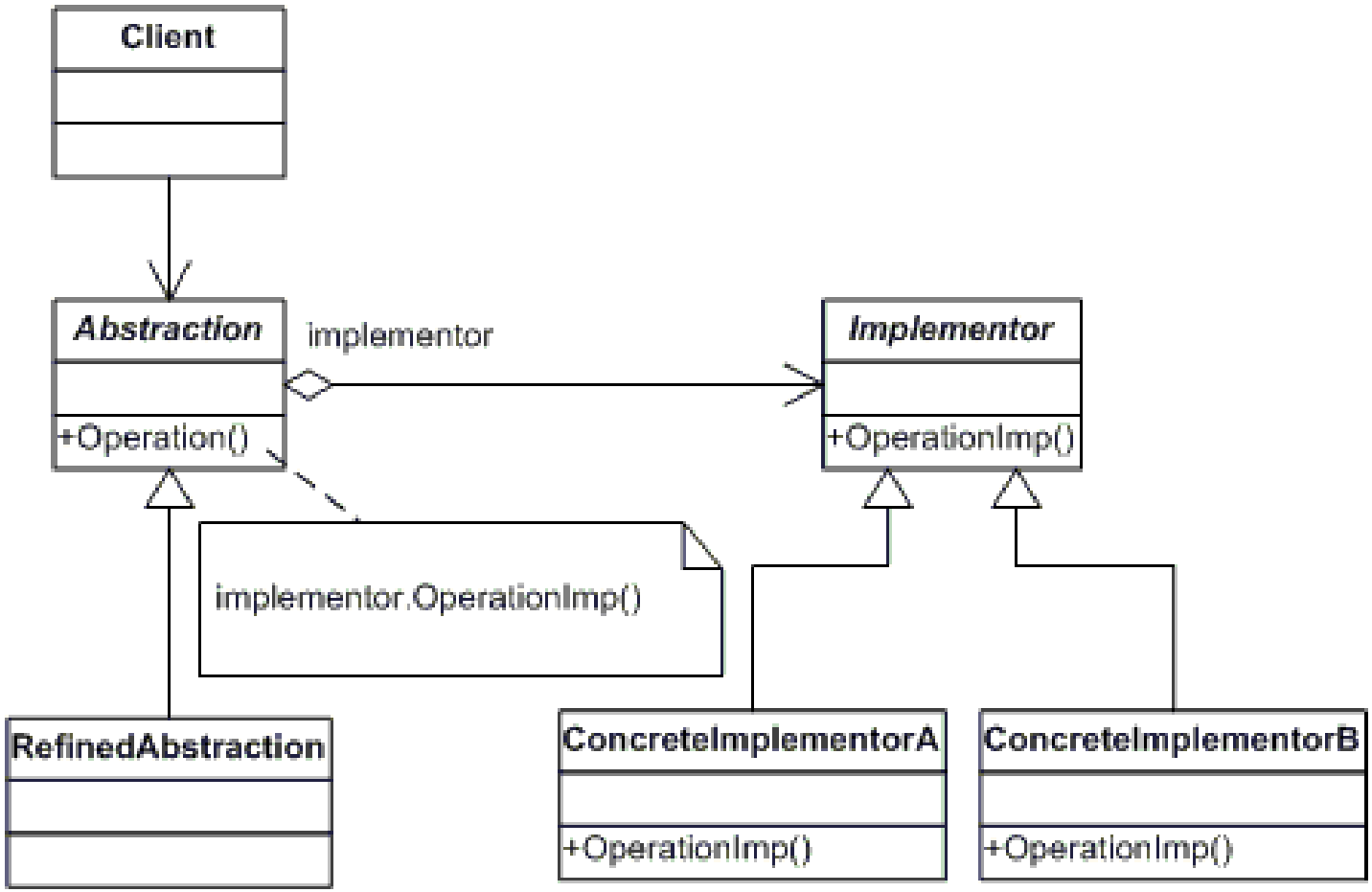
- vytvoříme třetí objekt, nazývaný "právník", který zná ikonu i objekt, který ikona reprezentuje
- uživatelské rozhraní manipuluje s právníky místo s ikonami, resp. může prostřednictvím právníků komunikovat jak s ikonou, tak s modelem

Most (Bridge)

- klasifikace: strukturální, objekt
- smysl: oddělení abstrakce a implementace, lze je nezávisle měnit

- motiv:
    - řešení vztahu mezi abstrakcí a implementací děděním (abstraktní a implementační vrstva) -> mohou vznikat efekt „množení tříd“ (explosion of subclasses)
    - lepší řešení: zavedení abstraktní třídy implementace, ta se spojí (přemostí) s vrcholovou abstraktní třídou – vznikne objektová vazba
    - lze jednoduše přidat novu implementaci stejně jako rozšířit strom abstrakce nezávisle na sobě

Struktura vzoru



- Abstraction - vrchol stromu abstrakce, udržuje referenci na objekt typu Implementor
- RefinedAbstraction – speciální abstrakce
- Implementor – rozhraní pro implementační třídy, může být rozdílné oproti rozhraní Abstraction
- ConcreteImplmentator – konkrétní implementace

Použití:

- fyzické oddělení abstrakce a implementace přes rozhraní (u implementace použití komponentových technologií)
- nezávislost rozšíření stromů abstrakce a implementace
- řešení chyb souvisejících s problematickým použitím dědičnosti (množení tříd)

Související vzory
- volba třídy implementace – Abstraktní továrna, Prototyp

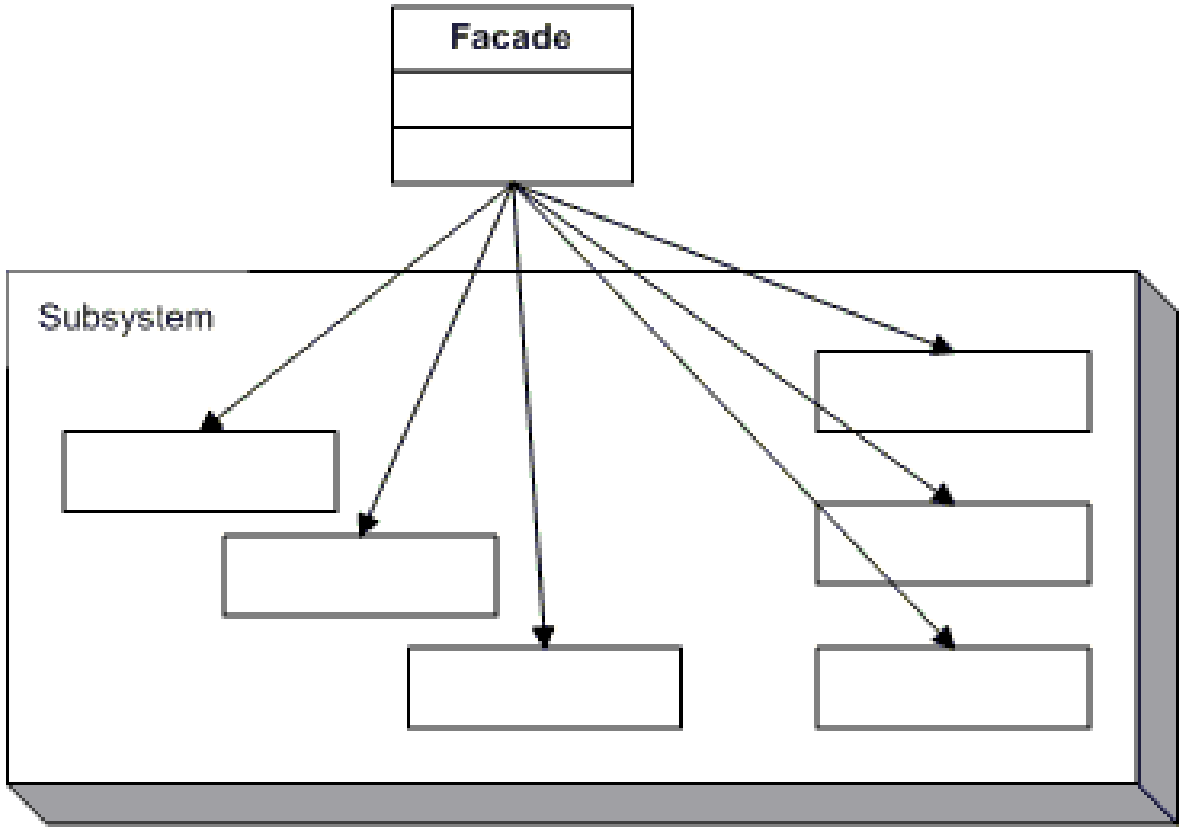
Fasáda (Facade)

- Klasifikace: strukturální, objekt
- Smysl: zjednodušuje přístup klienta k subsystému přes jeden nebo více rozhraní

- Motiv:

    - Užívaný zejména u komponentové technologie
    - Je lepší nabídnout klientovi jedno (popřípadě více) rozhraní místo celou množinu tříd – je to pro něj jednodušší
    - Hlavně pro vnitřně složité subsystémy – rozhraní = pohled na subsystém z pohledu případu užití (ostatní funkcionality se odstíní)

Struktura vzoru



Použití

- odstínění „nepotřebných“ tříd z pohledu případu užití
- sdílení instancí (sdílení objektů v paměti)
- pozor na příliš velké omezení přístupu k podsystému

Související vzory

- vzory, které odstiňují volání tříd v konstrukci objektů -> jednoduchost fasády

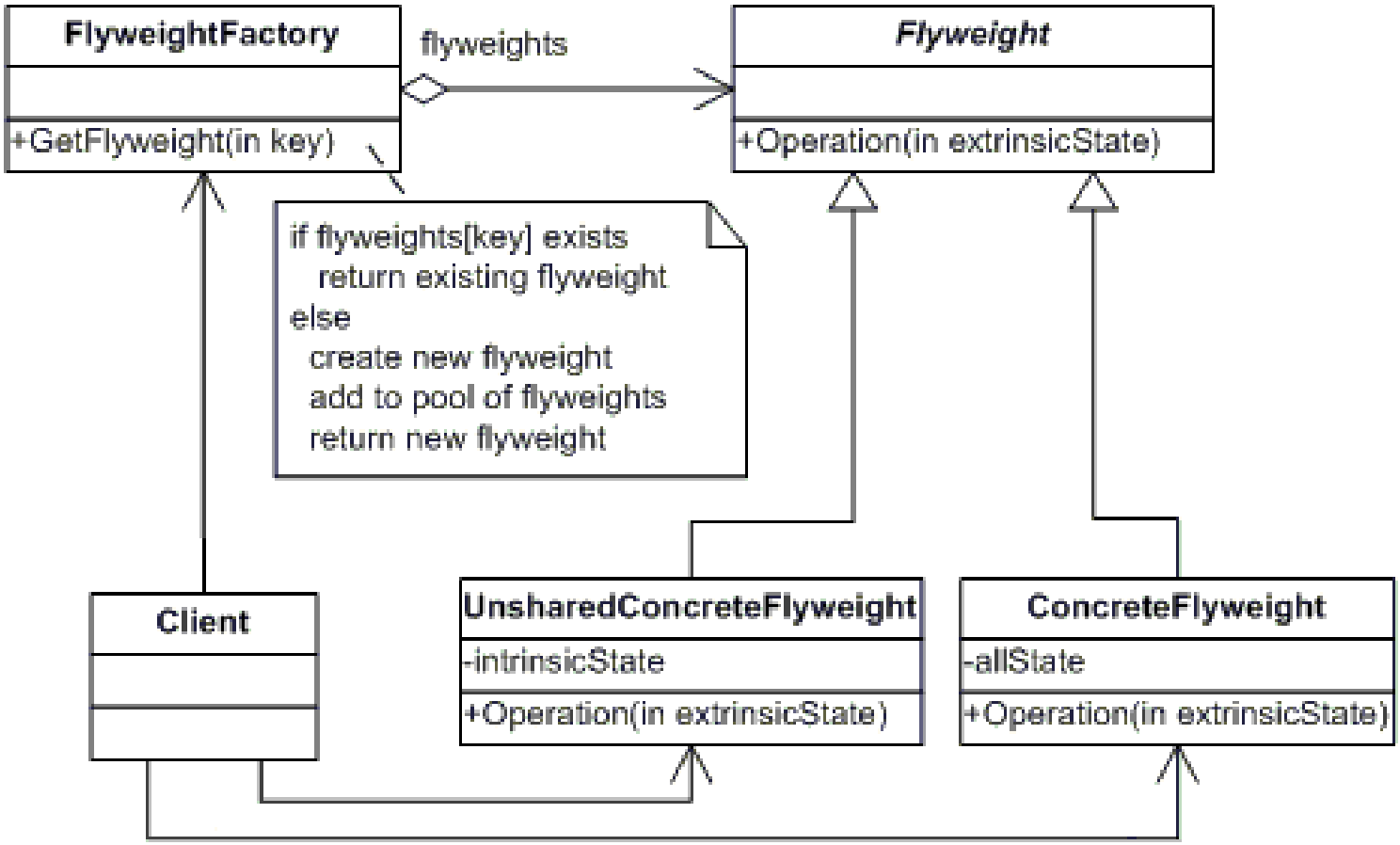
Muší váha (Flyweight)

Klasifikace: Strukturální, objekt
Smysl: systémy s velkým počtem objektů – řešení sdílením

Motiv:
- editor textu (fonty, velikost písma, pozice znaku) – text se skládá ze sloupců, sloupec ze řádek, řádka z písmen obsahujících např. pozici a font -> příliš mnoho potenciálních objektů v paměti
- řešení: každý jeden znak dokumentu nebude objekt v paměti -> problém je nutné „odlehčit“
- u objektů, jejichž počet chceme redukovat, určíme stavy objektů a zavedeme je jako vstupní parametry požadovaných operací (u znaku např. operace Draw())
- různé hodnoty vstupních parametrů -> jiný kontext použití objektu v paměti (sdílený objekt), řídící proces a datový kontext

- u editoru počet objektů = počet možných znaků (číselník znaků), operace Draw() s parametry např. pozice a font (kontext), řídící objekt mapper (na obrázku je to přímo klient) – načte z dat id znaku, pozici a font, požádá číselník o vydání objektu s daným id, zavolá operaci Draw(pozice, font)
- znak je jako objekt použit tolikrát, kolikrát se v dokumentu vyskytuje
- kontext nemusí být předáván přímo z dat do číselníku, mohou existovat další meziúrovně (nesdílené FLYWEIGHT), u editoru např. instance řádku
- nesdílené FLYWEIGHTS – vznikají a zanikají jako objekty, nárůst objektů v aplikaci, ale stále podstatně méně než např. všechny znaky dokumentu

Struktura vzoru



Flyweight – operace pro přijetí a zpracování kontextu
ConcreteFlyweight – sdílené objekty číselníku, vnitřní stavy nezávislé na kontextu
UnsharedConcreteFlyweight – nesdílené objekty číselníku přidávané (a odebírané) za běhu
FlyweightFactory – číselník objektů, zodpovídá za správu objektů
Client (Mapper) – zná referenci do číselníku a kontext

Použití

- není vhodné používat v případě malého počtu objektů, v případě, že není co sdílet (těžko se vyrábí vnější stavy), když se nám nehodí ztráta reference
- pro aplikace s velkým počtem objektů (ale pozor na režii kolem mapování – výpočet stavu, získání sdíleného objektu z číselníku)

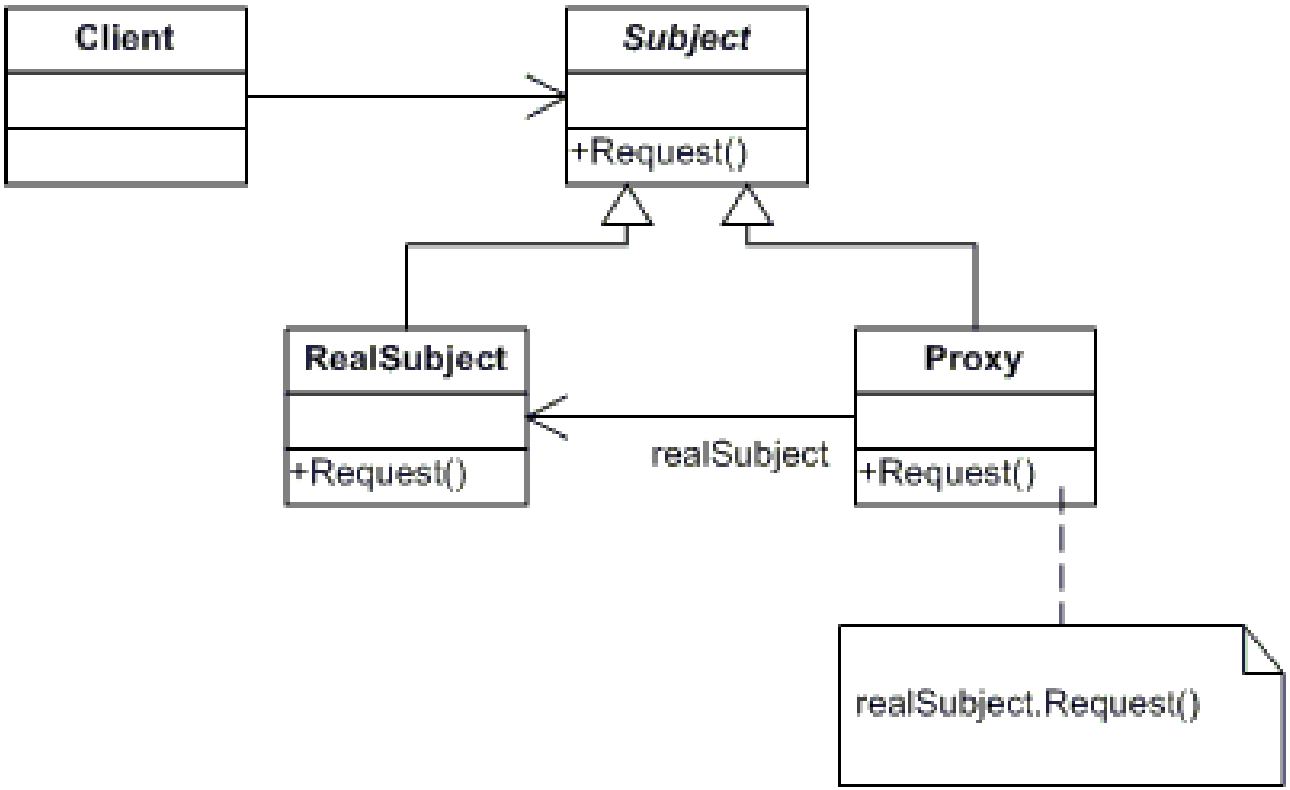
Související vzory
- Composite (listy jsou sdílené Flyweights, vnitřní uzly nesdílené Flyweights)

Zástupce (Proxy)

Klasifikace: Strukturální, objekt
Smysl: objekt zástupce, který kontroluje přístup k objektu

Motiv:
- Přístup ke vzdálenému objektu na jiném stroji, klient chce s objektem pracovat jako by byl lokální
- Řešení: objekt zástupce
    - stejné rozhraní jako vzdálený objekt
    - uchovává technologii propojení mezi stroji

Struktura vzoru



Použití

- remote proxy (součást vývojových prostředků – Java, .NET)
- protection proxy – má zabudováno volání přístupových práv s předmětem práv „povolení operací objektu“, po vyhodnocení práva pokračuje/ nepokračuje operací původního objektu
- virtual proxy – nahradí původní objekt a provádí operace až ve chvíli, kdy jsou skutečně třeba (např. upload dat), když operace není třeba, je za ním prázdný objekt
- smart reference – řízený přístup k objektům (např. zamykání objektů)

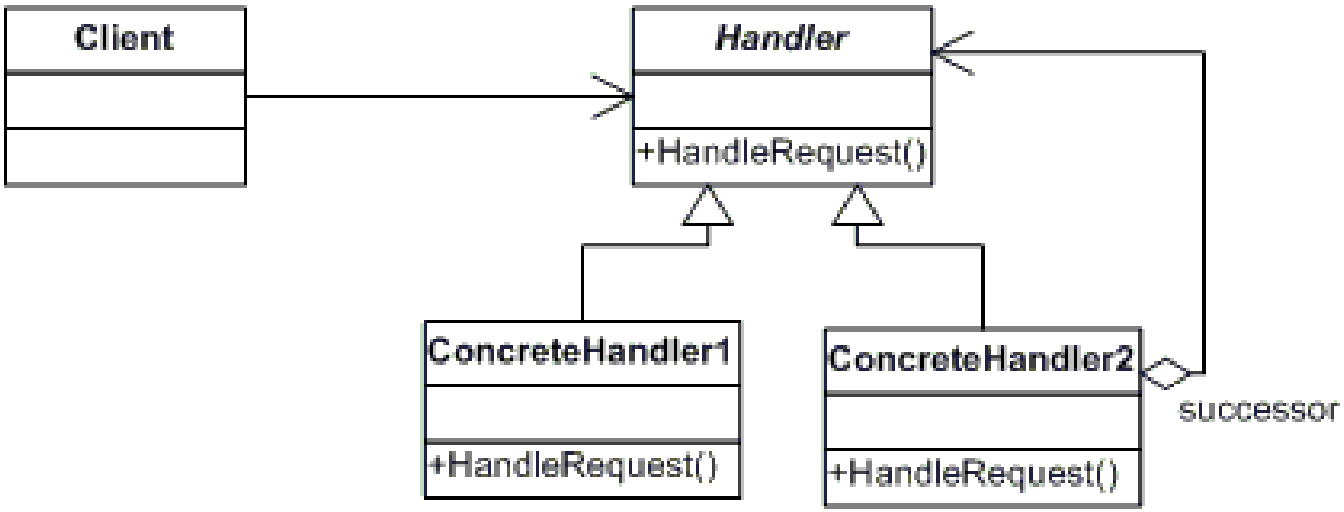
Související vzory
- Adapter – dvě různá rozhraní
- Decorator – stejné rozhraní v řetězu za sebou, přidává funkcionalitu

Řetěz odpovědnosti (Chain of responsibility)

- klasifikace: chování, objekt
- smysl: měnitelný řetěz objektů propojených stejným rozhraním k předání a zpracování požadavku

Motiv:
- Zpracování Helpu, každý prvek GUI – požadavek na Help, pokud prvek help má, provede se, pokud nemá, pošle tento požadavek jinému objektu (rodič v GUI), může doputovat až na vrchol hierarchie

Struktura vzoru



Handler – společné rozhraní pro prvky stromu
ConcreteHandler – objekty z řetězu , některé prvky ukončují řetěz vlastní funkcionalitou, některé posílají požadavek dál.

Použití

- dynamický řetěz – konfigurace v runtime (pozor na zpracování na konci řetězu, zacyklení, vkládání zbytečných prvků na konec)

Související vzory

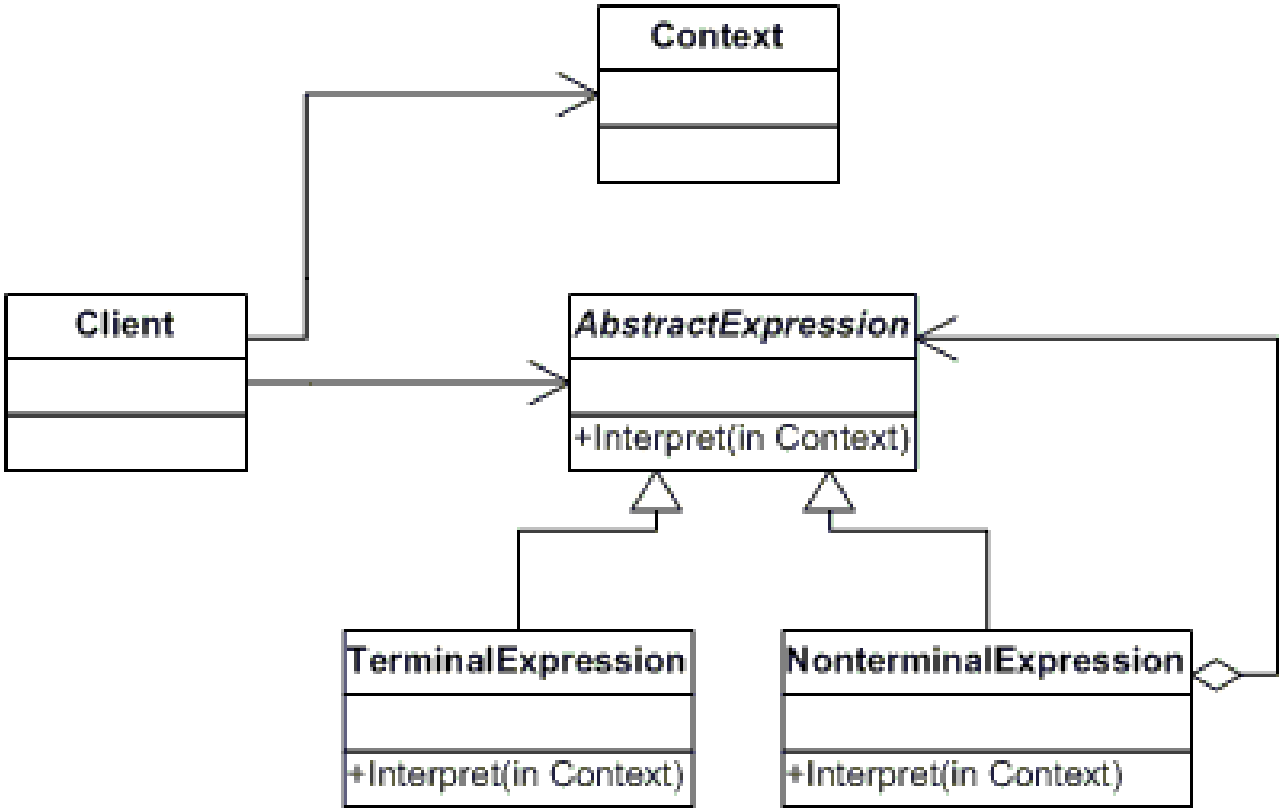
- vazba pro následovníka pro Řetěz odpovědnosti už je např. přirozeně u Compositu (reference na rodiče), viz GUI a Help

Interpreter

- Klasifikace: chování, třída
- Smysl: reprezentace pravidel gramatiky modelem tříd, zavedení odpovídajícího interpretu pravidel operacemi objektů těchto tříd

Motiv
- Definice jednoduché gramatiky na metaúrovni jako vztahy mezi třídami, její interpretace na konkrétních instancích (vztahy mezi instancemi)
- Všechny třídy patří do téhož stromu dědičnosti

Struktura vzoru



Použití

- Není vhodný pro složité gramatiky, každé pravidlo zavádí třídu
- Vzor neukazuje, jak vybudovat strom objektů daného výrazu, ale jak jej „využít“

Související vzory

- Strom reg. výrazu jako Composite (průchod Iterátor)
- Non-terminální symboly – možnost optimalizace přes Flyweight

Mediator

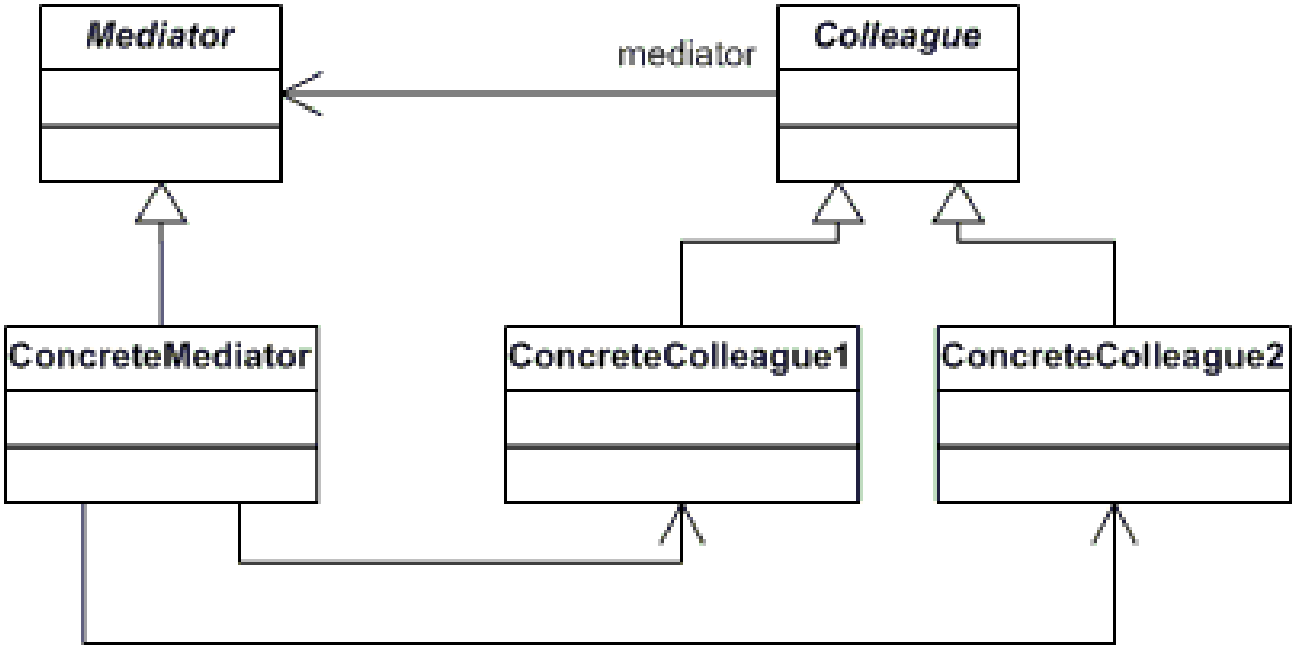
- Klasifikace: chování, objekt
- Smysl: zavedený objekt je prostředníkem mezi jinými objekty (tyto pak nemusí mít mezi sebou přímé vazby), odděluje je a zprostředkovává mezi nimi komunikaci

- Alias: Controller

- Motiv:
  - spolupráce objektů v GUI, změna jednoho prvku -> nutnost změny dalších prvků, když je prvků mnoho -> nepřehlednost
  - řešení – objekt „vidící“ všechny komunikující objekty (přes objektové reference) - mediátor, zároveň všechny objekty „vidí“ mediátor, při změně prvku v GUI volá prvek specifickou operaci v Mediátoru, který pak v rámci operace volá ostatní prvky

- často skryto v GUI - automatické zavedení vzoru, formulář zároveň mediátor – např Delphi , jiné řešení - listener s registrací – Java
- složitější konstrukce – přímé volání operace mediátoru (ne přes vyvolání události) – každý prvek musí obsahovat instanci mediátoru a vidí jeho rozhraní – polymorfní operace ColleagueChanged(Colleague colleague), colleague jako vstupní parametr (musíme vědět, co a jak se změnilo) je obecný prvek

Struktura vzoru



Použití

- systém událostí přehlednější než přímé volání rozhraní mediátoru
- formuláře jako mediátory (možnost přesunutí obslužného kódu do jiných formulářů (tzv. Controllerů – je to mediátor)

Související vzory

- Command – pvky při požadavku na spolupráci nevolají rozhraní mediátoru, ale MyCommand. execute(), každá třída Commandu obaluje jednu metodu Mediátoru (tříd Commandů = počet metod Mediatoru)
- Observer – událostní systém

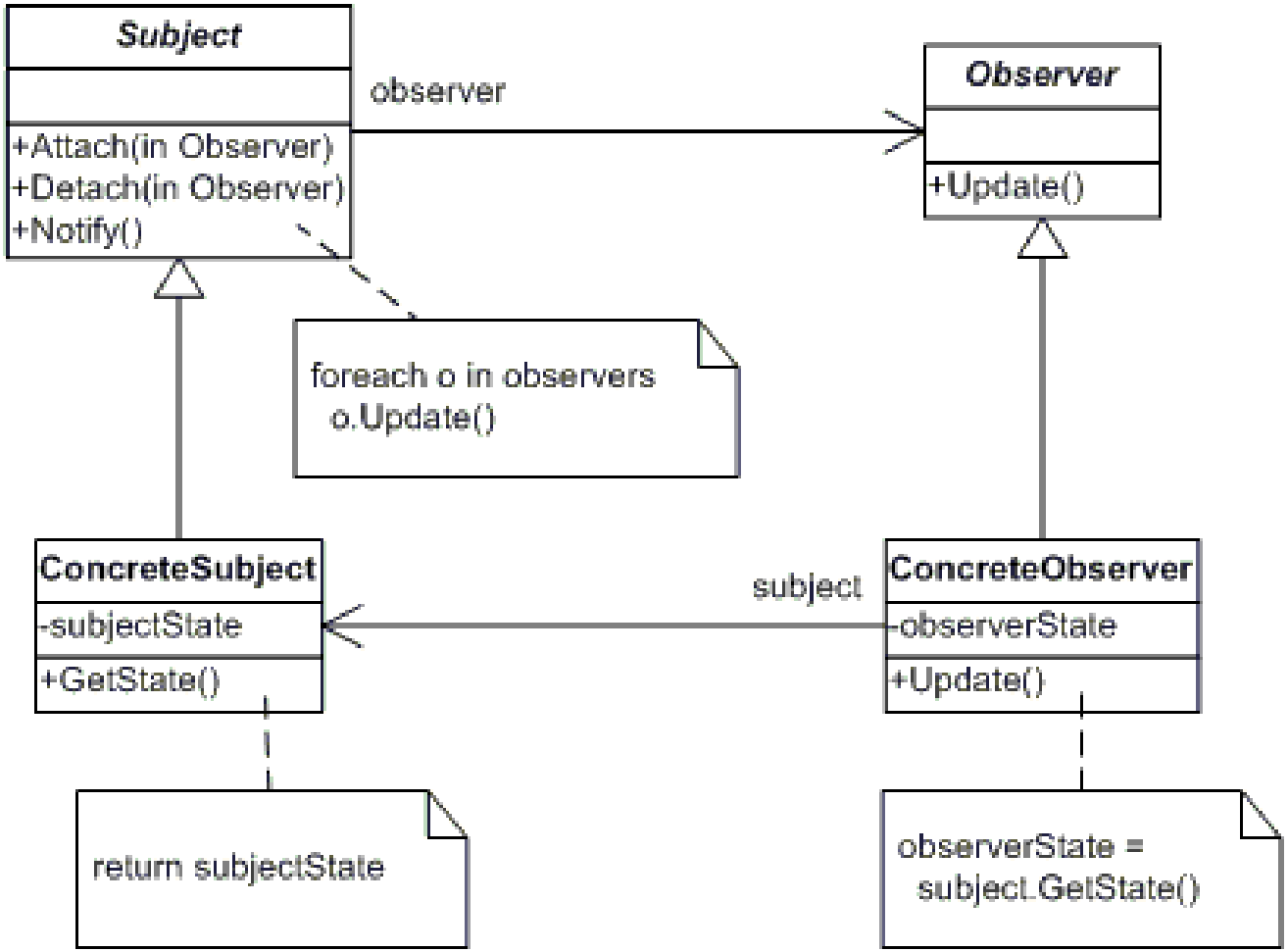
Observer

- klasifikace: chování, objekt
- smysl: možnost sledování změn u objektu, když objekt změní stav, ostatní objekty na to zareagují, avšak není přímá vazba od sledovaného objektu k těmto objektům

- Motiv

  - obdoba událostního systému (Java - listener)
  - první krok: objekt se spolu se svojí operací zaregistruje ke službě události jiného objektu, druhý krok: dojde k události, všechny registrované operace se zavolají
  - nutnost kompatibility volání zaregistrovaných operací -> vzor Observer

Struktura vzoru



- Abstraktní třída Subjekt – implementované operace pro přidání/odebrání objektu Observer a volání operace Update() všech zaregistrovaných Observerů = operace Notify()
- Observer – polymorfní operace Update()
- ConceretObserver – implementuje operaci Update(), udržuje referenci na ConcereSubject, může si vyžádat stav ConcreteSubjectu
- ConcreteSubject - v rámci operace SetState() dojde zavolání operace Notify(), tj. k události a následovnému zavolání Update()

Použití

- dopředu není jasné, kdo má sledovat změnu sledovaného objektu
- konkrétní Subject neví, koho obsluhuje, konkrétní Observer je odstíněn od Subjectu
- pokud je jasné, kdo na změnu bude reagovat, a nebude se to měnit, vzor se nepoužívá, raději přímé volání daných objektů
- pozor na zacyklení událostí (více vrstev Observerů)

- při volání operace Notify() je třeba, aby konkrétní Subject byl v konzistentním stavu (Observer může načíst nekonzistentní stavy -> průšvih)
- vzor neřeší multithreading
- registrace observera a asociativní třída – dvojice Subject + Observer

Související vzory

- Command – podobná funkcionalita a její obecné spouštění (nikoli sledování změny)

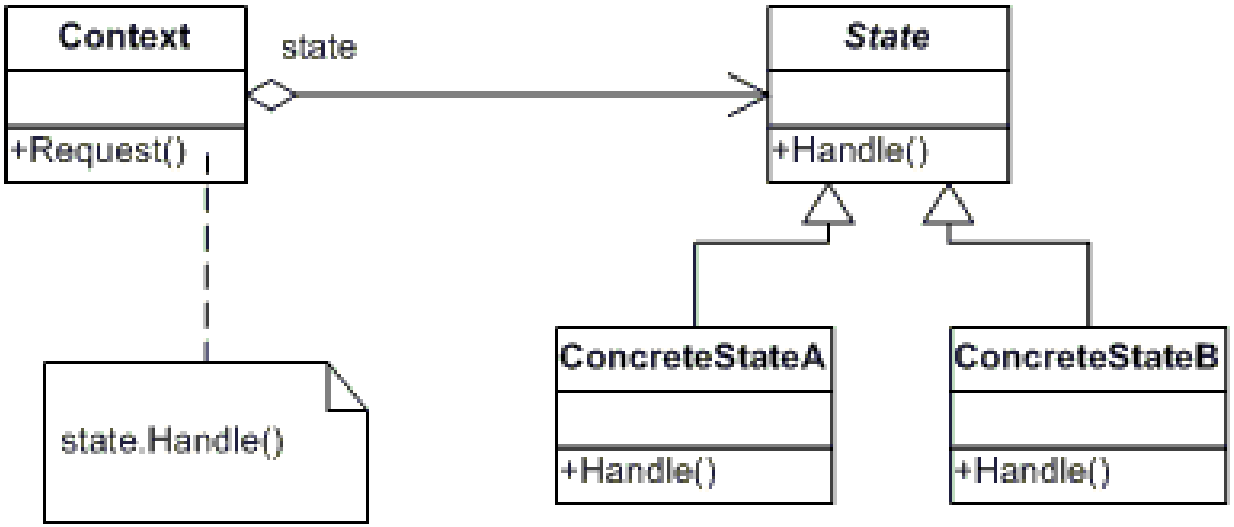
Stav (State)

- klasifikace: Chování, objekt
- smysl: změna stavu objektu prováděna výměnou objektu reprezentujícího stav

- motiv

    - vnitřní stav objektu
    - volání operací na základě vnitřního stavu objektu – implementace s if nebo switch může být nepřehledná
    - Řešení: abstraktní třída State, polymorfní operace zpracování stavu, dědicovové třídy State() (počet stavů = počet dědiců), přepnutí stavu = výměna objektu typu State()

Struktura vzoru



Context– rozhraní pro klienta, zná stavy, přesměruje požadavek na chování v daném stavu na objekt state
State – abstraktní třída pro chování ve stavu
ConcreteState – implementace chování v konkrétním stavu

Použití

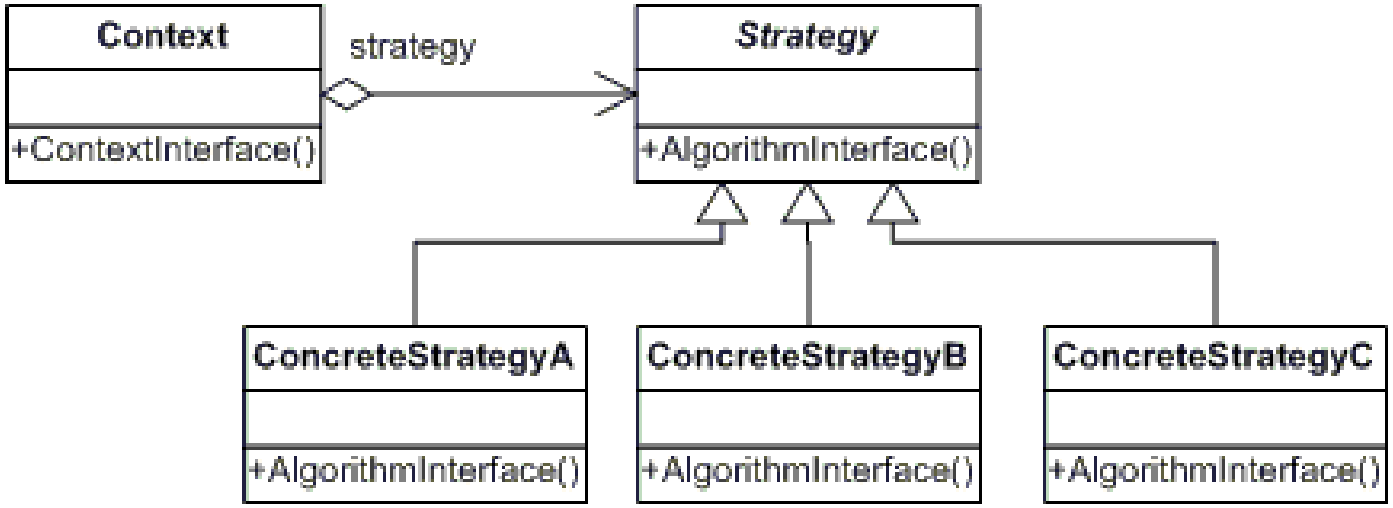
- přehlednost a flexibilita – nový stav= nový potomek třídy State
- menší riziko chyb a nesmyslných mezistavů
- přechody mezi stavy, nejlépe umí dědicové přímo v metodách Handle(), nutno mít referenci (zpětnou) na rozhraní objektu typu Context
- stavové atributy společné pro všechny stavy je lépe umístit do objektu Contextu (State má na Context referenci) nebo úplně mimo + svázat přes asociaci, nikoli do abstraktní třídy State

Strategie (Strategy)

Klasifikace: Chování, objekt
Smysl: Definuje množinu algoritmů, které lze za běhu vyměnit

Motiv:
- máme k dispozici více algoritmů vyhledávání dat, požadavek na použitý algoritmus dán obsluhou za běhu programu, neflexibilní řešení: přepínač
- jiné řešení: rozhraní Strategy s operací např. SearchAlgorithm(), dědicové ji přepisují, výběr algoritmu na základě výběru instance třídy Strategy

Struktura vzoru



- Strategy – rozhraní pro všechny operace (podporované algoritmy)
- ConcreteStrategy – implementace konkrétních operací
- Context – udržuje referenci na objekt Strategy, konfigurace objektem ConcreteStrategy, může definovat rozhraní, aby mohl objekt Strategy přistupovat k datům Contextu

Související vzory

- Podobný vzoru State
- Strategy – sdílený algoritmus -> Singleton nebo Flyweights

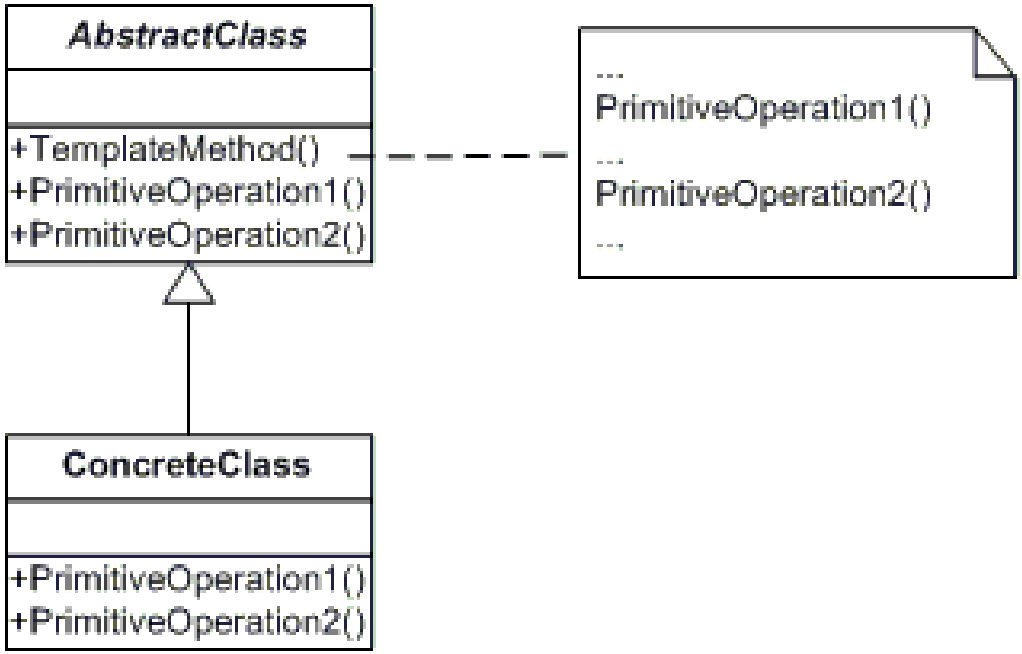
Šablona (Template Method)

- klasifikace: chování, třída
- smysl: existence scénáře na abstraktní úrovni, obsahuje polymorfní metody, potomci je přepisují a volají scénář

- motiv
  - dokument u factory method – na úrovni předka operace (scénář) openDocument(), obsahuje polymorfní operaci createDoc(), potomci scénář využijí, operace přepíší

```
public void openDocument() {
    myDoc = createDoc();
    docs.add(mydoc);
    myDoc.open();
}
```

Struktura vzoru



- AbstractClass
  - obsahuje TemplateMethod() – scénář
  - obsahuje primitivní operace (nemusí být nutně abstraktní – implicitní chování)

    - samotná třída nemusí tedy být abstraktní
- ConcreteClass – přepisuje polymorfní operace

Použití

- přesně definované postupy – předefinování operací a zavolání TemplateMethod()

Související vzor

- Factory method

Návštevník (Visitor)

- klasifikace: chování, objekt

- smysl: zavádí operaci, která účinkuje na prvky objektové struktury, přitom se nemění kód ve třídách těchto prvků

- motiv
  - strom obrazců, obrazců je hodně, potřebujeme přidat operaci spoctiObvod(), nechceme se nám ale zasahovat do kódu třídy každého obrazce
  - řešení: nová operace (pseudooperace) bude využívat již existující operace obrazce (tzn. že obrazce musí umět operace jako getPolomer(), getDelkaStranyA() apod.), aneb z návratových hodnot zavedených operací musíme umět získat vše potřebné pro nově zavedenou operaci (pseudooperaci)

- celkový postup:
    - každý prvek stromu obrazců bude mít polymorfní operaci pro přijetí Visitora např. accept (CVisitor visitor), tj. vlastní implementaci této operace

    - *každá pseudooperace bude reprezentována jednou třídou Visitora* (Visitoři tvoří strom dědičnosti na vrcholu s abstraktní třídou Visitor), tj. pro obrazce jedna třída Visitora např. VisitorObvod a abstraktní třída Visitor (typ vstupního parametru operace accept)
    - -> *tj. obrazec umí přijmout libovolného Visitora*

    - existence dvojitého polymorfismu: každý Visitor má přesně tolik operací, kolik jednotlivých obrazců ve stromu (operace jsou polymorfní, každý Visitor si je implementuje) může navštívit

- šablona Visit<Konkrétní obrazec> se vstupním parametrem = členem, kterého hodlá Visitor navštívit (Visitor si tak může vyžádat vše potřebné), tj. např. operace

- *visitKruh (Ckruh kruh)*
- *vistObdelnik (CObdelnik obdelnik)*
- *visitTrojuhelnik (CTrojuhelnik trojuhelnik)*

- ve výsledku: *přijetí Visitora daným konkrétním obrazcem se převede na zavolání visitorské návštěvy tohoto konkrétního obrazce*

Např. kruh implementuje

```
public void accept (Visitor visitor) {
    visitor.visitKruh(this);
}
```

….obdélník implementuje

```
public void accept (Visitor visitor) {
    visitor.visitObdelnik(this);
}
```

…složený obrazec implementuje

```
public void accept (Visitor visitor) {
    for each item in obrazce item.accept(visitor);
}
```

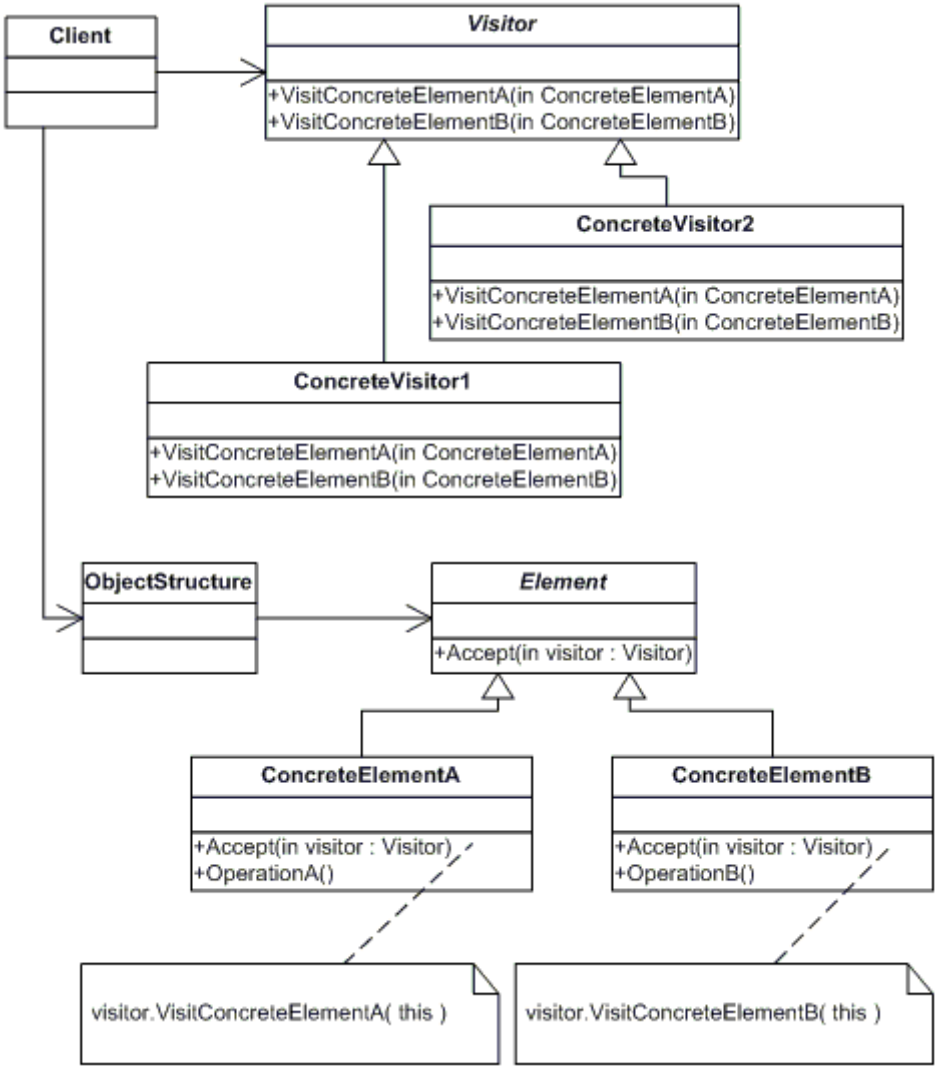
Scénář práce s Visitorem

- vybere se pseudooperace (tj konkrétní třída Visitora)
- zavolá se operace obraze accept(visitor), vstupním parametrem objekt vybrané třídy Visitora
- vybraný obrazec vrací řízení visitorovi např. visitor.visitKruh(this)
- uvnitř visitora máme k dispozici obrazec, provedeme nad ním pseudooperaci, kterou daný visitor umí (tj. např. spočtení obvodu kruhu)
- výsledek převezme visitor do sebe a může jej na požádání vydat klientovi, který spouští accept()
- Visitor může postupně projít celou strukturou a posbírat všechny údaje
- Je možné flexibilně vyrobit dalšího visitora

Použití visitora např.

VisitorObvod myVisitor = new VisitorObvod(); /v_obvod = 0;
myObrazec.accept(myVisitor);
obvod = myVisitor.getObvod();

Struktura vzoru



Použití

- flexibilita operací
- sběr údajů

Související vzory

- zpracování Compositu
- projití struktury s využitím Iterátoru

Transfer Object (J2EE pattern)

- kontext: klient si potřebuje vyměňovat data s komponentou

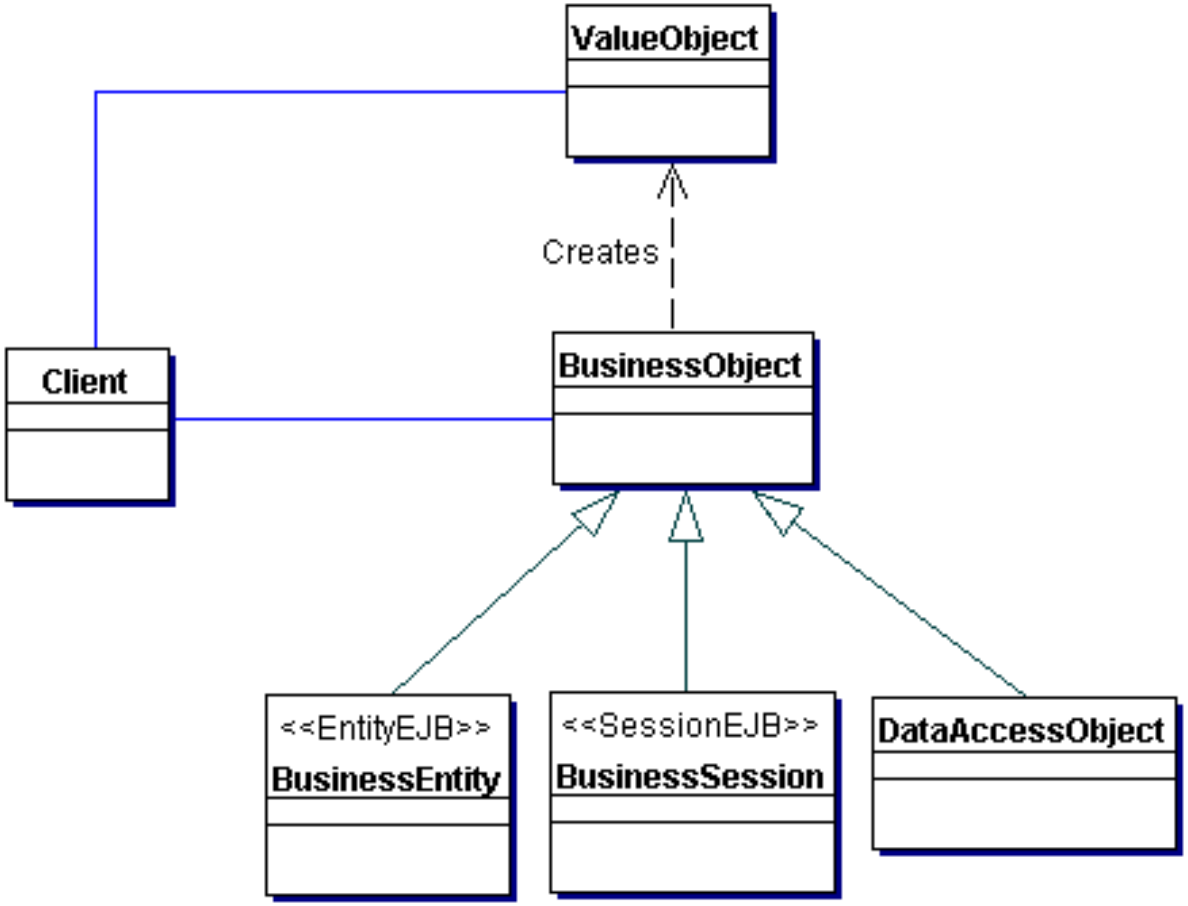
- motiv:

  - komponenty na straně serveru poskytují data, tj. dávají k dispozici přístupové metody, každé volání metody klientem je potenciálně vzdálené, pokud je volání mnoho a klienti žádají jednoduché atributy, klesá výkonnost celé aplikace

  - řešení: aplikační data jsou zapouzdřena do Transfer objectu, manipulace s tímto objektem pak probíhá přes jedinou metodu, klient žádající o data obdrží celý Transfer object poté, co je tento objekt zkonstruován

  - využití Transfer objectu je spojeno s mnoha strategiemi

Struktura vzoru

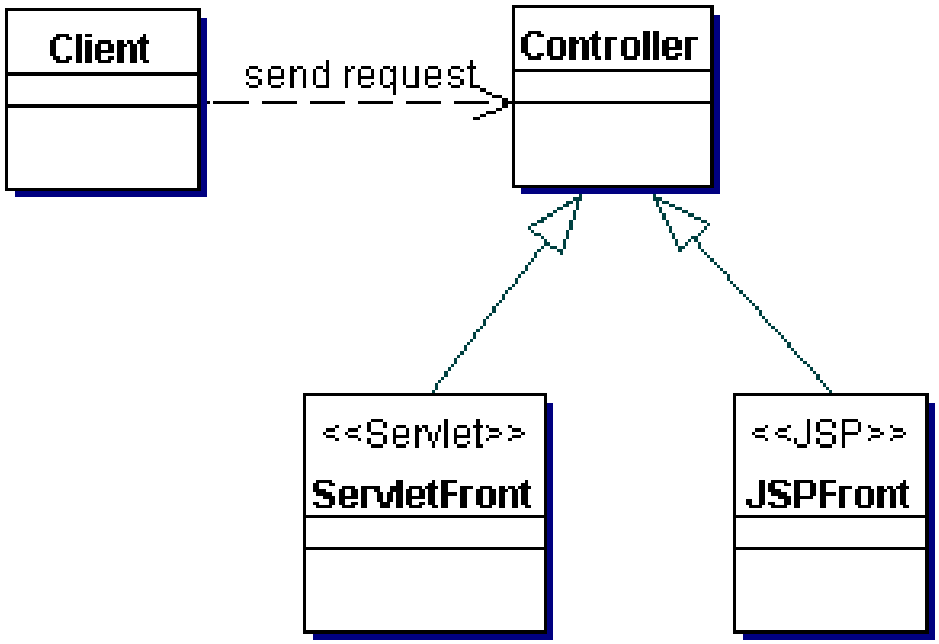


Front Controller (J2EE pattern)

- kontext – mechanismus řízení požadavků na úrovni prezentační vrstvy řídí a koordinuje zpracování uživatelských požadavků, tento mechanismus může být centralizovaný/decentralizovaný

- motiv
  - pokud uživatel může přistupovat k prezentační vrstvě bez „průchodu“ centralizovaným mechanismem řízení požadavků, je často nutné duplikovat kód jednotlivých view, navíc dochází ke smísení obsahu a řízení daného view
  - řešení: použití Controlleru jako prvotního bodu kontaktu s požadavkem, controller spravuje další zpracování požadavku (např. přihlášení, výběr korektního view apod.)
  - pro implementaci Front Controlleru existuje několik strategií

Struktura vzoru

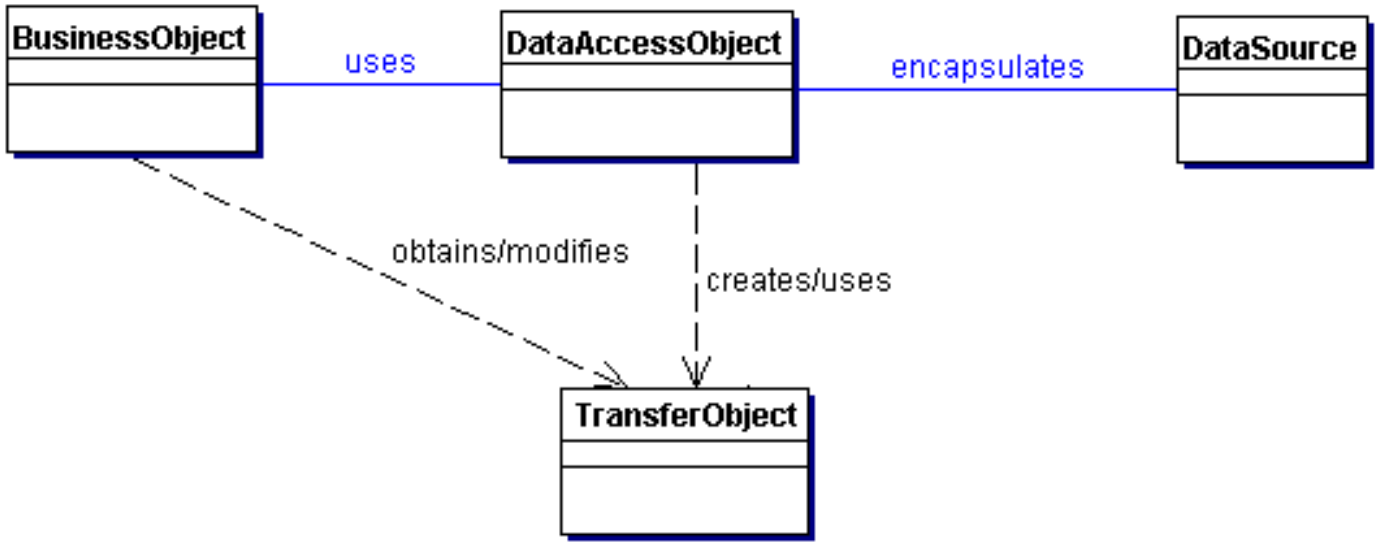


Data Access Object (DAO) (J2EE pattern)

- kontext – přístup k úložišti dat závisí na typu daného úložiště (relační databáze, soubor,…) a jeho implementaci

- motiv

  - potřebujeme získávat/ukládat data z/do persistentních datových úložišť, tyto se mohou měnit, mají různá API, klient chce však používat jednotné rozhraní

  - řešení: Data Access object zapouzdřující přístup k datovému zdroji, DAO zařizuje veškerou komunikaci s datovým zdrojem ohledně ukládání/získávání dat (je to takový Adaptér)

Struktura vzoru