Návrhové vzory
SI3

Ondřej Macek
Tomáš Černý

Návrhové vzory (Design Patterns)

Naše poznatky při návrhu software jsou následující:

- Problémy při návrhu se opakují
- Mnoho problémů je stejnou strukturu řešení
- Přesné řešení záleží na kontextu
- Vývojář znalý návrhových vzorů může řešit nové problémy rychleji a lépe
- „Best practices“

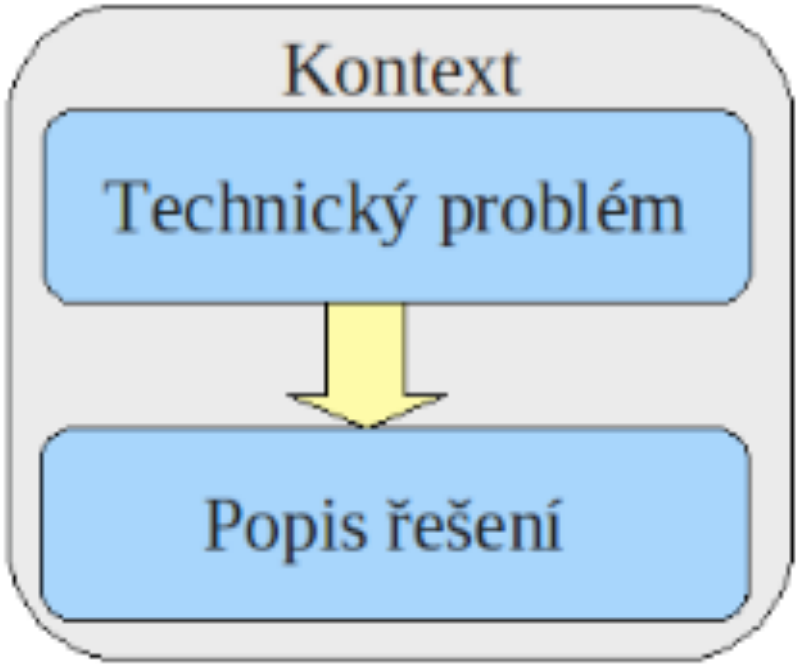
Návrhové vzory (Design Patterns)

Na návrhové vzory lze též nahlížet jako na **trojici**, která vyjadřuje vztah mezi určitým

1.kontextem

2.problémem

3.řešením



Návrhové vzory

Jaké vzory znáte?

Návrhové vzory

Jaké vzory znáte?

```
List<String> aList = new ArrayList<String>();
// Add Strings to alist
for (Iterator<String> iter = aList.iterator();
                          iter.hasNext(); ) {
    String s = iter.next();
    // No downcasting required.
    System.out.println(s);
}
```

Návrhové vzory

Jaké vzory znáte?

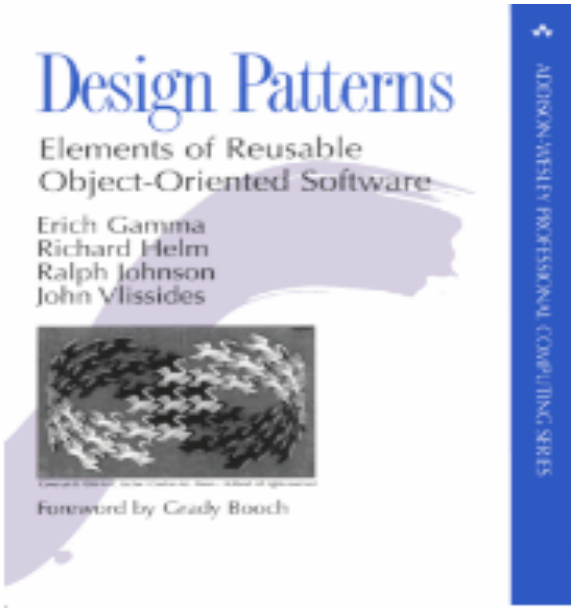
```
List<String> aList = new ArrayList<String>();
// Add Strings to alist
for (Iterator<String> iter = aList.iterator();
                              iter.hasNext(); ) {
    String s = iter.next();
    // No downcasting required.
    System.out.println(s);
}
```

Nebo skrytě

```
for (String s : aList) {
    System.out.println(s);
}
```

Návrhové vzory (Design Patterns)

- Mnoho knih o návrhových vzorech
- Gang of 4 neboli Gamma book
  - katalog vzorů pro návrh řešení studovaného problému



Návrhové vzory (Design Patterns)

Návrhový vzor zachycuje

- nejlepší dostupnou zkušenost z vývoje software
  - ve formě problém-řešení

Návrhový vzor se skládá z:

- charakteristiky problému
- charakteristiky řešení
- množiny transformačních doporučení

Návrhové vzory (Design Patterns)

Řeší strukturní problémy jako

- Abstrakci
- Zapouzdření
- Schování informací
- Rozdělení zájmů / účasti
- Coupling & cohesion – párování a provázanost
    - menší párování je lepší
    - větší provázanost (jasnost, soudržnost) je lepší
- Rozdělení rozhraní od implementace
- Jeden odkazový bod
- Rozděl a panuj

Návrhové vzory (Design Patterns)

Řeší nefunkční problémy jako:

- Zaměnitelnost
- Spolupráci
- Efektivnost
- Spolehlivost
- Testovatelnost
- Znovu použití

Typy vzorů

Vzory požadavků

- Popisuje množinu požadavků pro množinu aplikací
  - check-in check-out pattern
    - systém knihovny
    - půjčovna automatická
    - video systémy

Typy vzorů

Vzory požadavků

- Popisuje množinu požadavků pro množinu aplikací
  - check-in check-out pattern
    - systém knihovny
    - půjčovna automatická
    - video systémy

Programovací styly

- Popisuje specifické řešení pro programovací jazyk
  - Reflexe pro Javu
    - Co jsou to Reflexe?

Typy vzorů

Architektonické vzory

- Popis architektur
  - Broker pattern
    - distribuované systémy, kde pozice souboru a služby je transparentní.
    - COR**B**A
  - **Model View Controller (GUI)**
  - Pipe-and-Filter (Staré překladače)
  - Vícevrstvé architektury (TCP/IP, J2EE)

Typy vzorů

Architektonické vzory

- Popis architektur
  - Broker pattern
    - distribuované systémy, kde pozice souboru a služby je transparentní.
  - Model View Controller
  - Pipe-and-Filter
  - Vícevrstvé architektury

Návrhové vzory

- Popisuje návrhová řešení v menším měřitku
  - Gang of Four

Poznatek

Vzory adresují opakující se vývojové **problémy**, vyskytující se určitých **kontextech** a podávají **řešení** problému.

Vzor zahrnuje nejlepší praktiky „best practices“ v softwarovém vývoji.

Tyto vzory též slouží ke **komunikaci** mezi vývojáři.

Jaký je rozdíl mezi architekturou, vzorem a frameworkem?

1. **Architektura** ???

2. **Vzory** ???

3. **Frameworky** ???

Jaký je rozdíl mezi architekturou, vzorem a frameworkem?

1.**Architektura** modeluje softwarovou strukturu na nejvyšší úrovni a podává pohled na systém jako celek. Architektura používá až několik různých vzorů v různých komponentách.

2.**Vzory** jsou menší či místní architektury či architektonické komponenty.

3.**Frameworky** jsou částečně kompletní softwarové systémy, které se mohou zaměřovat na konkrétní typ aplikace. Tyto systémy jsou pak upraveny na míru kompletováním nedokončených komponent.

Rozdělení GoF vzorů

- Tvorba nových instancí a tříd
- Strukturní vzory
- Vzory chování.

Rozdělení GoF vzorů

- Jaké že jsou? Ty tři? :D

Rozdělení GoF vzorů

Vzory pro tvorbu instancí a tříd jsou:

- Factory method
- Abstract Factory
- Prototype
- Builder
- Singleton
- ..

Rozdělení GoF vzorů

Strukturní vzory jsou:

- **Adapter**
- **Bridge**
- Composite
- Decorator
- **Facade**
- Flyweight
- Proxy
- ..

Rozdělení GoF vzorů

Vzory chování:

- Interpreter
- Template Method
- Chain of responsibility
- Command
- **Iterator**
- Mediator
- Observer
- State
- Strategy
- Visitor
- ...

Architektonický vzor Model-View-Controller (MVC)

Pokud píšeme aplikaci a mixujeme dohromady kód pro přístup dat, business logiku a prezentaci, tak jsme na nejlepší cestě do pekel.

- Úprava malé části kódu má nedozírné následky
- Velké spárování kódu
- Nemožnost znovu-použití kódu (re-use)
- Oprava chyby pak vyžaduje přístup na mnoha místech

Model View Controller řešící tyto problémy

MVC

Vzor odděluje blok pro přístup k datům, business logiku a datovou prezentaci v UI.

- Kontext: Vývoj uživatelského rozhraní
- Problém: Jak vytvořit UI pružné ku změnám v layoutu a funkčnosti
- Faktory: změny v UI by měli být snadné a možné za chodu, adaptování UI by nemělo zasáhnout implementaci funkčnosti.

MVC

Komponenty

- Model
- Pohled (view)
- Kontrolér

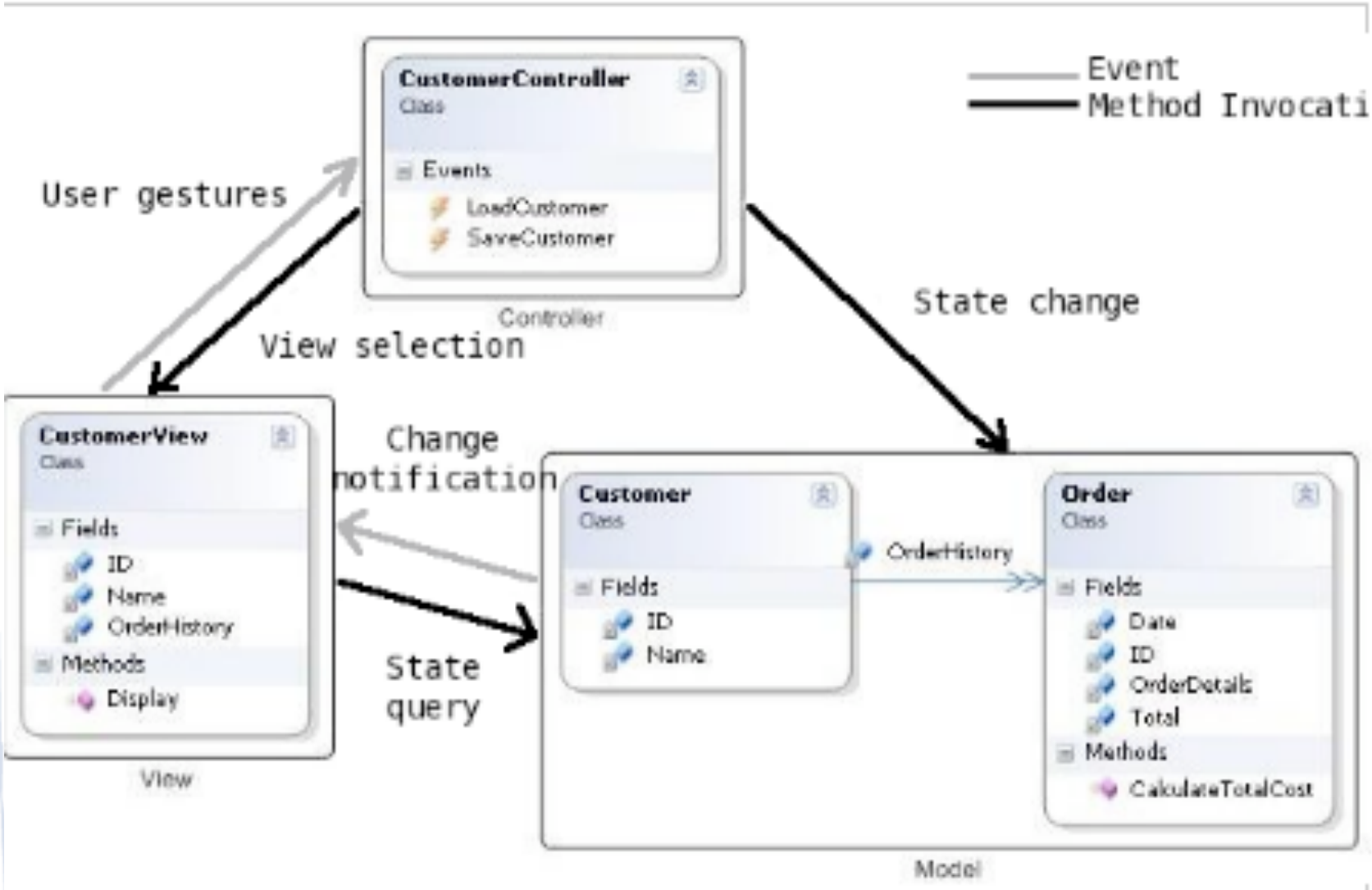
MVC

Model zapouzdřuje hlavní funkčnost, je nezávislý na vstupně-výstupní reprezentaci a chování.

Pohled zobrazuje data z modelu, pro jeden model může existovat mnoho pohledů.

Kontrolér řídí vstupy z pohledu. Uživatel pracuje se systémem přes komponenty kontroléru.

MVC


CustomerController
Class
Events
LoadCustomer
SaveCustomer
Controller
Event
Method Invocati
User gestures
View selection
State change
CustomerView
Class
Fields
ID
Name
OrderHistory
Methods
Display
View
Change
notification
State
query
Customer
Class
Fields
ID
Name
OrderHistory
Order
Class
Fields
Date
ID
OrderDetails
Total
Methods
CalculateTotalCost
Model

MVC

MVC odděluje pohledy a modely zavedením subscribe/notify protokolu.

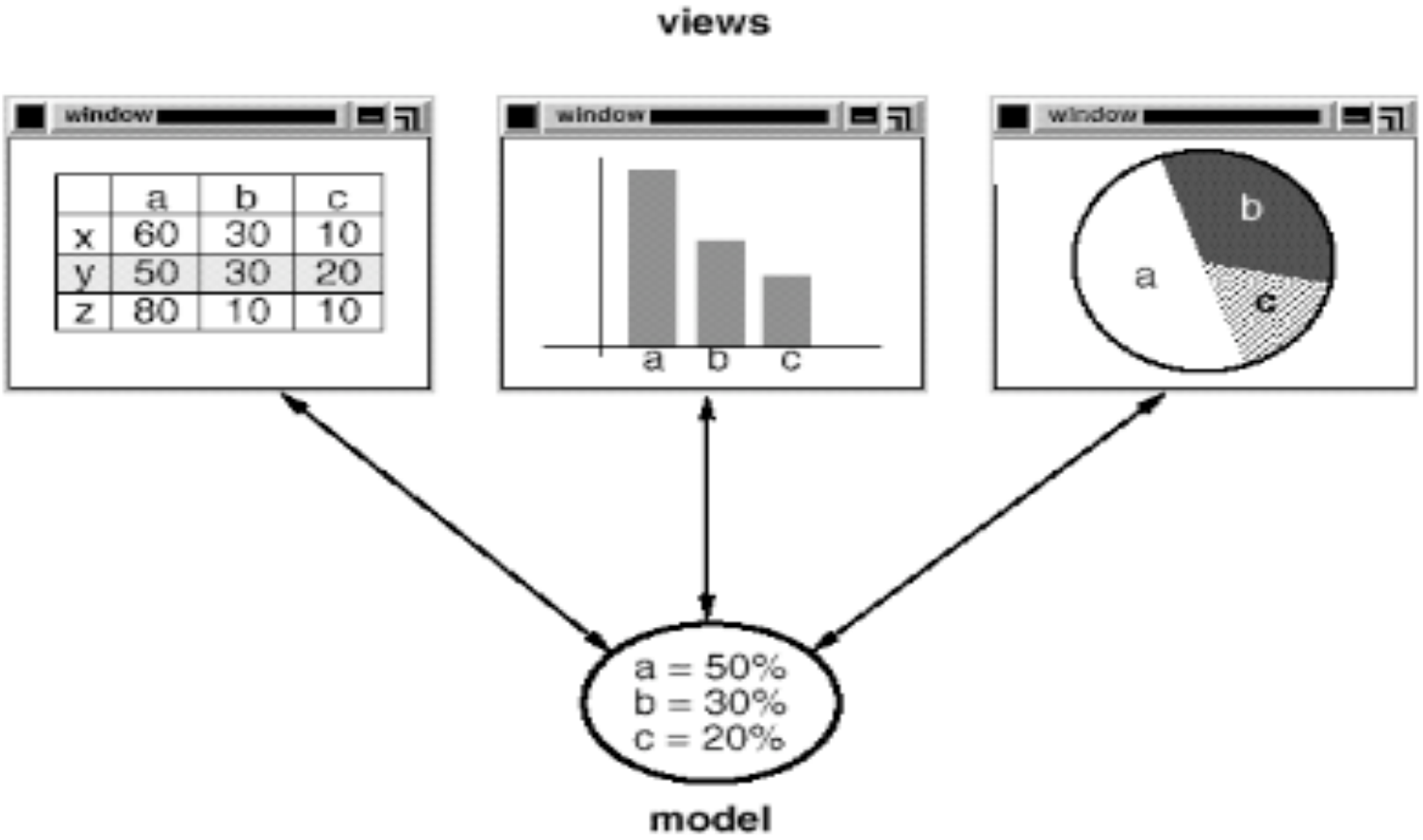
Pohled musí odpovídat stavu modelu.

Pokud se stav modelu změní, model upozorní pohledy, které na něm závisí.

Každý pohled má možnost upravení sebe sama.

Tento přístup umožňuje existenci mnoha různých pohledů nad daným modelem.

MVC



Vypustili jsme kontrolér pro jednoduchost

GoF - Strukturní vzory

Hlavní myšlenkou je rozdělení rozhraní a implementace.

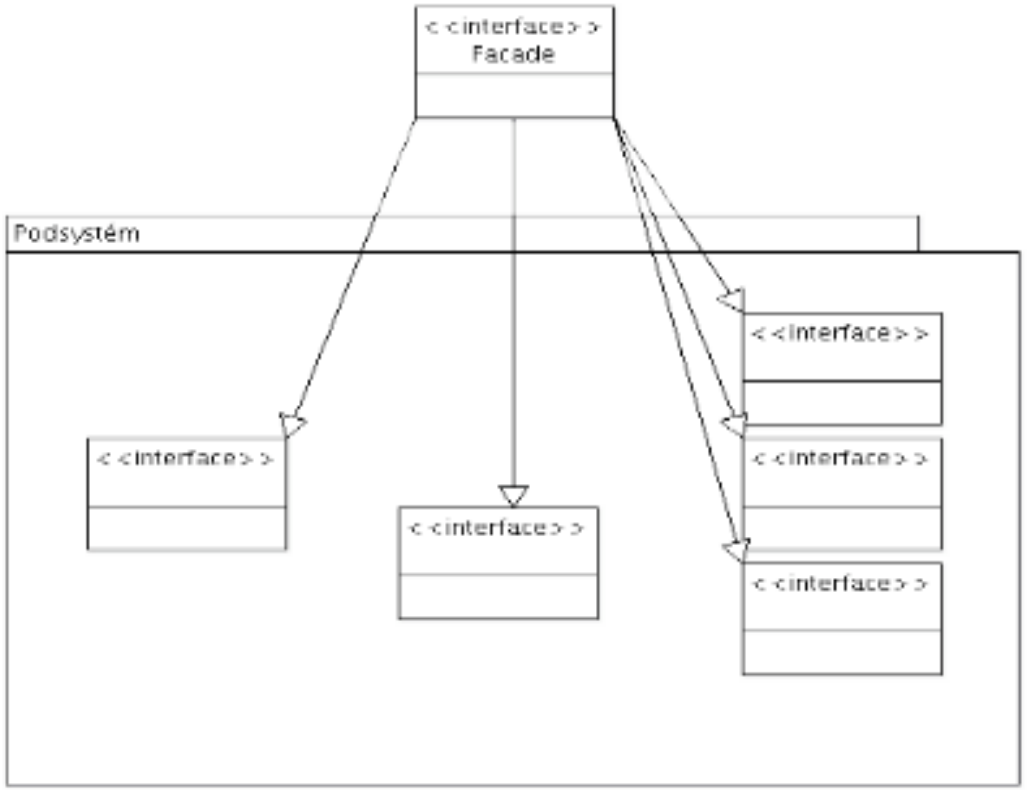
Composite: struktura pro budování rekurzivních agregací
Adapter: Překlad adaptující rozhraní servru a klienta
Bridge: Abstrakce pro spojení jedné či více implementací
Decorator: transparentní rozšíření objektu
Facade: zjednodušuje rozhraní a tedy přístup do podsystému
Flyweight: Mnoho objektů s efektivním zdílením
Proxy: Jeden objekt aproximuje jiný

GoF - Strukturní vzory

Hlavní myšlenkou je ???

Facade

Tento vzor poskytuje jednotné uživatelské rozhraní pro množinu rozhraní v podsystému.



Facade

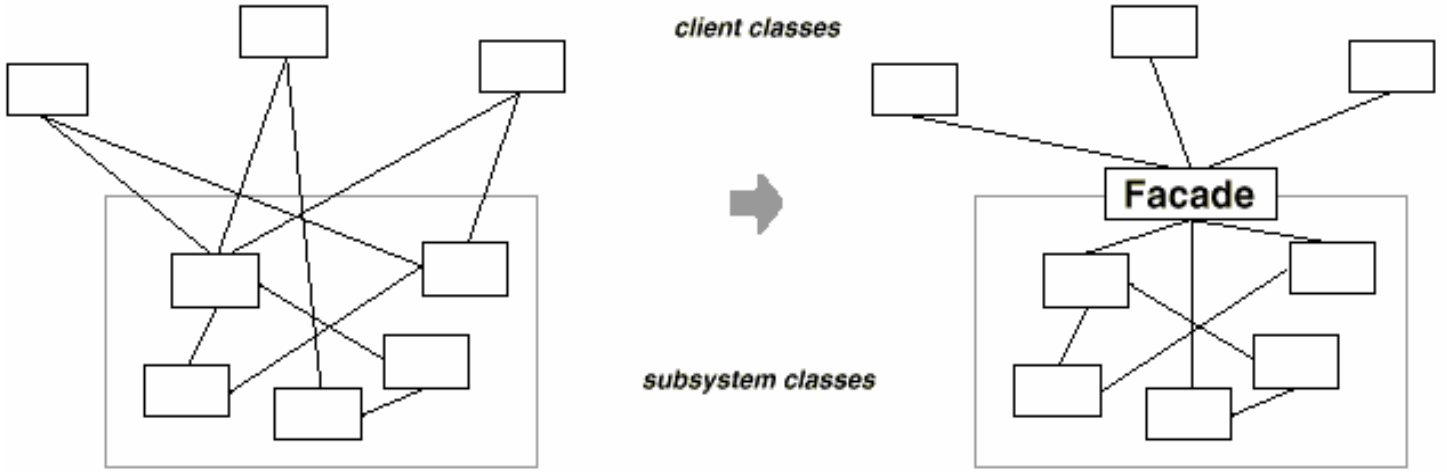
Motivace

- snadnější komunikaci mezi podsystémy
- menší závislosti mezi podsystémy
- zamezení použití vybraných metod rozhraní
- RISC = > CISC (spojení nižších metod do jedné metody)
- zjednodušení pro programátory vyšších vrstev
  - není třeba vědět co je uvnitř
  - stačí vědět co to umí

Účel

- sjednocené high-level rozhraní pro podsystém
- snadné použití podsystému (popř. restrikce)
- zapouzdření – skrytí návrhu před uživateli

Facade



Facade

- HiFi sestava
  - Sony TV
  - Panasonic zesilovač
  - Phillips reproduktor
  - Yamaha DVD

  - 4 dálkové ovladače

Facade

- HiFi sestava
  - Univerzální dálkový ovladač
    - Zapni vše
    - Spusť film

Facade

Použití

- když je subsystém složitý na běžné použití
- když existuje mnoho závislostí vně podsystému
- při vytváření vstupních bodů vrstveného systému
- zamezení přístupu k určitým funkcím

Pozorování

- Facade zná přístup do systému, jediný přístupový bod
- klient jde vždy přes Facade (role kontroléru)
- klient nezná implementace nižších vrstev

Facade

Vhodné když

- je třeba jeden přístupový bod pro služby systému
- chceme schovat interní logiku
- chceme využít jen některé funkce
- zjednodušit přístup z daného kontextu

Facade

Použití

- J2EE session bean
  - zapouzdřuje vnitřní složitost a interakce mezi business objekty.
- Session Facade využívá business objekty
  - a poskytuje jednotný přístup ke službám vrstvy.

Výhody

- jednoznačně snížení párování mezi podsystémy a vrstvami
- schování privátních metod podsystému

Nevýhody

- přílišné omezení funkcí nabízených podsystémem.

Facade

Co to je?

Adapter

Motivace

- existující rozhraní třídy je již použito v systému, nový klient ale očekává poněkud odlišné rozhraní
- v aplikaci je již existující třída, která má požadovanou funkčnost ale má špatné rozhraní
- přizpůsobení rozhraní pro existující blok

Účel

- přetváří rozhraní třídy do rozhraní jaké očekává klient

Adapter – pojmenování tříd

Client (původní aplikace)

- očekává třídy s určitým rozhraním – pro se přizpůsobuje

Target

- definuje rozhraní, které vyžaduje Client

Adaptee

- implementuje požadované operace
- alespoň z větší části má ale jiné rozhraní než Target
- kód nelze měnit nebo ho měnit nechceme

Adapter

- přizpůsobuje Adaptee na rozhraní Target

Adapter

- chceme použít cizí třídu, ale její rozhraní je jiné, než potřebujeme

Adapter

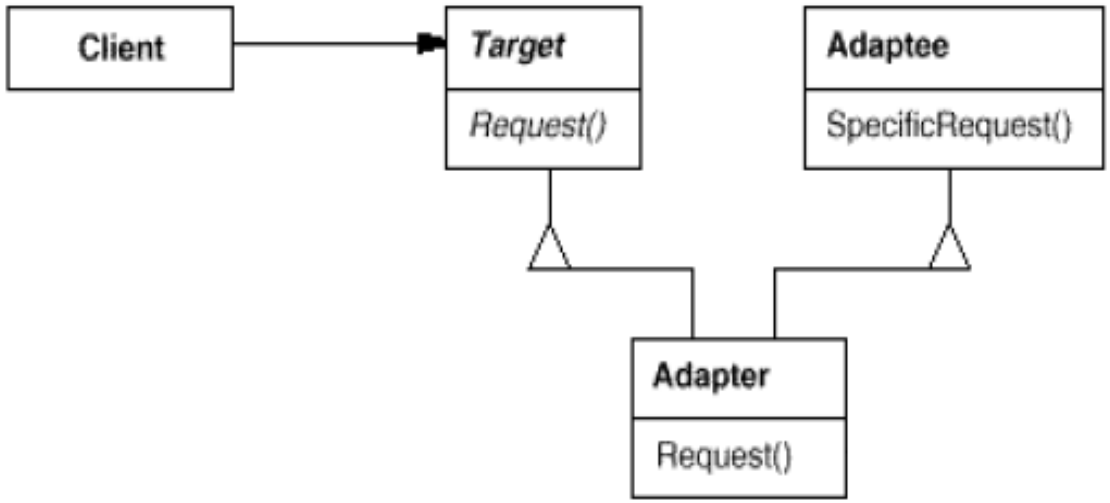
Vlastnosti

- Adaptér přetváří rozhraní třídy do rozhraní jaké očekává klient.
- Vzor tedy umožňuje spolupráci nekompatibilních tříd.
- Adaptér také může rozšířit funkčnost upravované třídy.
- Vhodné pro legacy software (předchozí verze).

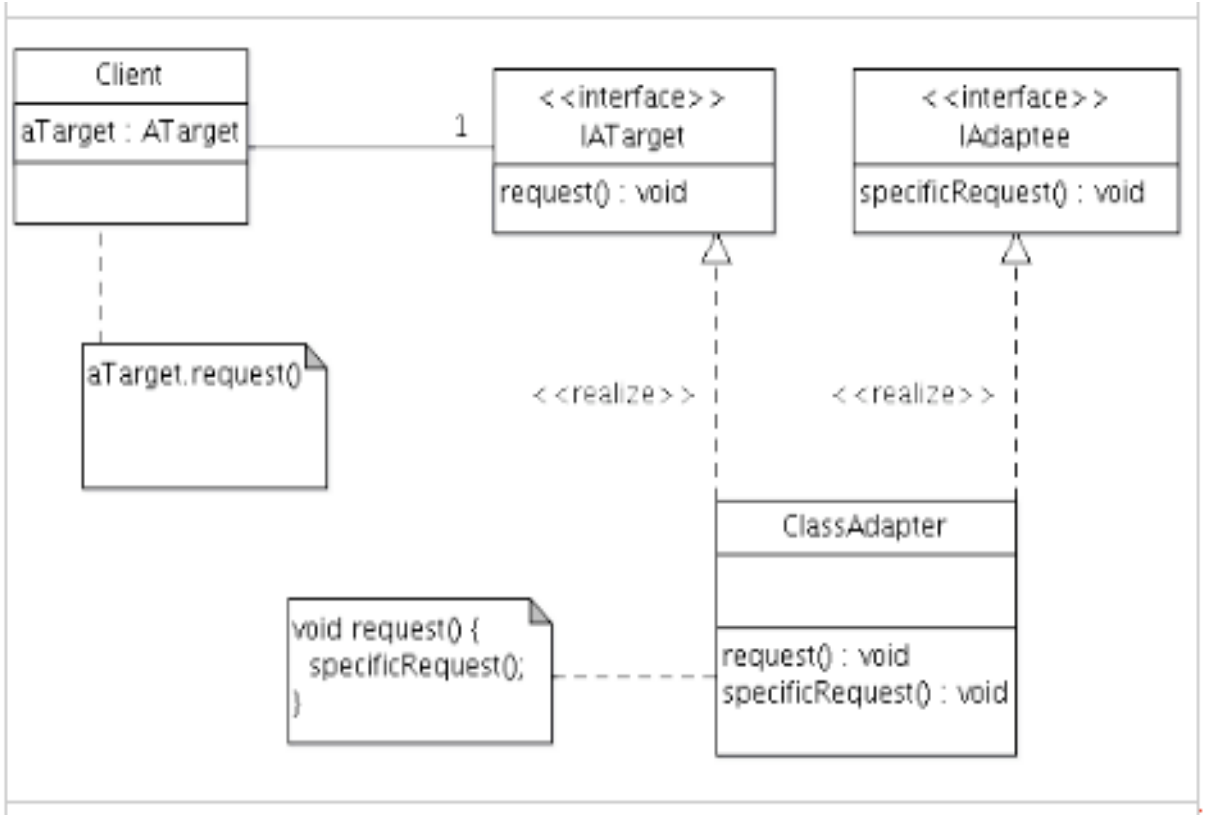
Existují dva způsoby použití tohoto vzoru

- přes objekt
- přes třídu

Adapter - přes třídu



Adapter - přes třídu

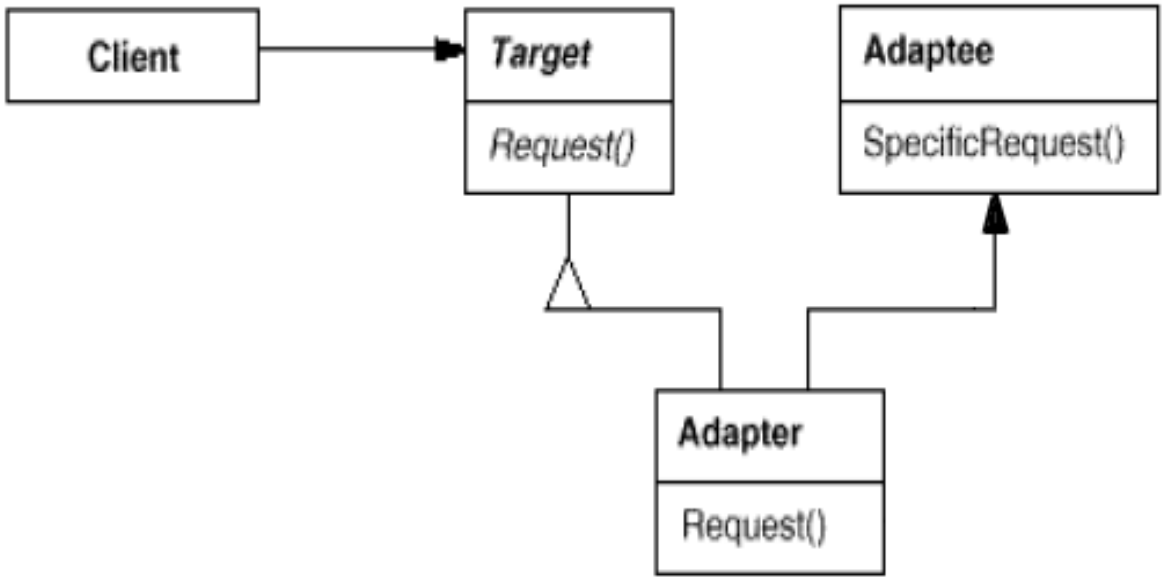


Adapter - přes třídu

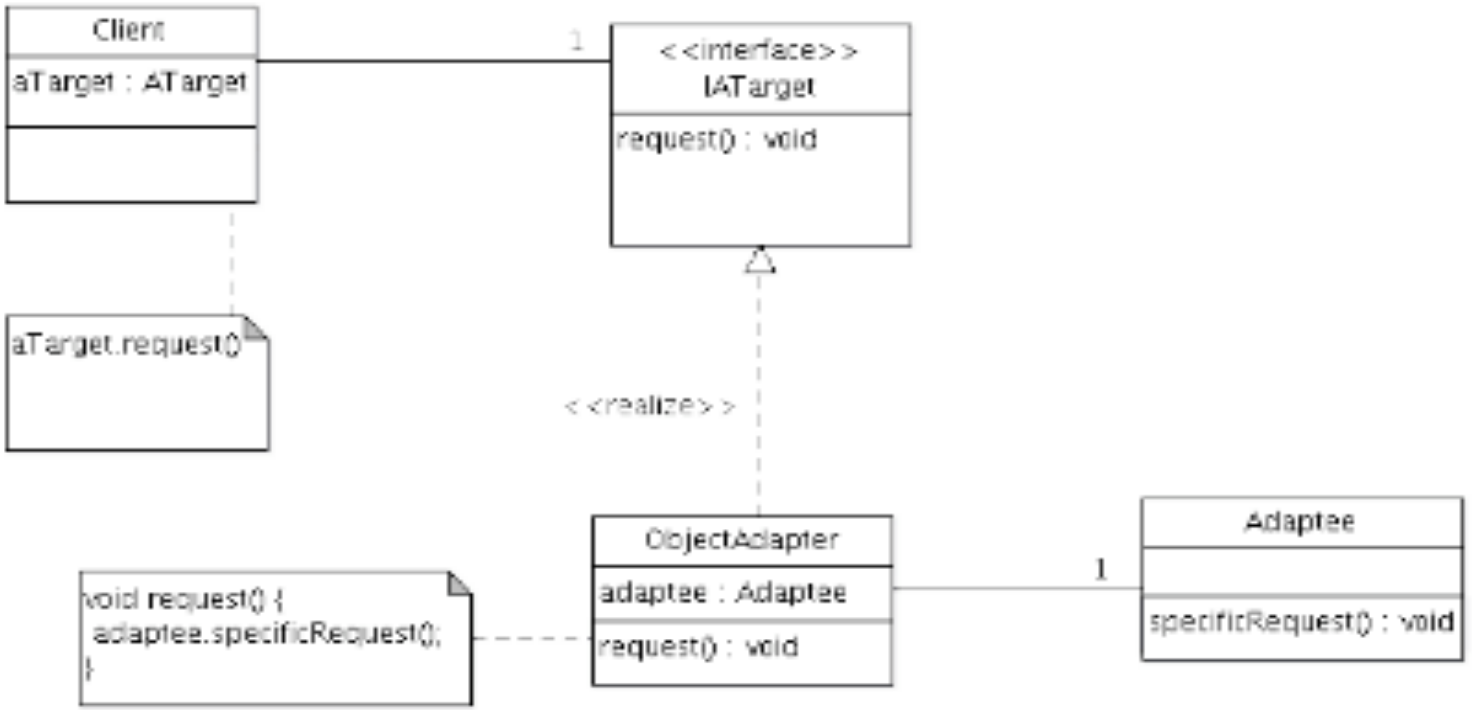
Vlastnosti

- využívá vícenásobnou dědičnost
- adaptér přes třídu replikuje kód
  - může způsobit jmenné konflikty.

Adapter - přes objekt



Adapter - přes objekt



Adapter - přes objekt

Vlastnosti

- využívá kompozici
- nevidí private a protected metody
- lze adaptovat potomky adaptee

Adapter

Důsledek

- kladným přínosem je, že lze použít nekompatibilní komponenty.
- na druhou stranu pokud adaptér nabízí jen některé funkce k uspokojení požadavku
  - další funkčnost musí byt do-implementována

Použití

- java I/O – StringBufferInputStream
- nezávislost na platformách

Adapter

Co to je?

Bridge

Účel

- odpárovat abstrakci a implementaci,
  - tak že se mohou měnit nezávisle
- použití při návrhu

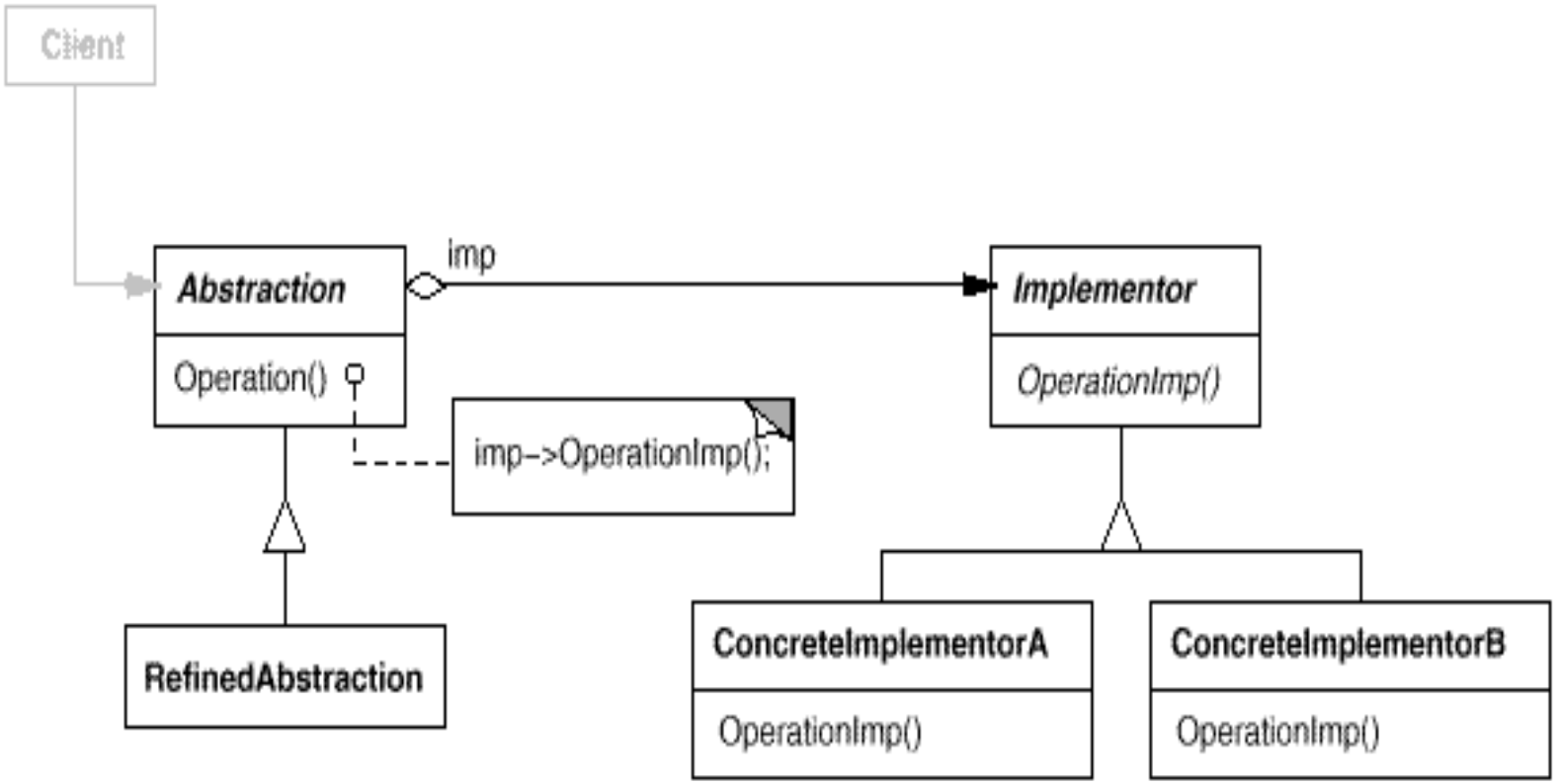
Problém

- variace v abstrakci konceptu
- variace implementaci konceptu
- skrytí detailů implementace

Řešení

- najdi co se mění a zapouzdři
- použij raději kompozici než dědičnost

Bridge - struktura



Bridge

Abstraction

- definuje rozhraní objektů, obsahuje odkaz na implementaci
- metody realizovány pomocí metod rozhraní *Implementor*

Implementor

- představuje rozhraní implementací, rozhraní se může lišit od *Abstraction*

RefinedAbstraction

- rozšiřuje rozhraní definované v *Abstraction*

ConcreteImplementor

- konkrétní implementace rozhraní *Implementor*

Bridge

Vlastnosti

- odděluje rozhraní a implementaci
- variace implementace nezávislá na rozhraní
- snadná rozšiřitelnost
- skrytí implementace
- lepší strukturovanému objektovému návrhu

Usnadní

- přidání nové implementace
- přidání nového potomka *Abstraction*

Bridge

Kdy použít

- pokud máme abstrakci s různými implementacemi
- pokud chceme umožnit změny implementace a abstrakce nezávisle na sobě

Bridge

Postup

1. identifikuj struktury co se nebudou časem měnit (common concepts)
2. identifikuj struktury které se asi budou měnit časem (variable concepts)
3. reprezentuj neměnné koncepty jako abstraktní třídy
4. implementuj proměnné koncepty jako konkrétní třídy

Dále

- najdi co se mění a zapouzdři
- upřednostni kompozici před děděním

Bridge

Důsledek

- výhodou je nezávislost abstraktního rozhraní a implementace
- dynamická změna implementace

- nevýhoda je, že jedno rozhraní pro abstrakce a implementace musí sedět všem podomkům

Použití

- ET++ Window/WindowPort
- Libg++ Set/LinkedList
- AWT Component/ComponentPeer

Bridge

Co to je?

GoF - Vzory chování

- **Chain of responsibility**: požadavek vyslán do zodpovědné obsluhy
- **Observer**: one-to-many závislost mezi objekty, tak že změna stavu jednoho způsobí oznámení všem ostatním
- **Command**: požadavek jako prvotřídní objektového
- **Interpreter**: jazykový překlad pro menší gramatiky
- **Iterator**: k nasbíraným elementům přistupujeme sekvenčně
- **Mediator**: koordinuje interakce mezi přiřazenými objekty

GoF - Vzory chování

Memento: snapshot zachytí a obnoví soukromé stavy objektů
State: objekt jehož chování závisí na jeho stavu
Strategy: abstrakce pro výběr jedné ze strategií
Template Method: algoritmus s kroky dodanými odvozenou třídou
Visitor: operace aplikovány na elementy heterogenní objektové struktury

Vzory pro tvorbu instancí a tříd

- Factory method
- Abstract Factory
- Prototype
- Builder
- Singleton

Odkazy

Kniha

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, November 2000, ISBN 0-20-63361-2

Přednášky

- Baylor, Dr. Song
- MFF UK, Dr. Zavoral