
Návrhové vzory



Návrhové vzory

■ OO jazyky - široká paleta technických prostředků

- dědičnost, polymorfismus, šablony, reference, přetěžování, ...
- problém - jak toto všechno efektivně používat
- cíl - udržovatelný a rozšiřovatelný 'velký' software
 - rozhraní !!
 - volnější vazby, parametrizace
 - dědičnost implementace vs. dědičnost rozhraní
 - dědičnost vs delegace



■ Návrhový vzor

- pojmenované a popsané řešení typického problému
- principiálně existují již dlouho
 - architektura: Christopher Alexander - pojem 'Pattern'
 - literatura: tragický hrdina, romantická (tele)novela, ...



Návrhové vzory v software

■ Software

- asi žádný jiný obor si nelibuje ve vynalézání kola stále znovu
- strukturovaný přístup
 - spojové seznamy, stromy, rekurze, ...
- OOP - systém reusabilních návrhových vzorů (NV)

■ Co má NV pro typickou situaci popisovat

- jak a kdy mají být objekty vytvářeny
- jaké vztahy a struktury mají obsahovat třídy
- jaké chování mají mít třídy, jak mají spolupracovat objekty



Definice a použití

Návrhový vzor je popis komunikujících objektů a tříd
uzpůsobených k řešení obecného problému v konkrétním kontextu

■ Relativní komplexnost a obecnost

- pro rozsáhlejší systémy
 - předpoklad dlouhé životnosti, údržby a rozšiřování
- při návrhu nových systémů
- při rozsáhlých úpravách

■ Inženýrský přístup

- přehled o existenci a typickém použití
- při návrhu hledat uplatnění

■ 'Revouční' myšlenka GoF

- vytvoření utříděného katalogu 23 vzorů ve 3 kategoriích
- v současnosti množství dalších vzorů
 - často pro specializované použití



Základní prvky

■ **Název**

- co nejvíce vystihující podstatu, usnadnění komunikace - společný **slovník**

■ **Problém**

- obecná situace kterou má NV řešit, podmínky použití

■ **Řešení**

- soubor pravidel a vztahů popisujících jak dosáhnout řešení problému
- nejen statická struktura, ale i dynamika chování

■ **Souvislosti a důsledky**

- detailní vysvětlení použití, implementace a principu fungování
- způsob práce s NV v praxi

■ **Příklady**

- definice konkrétního problému, vstupní podmínky, popis implementace a výsledek

■ **Související vzory**

- použití jednoho NV nepředstavuje typicky ucelené řešení - řetězec NV
- okolnosti pro rozhodování mezi různými NV



Kategorie základních NV

	Creational <i>Tvořivé vzory</i>	Structural <i>Strukturální vzory</i>	Behavioral <i>Vzory chování</i>
Třída	Factory Method	Adapter	Interpreter Template Method
Objekt	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

vytváření objektů

uspořádání tříd a objektů

chování a interakce objektů a tříd



Kategorie základních NV

	Creational <i>Tvořivé vzory</i>	Structural <i>Strukturální vzory</i>	Behavioral <i>Vzory chování</i>
Třída	Factory Method	Adapter	Interpreter Template Method
Objekt	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

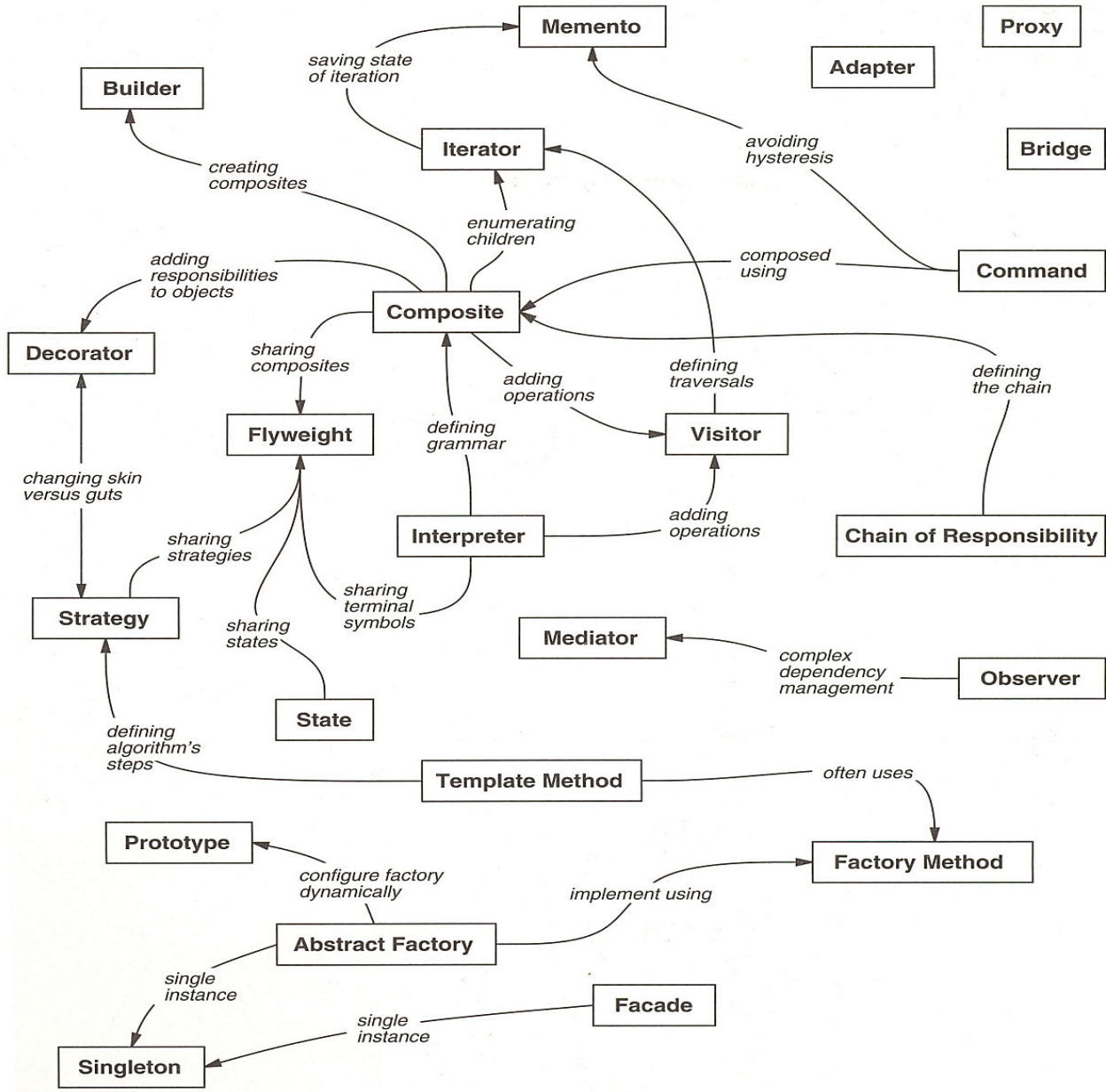
vytváření objektů

uspořádání tříd a objektů

chování a interakce objektů a tříd

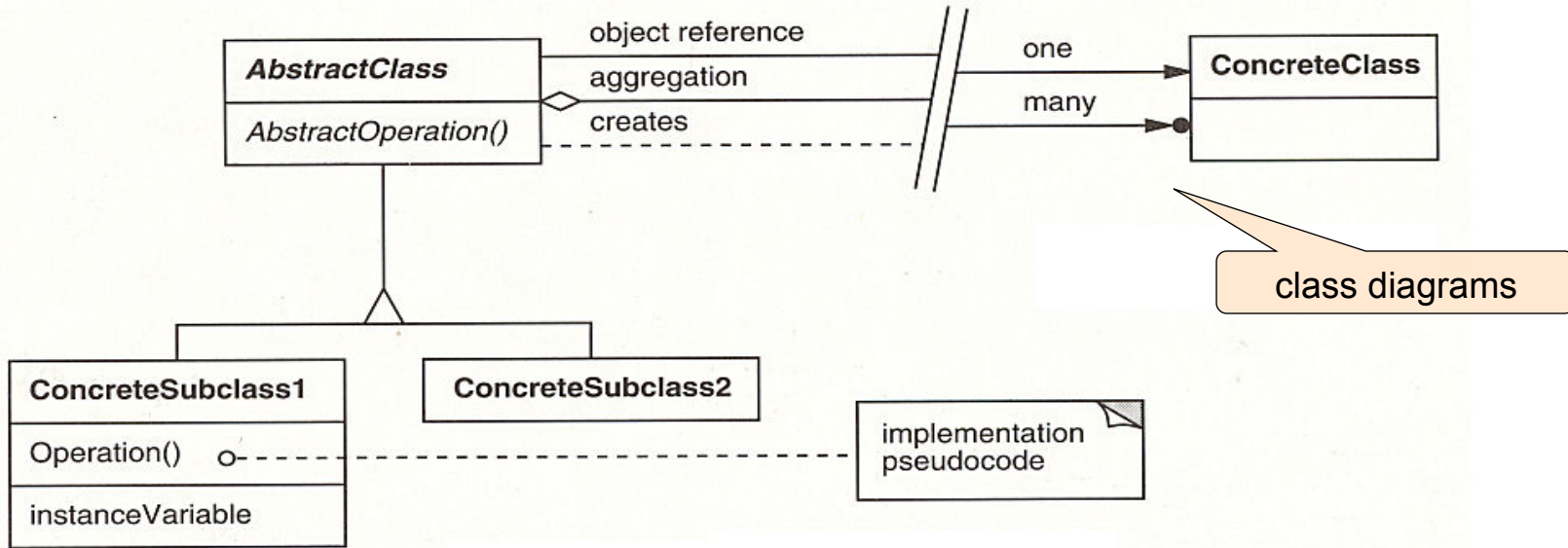


Vztahy mezi NV

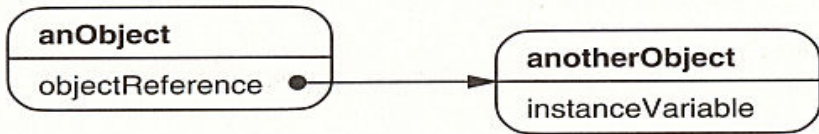




Značení – Object Modeling Technique

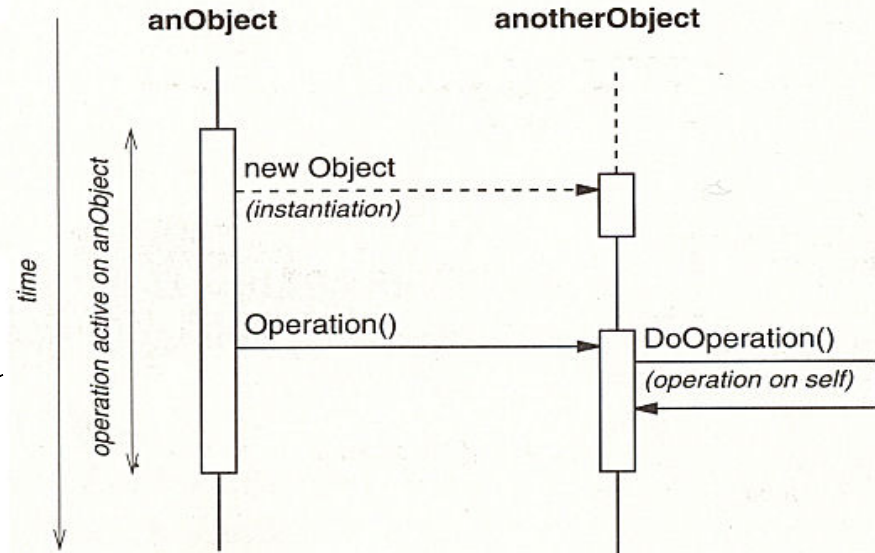


class diagrams



object diagrams

interaction diagrams





■ Creational Patterns

■ Abstrakce procesu vytváření objektů

- umožňují ovlivnit způsob vytváření objektů a jejich počet
- často nestačí použít *new*, např. pokud typ objektu závisí na parametrech

■ Užitečné při převažující objektové kompozici (místo dědičnosti)

- místo napevno naprogramovaného chování množina obecnějších metod
- větší flexibilita **co** se vytváří, **kdo** to vytváří, **jak** a **kdy** se to vytváří

■ Typické prostředky

- zapouzdření znalosti o použití konkrétní třídy
- zakrytí vzniku a skládání objektů

■ Tvořivé vzory

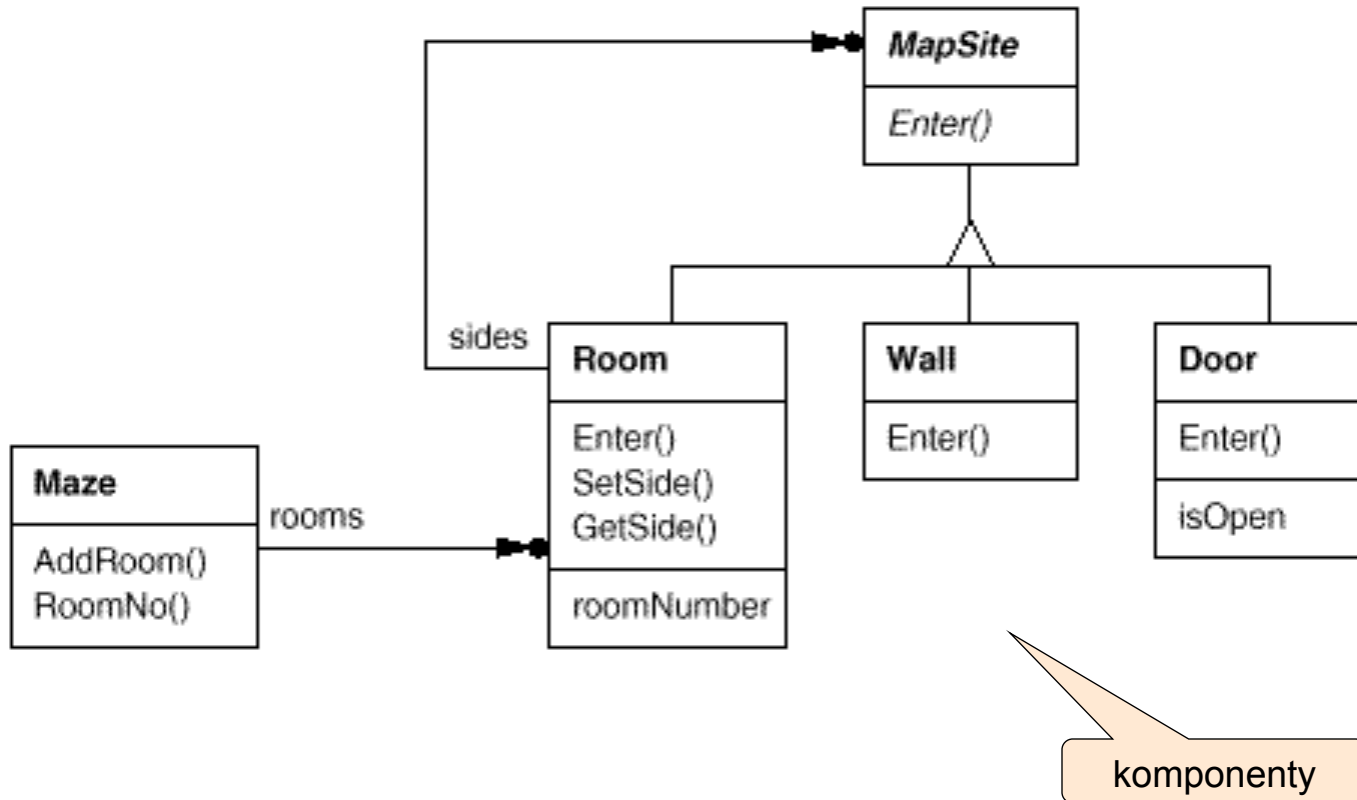
- **Singleton** - zaručí pouze jednu instance třídy
- **Factory Method** - vytváří instance vybrané třídy - virtuální funkce místo *new*
- **Abstract Factory** - vytváří objekty pro vybranou skupinu tříd - tovární třída
- **Builder** - odděluje způsob vytvoření objektu od reprezentace, postupné vytváření
- **Prototype** - umožňuje zkopírovat (klonovat) inicializovanou instanci



Bludiště

■ Jednotný příklad pro tvořivé NV - vytvoření bludiště

- množina místností, místnost zná své sousedy - zeď, jiná místnost nebo dveře





Bludiště - prvotní implementace

```
import static System.*;

enum Direction {North, South, East, West};

interface MapSite {
    void enter();
};

class Room implements MapSite {
    MapSite sides[4];
    int roomNumber;

    Room(int roomNo) {
        roomNumber = roomNo;
    }

    MapSite getSide(Direction d) {...};
    void setSide(Direction d, MapSite m){...};
    void enter() {...};
};
```

Enter - soused je:
místnost nebo otevřené dveře → projít 😊
zeď nebo zavřené dveře → stát ☹

```
class Wall implements MapSite {
public:
    Wall() {};
    void enter() {};
};

class Door implements MapSite {
    Room room1;
    Room room2;
    boolean isOpen;

    Door(Room r1, Room r2) {};

    void enter() {};
    Room otherSideFrom(Room r) {};
};

class Maze {
    Maze() {out.print("creating")}

    void addRoom(Room r) {};
    Room roomNo(int) {};
};
```



Bludiště - prvotní implementace

```
import static System.*;

enum Direction {North, South, East, West};

interface MapSite {
    void enter();
};

class Room implements MapSite {
    MapSite sides[4];
    int roomNumber;

    Room(int roomNo) {
        roomNumber = roomNo;
    }

    final MapSite getSide(Direction d) {...};
    final void setSide(..){...};

    void enter() {...};
};
```

Enter - soused je:
místnost nebo otevřené dveře → projít 😊
zeď nebo zavřené dveře → stát ☹

```
class Wall implements MapSite {
public:
    Wall() {};
    void enter() {};
};

class Door implements MapSite {
    Room room1;
    Room room2;
    boolean isOpen;

    Door(Room r1, Room r2) {};

    void enter() {};
    final Room otherSideFrom(..) {};
};

class Maze {
    Maze() {out.print("creating")}

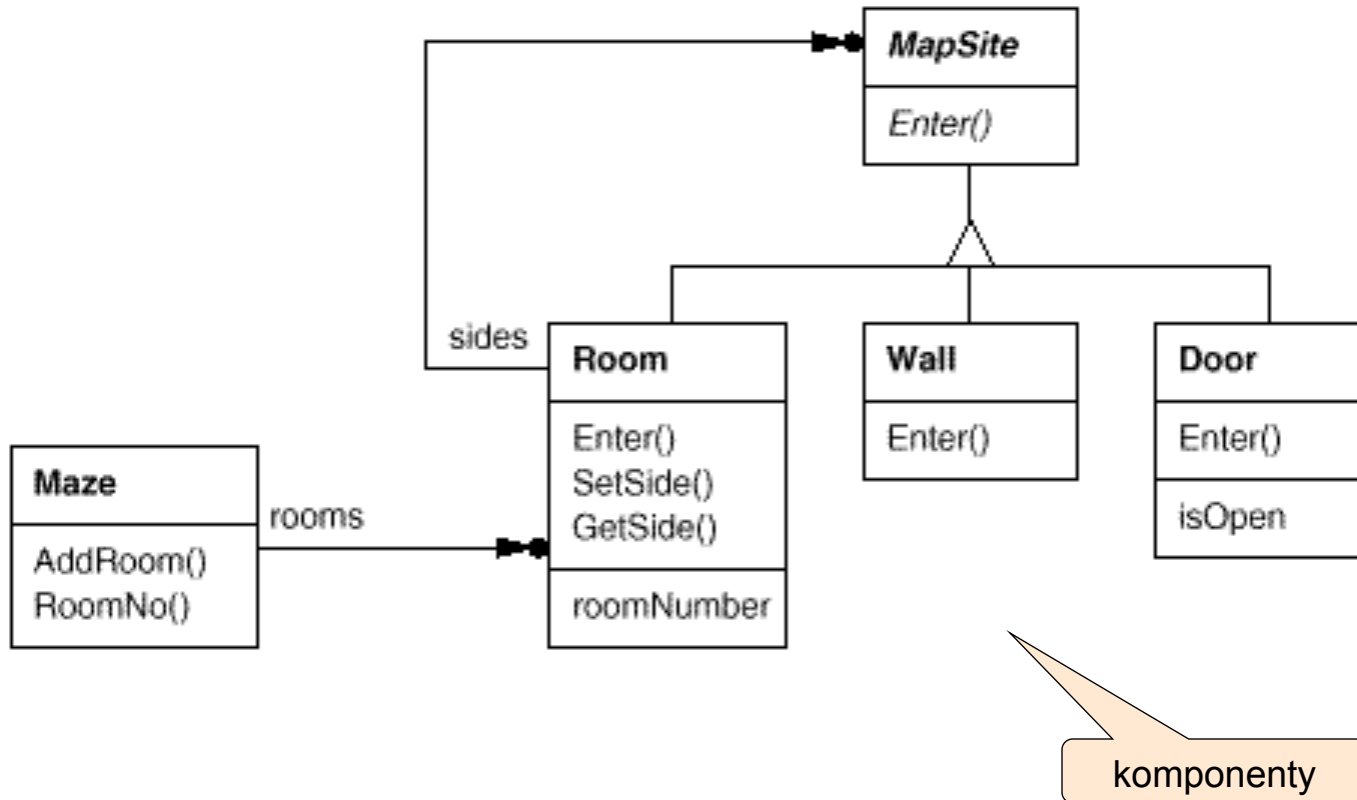
    final void addRoom(Room r) {};
    final Room roomNo(int) {};
};
```



Bludiště

■ Jednotný příklad pro tvořivé NV - vytvoření bludiště

- množina místností, místnost zná své sousedy - zeď, jiná místnost nebo dveře





Bludiště - vytvoření

```
import static package.Direction.*;

class MazeGame {
Maze createMaze () {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door theDoor = new Door(r1, r2);

    maze.addRoom(r1);
    maze.addRoom(r2);

    r1.setSide(North, new Wall());
    r1.setSide(East, theDoor);
    r1.setSide(South, new Wall());
    r1.setSide(West, new Wall());

    r2.setSide(North, new Wall());
    r2.setSide(East, new Wall());
    r2.setSide(South, new Wall());
    r2.setSide(West, theDoor);

    return maze;
}
```

vytvoření bludiště se 2 místnostmi

místnosti a dveře mezi nimi

hranice místností

- poměrně komplikované
- neflexibilní !!
 - změna tvaru - změna metody
 - změna chování - nutnost přepsání
 - DoorNeedingSpell, SpecialRoom
- hard coded 💣*



Možná vylepšení pomocí tvořivých NV

Zvýšení flexibility - odstranění explicitních referencí na konkrétní třídy

■ Singleton

- zaručí jedinečnost instance bludiště a přístup k ní bez potřeby globálních dat

■ Factory Method

- `CreateMaze` při vytváření komponent volá virtuální funkci místo konstruktoru
- potomek `MazeGame` může změnou virtuální funkce vytvářet instance jiných tříd

■ Abstract Factory

- `CreateMaze` dostane parametr objekt pro vytváření komponent
- možnost změny instanciovanych tříd předáním jiného parametru

■ Builder

- `CreateMaze` dostane parametr objekt s operacemi pro přidávání komponent
- pomocí dědičnost lze změnit jednotlivé vytvářené části nebo způsob vytváření

■ Prototype

- `CreateMaze` dostane parametry prototypy objektů které se umí klonovat
- možnost změny předáním jiných (poděděných) parametrů



Konkrétní tvořivé NV

■ Na samostatných slajdech

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype



Tvořivé NV - shrnutí

■ Singleton

- jednoduché použití pokud není nutné řešit destrukci objektů
- různé varianty pro vzájemně provázané objekty

■ Factory Method

- flexibilita za relativně malou cenu - obvyklá základní metoda, virtual constructor
- nevýhoda: nutnost dědičnosti i pro změnu třídy produktu
 - Product/ConcreteProduct, Creator/ConcreteCreator - šablony / generics
- odložená inicializace

■ Abstract Factory, Builder, Prototype

- flexibilnější, složitější - použití až při zjištění potřeby větší flexibility
- kompozice objektů
 - továrního objektu zodpovědný za znalost třídy produktů a jejich výrobu
- Abstract Factory
 - tovární objekt vytváří objekty více souvisejících tříd
- Builder
 - tovární objekt vytváří složený objekt postupně pomocí odpovídajícího protokolu
- Prototype
 - nové objekty se vytvářejí kopírováním prototypových objektů
 - prototypy se umějí samy klonovat



Tvořivé NV - srovnání

```
Maze createMaze() {  
    Maze maze = makeMaze();  
    Room r1 = makeRoom(1);  
    Room r2 = makeRoom(2);  
    Door theDoor = makeDoor(r1, r2);  
}
```

```
Maze createMaze( MazeFactory factory) {  
    Maze maze = factory.makeMaze();  
    Room r1 = factory.makeRoom(1);  
    Room r2 = factory.makeRoom(2);  
    Door door = factory.makeDoor(r1, r2);  
}
```

```
Maze createMaze( MazeBuilder builder) {  
    builder.buildMaze();  
    builder.buildRoom(1);  
    builder.buildRoom(2);  
    builder.buildDoor(1, 2);  
    return builder.getMaze();  
}
```

Factory Method

Abstract Factory

Builder

```
MazeProtoFactory simpleMazeFactory =  
    new MazeProtoFactory (  
        new Maze(), new Wall(), new Room(), new Door()  
    );  
Maze maze = game.createMaze( simpleMazeFactory);
```

Prototype

```
MazeProtoFactory {  
    Wall makeWall () {  
        return protoWall.clone();  
    }  
  
    Door makeDoor (Room r1, Room r2) {  
        Door door = protoDoor.clone();  
        door.initialize(r1, r2);  
        return door;  
    }  
}
```



Strukturální NV

■ Structural Patterns

- jak jsou třídy a objekty složeny do větších struktur

■ Strukturální NV tříd

- dědičnost pro skládání rozhraní nebo implementací
- **Adapter** - přizpůsobení rozhraní třídy jiným rozhraním

■ Strukturální NV objektů

- skládání objektů pro dosažení nové funkcionality
- runtime skládání - větší flexibilita
- **Bridge** - lepší separace rozhraní a implementace
- **Facade** - reprezentace celého systému jedním objektem, jednotné rozhraní
- **Proxy** - zástupce jiného objektu
- **Decorator** - dynamické přidávání funkčnosti k objektům
- **Composite** - hierarchie tříd tvořená dvěma druhy objektů - primitivní a složené
- **Flyweight** - efektivní struktura pro velké množství sdílených objektů



Konkrétní strukturální NV

Na samostatných slajdech

□ Adapter

- přizpůsobení rozhraní třídy na rozhraní jiné třídy, spolupráce tříd s různým rozhraním

□ Bridge

- odděluje abstrakci od implementace
- předchází nárůstu počtu tříd při přidávání implementací

□ Facade

- definuje jedno společné rozhraní pro subsystém

□ Proxy

- zástupce/náhradník objektu, kontrola přístupu k objektu

□ Decorator

- rozšiřuje objekt o nové vlastnosti
- dynamický (dědění=statické), transparentní (rozšiřovaný objekt nic neví)

□ Composite

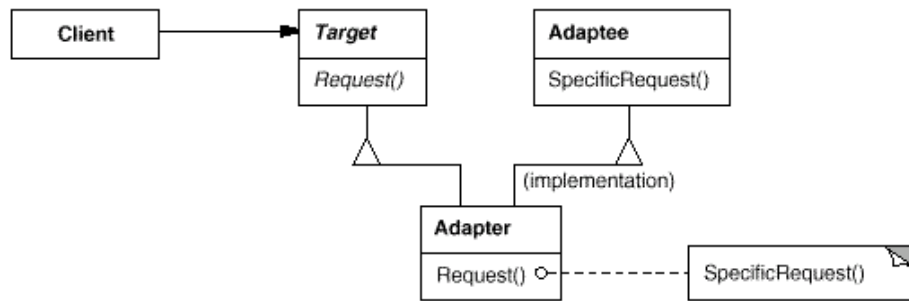
- jak postavit hierarchii tříd složenou ze dvou druhů objektů: primitivní a složené
- složené objekty se rekurzivně skládají z primitivních a dalších složených objektů

□ Flyweight

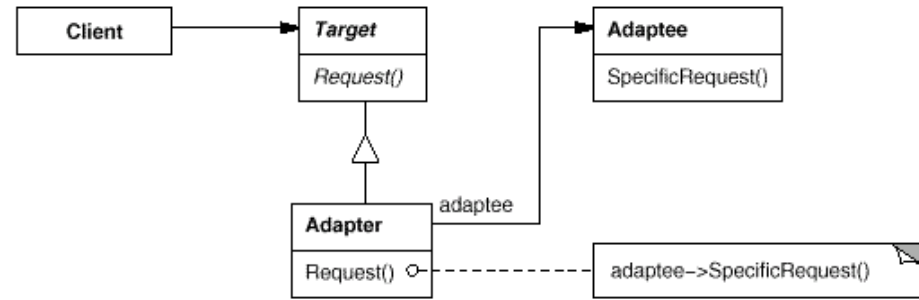
- podpora většího počtu jednoduchých objektů



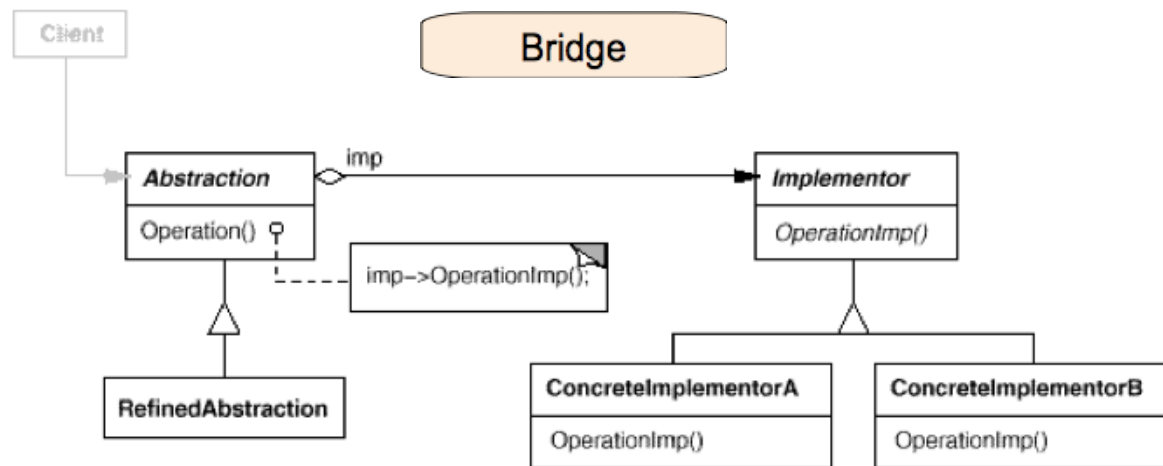
Adapter vs. Bridge



Class Adapter



Object Adapter



Bridge



Adapter vs. Bridge

■ Společné vlastnosti

- flexibilita - stupeň indirekce vůči jiným objektům, zasílání zpráv přes jiné rozhraní

■ Základní rozdíl - účel

□ Adapter

- vyřešení nekompatibilit mezi dvěma existujícími rozhraními
- jak zajistit aby dvě nezávislé třídy mohly spolupracovat bez reimplementace

□ Bridge

- poskytuje relativně stabilní rozhraní pro potenciálně velký počet implementací
- zachovává rozhraní i při dalším vývoji a změně implementačních tříd

■ Důsledek: časté použití při různých fázích vývojového cyklu

□ Adapter

- při nepředvídané (pozdější) potřebě spolupráce dvou nekompatibilních tříd
- použití **PO** návrhu

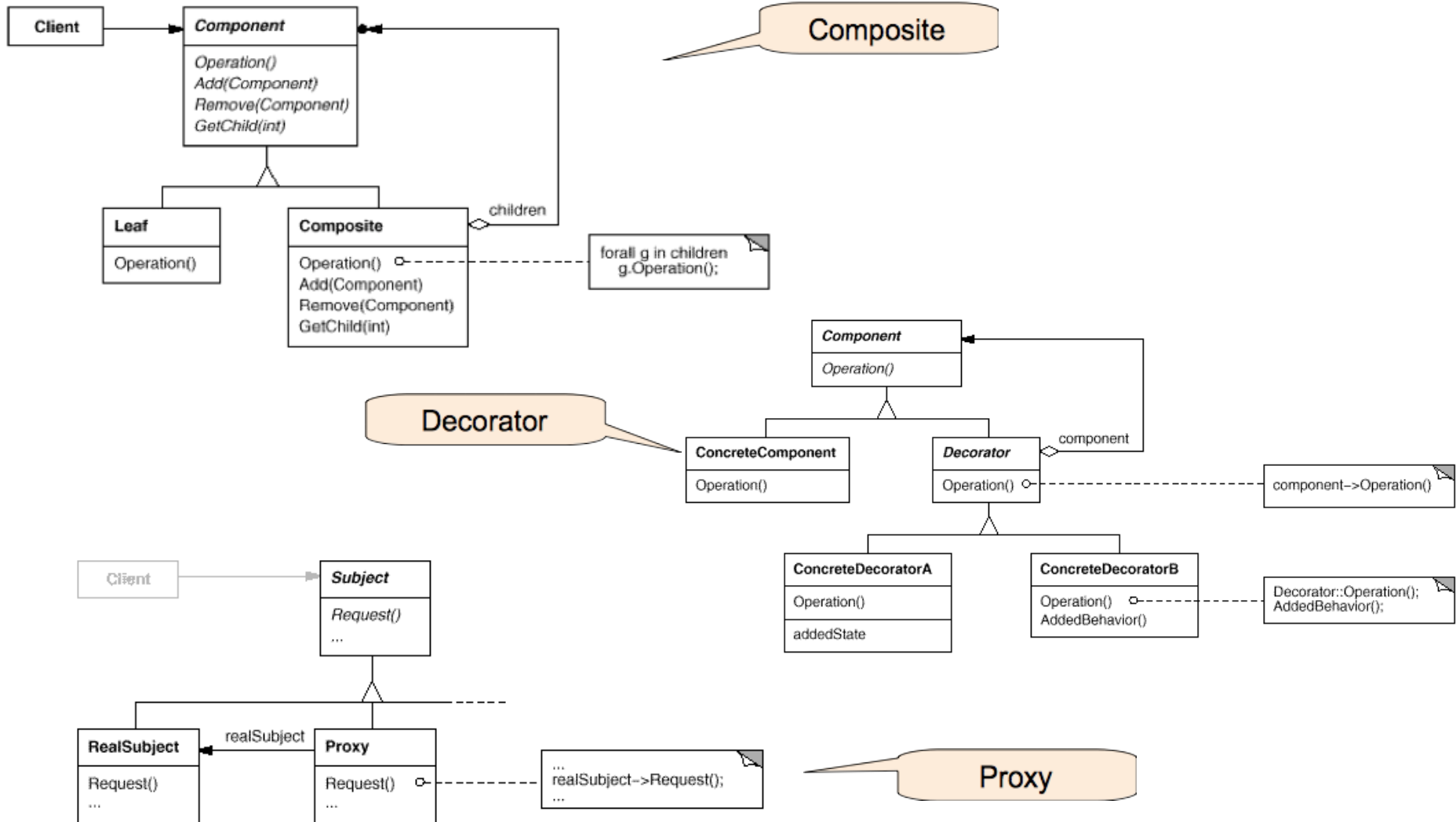
□ Bridge

- při poznání, že abstrakce může mít více implementací, které se můžou dále vyvíjet
- použití **PŘI** návrhu

- ... což neznamená, že Adapter je méně hodnotný než Bridge, jen řeší jiný problém



Composite vs. Decorator vs. Proxy





Composite vs. Decorator vs. Proxy

■ Composite a Decorator - podobná struktura

- rekurzivní struktura - Decorator jako speciální případ Compositu?
- ne - různé účely
- Decorator
 - zabraňuje explozi odvozených tříd při přidávání kombinací funkčnosti
- Composite
 - struktura - objekty různého druhu (vč. složených) se zpracovávají jednotně
 - zaměřen na reprezentaci objektů, ne na zdobení

■ Proxy a Decorator - podobná struktura

- Proxy
 - stupeň indirekce pro přístup k objektu - jednotné rozhraní
 - implementace obsahují reference na jiný objekt, kterému zasílají zprávy
 - není určen k dynamickému přidávání a odstraňování vlastností
 - není určen k reprezentaci rekurzivní struktury
- Decorator
 - component poskytuje pouze část funkčnosti
 - celková funkčnost objektu nemusí být určena v době kompilace
 - základní rys Decoratoru - neomezenost pomocí rekurzivní struktury



NV chování

■ Behavioral design patterns

- rozdělení funkčnosti a zodpovědnosti mezi objekty
- komunikace mezi objekty
- složitější struktura provádění kódu
- umožňuje zaměřit se při návrhu na propojení tříd, ne na běhové technické detaily
- dynamické vztahy - RT vlastnosti
- vzájemná provázanost

■ Behavioral class patterns

- použití dědičnosti pro rozložení chování mezi třídy
- **Template method**
 - abstraktní definice algoritmu po jednotlivých krocích
 - každý krok je buď primitivní nebo abstraktní operace definovaná v odvozených třídách
- **Interpreter**
 - reprezentace gramatiky jako hierarchie tříd
 - implementace interpretru jako operace na objektech



NV chování

■ Behavioral object patterns

- spolupráce mezi skupinami objektů pro dosažení funkčnosti

■ Objektová kompozice místo dědičnosti

□ Mediator

- odstraňuje nutnost referencí na všechny spolupracující objekty

□ Chain of Responsibility

- zasílání zpráv neznámým objektům přes zřetězené objekty

□ Observer

- definování závislosti objektu k více objektům, šíření události k závislým objektům

■ Zapouzdření chování objektu a řízení přístupu

□ Strategy

- zapouzdření funkčnosti algoritmu do objektu, možnost jejich záměny

□ Command

- zapouzdření požadavku na funkci, oddělení požadavku a vykonání funkce

□ State

- zapouzdření stavu, možnost změny chování objektu při změně stavu

□ Visitor

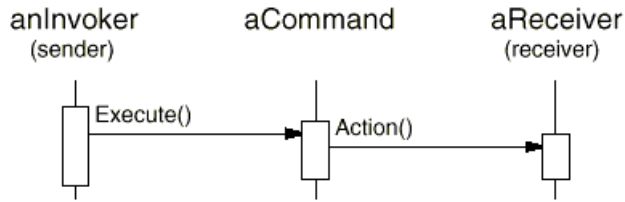
- zapouzdření chování, které by jinak bylo rozloženo mezi více tříd

□ Iterator

- abstrakce procházení agregovaných objektů

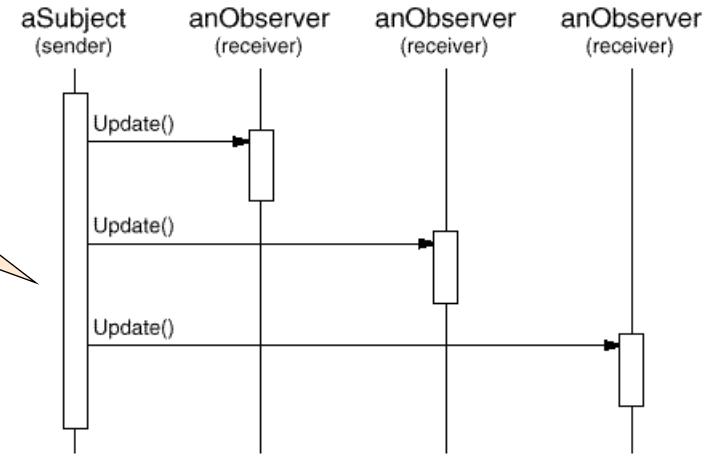


Vztahy mezi odesílateli a příjemci zpráv



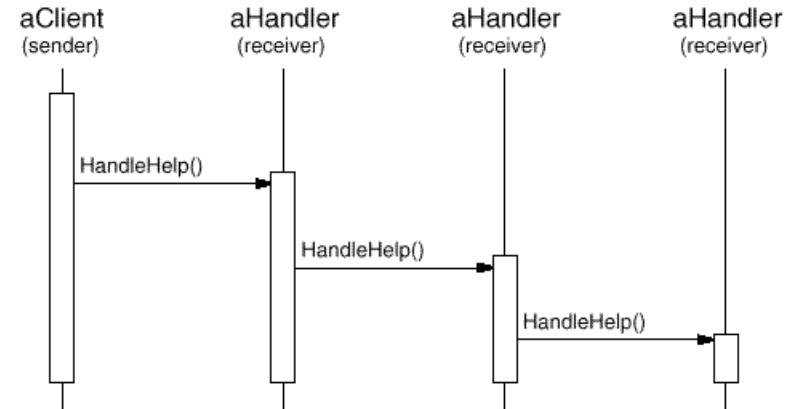
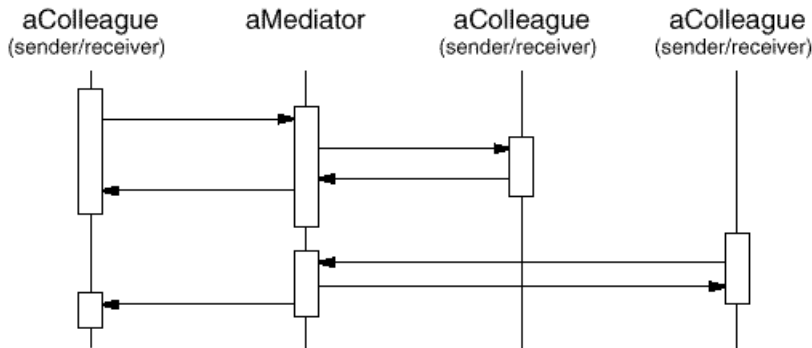
Comand
Separace příjemce (a parametrů)
od okamžiku volání

Observer
neznámý počet
příjemců



Mediator
centralizace komunikace
přes prostředníka

Chain of Responsibility
neznámá struktura příjemců i
v okamžiku volání





Závěr

Shrnutí

- ❑ 'Žádné velké moudro'
 - ... jak pro koho
- ❑ Slovník!
- ❑ Implementace bez vymyšlení kola
 - ... a 'bez chyb'
- ❑ Kompozice NV
- ❑ Generické implementace
- ❑ Mnoho dalších rozšiřujících vzorů
 - často cíleně zaměřených

