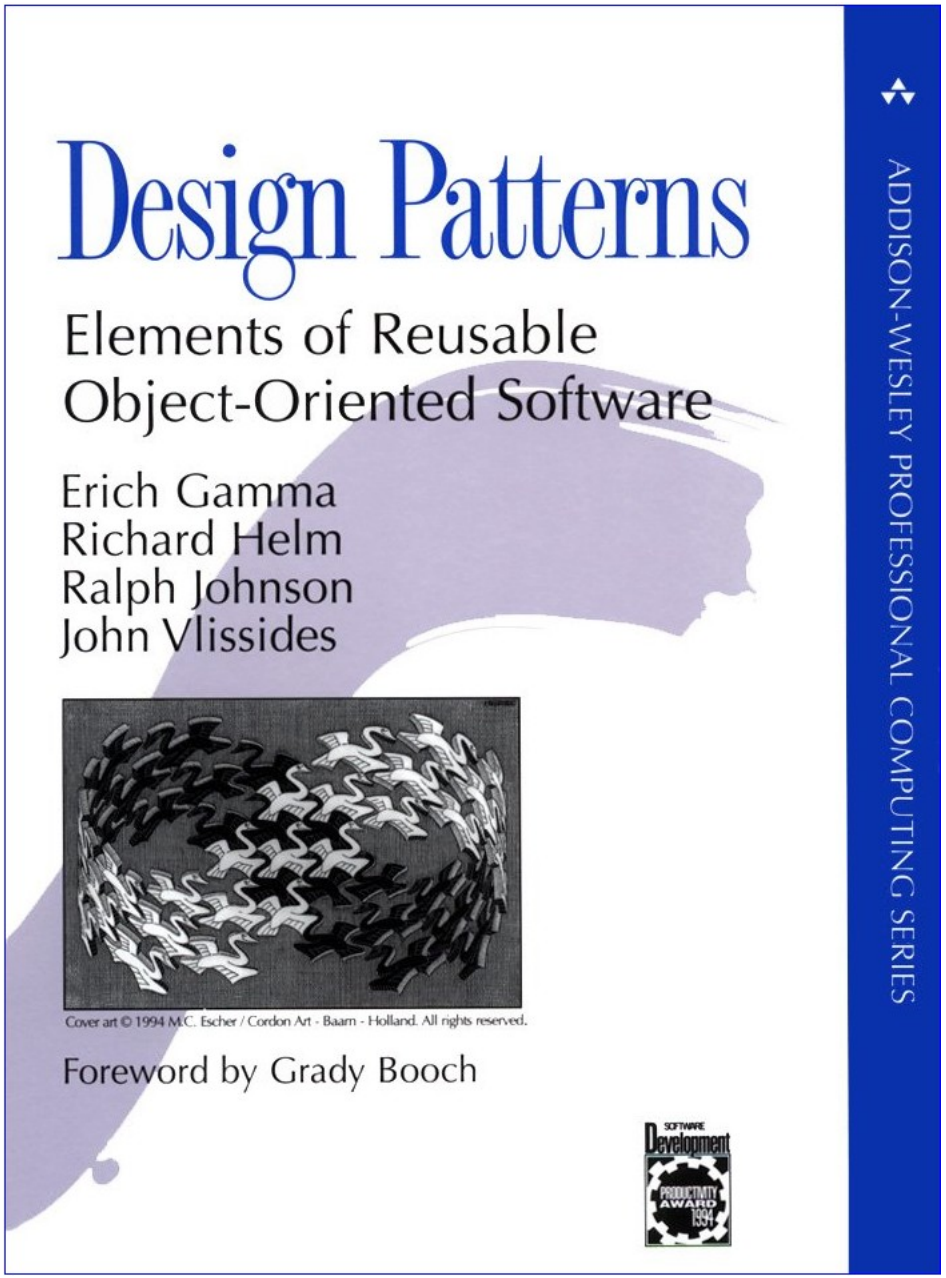
# Návrhové vzory

# Literatura

- **E. Gamma, R. Helm, R. Johnson, J. Vlissides**
  The Gang of Four (GoF)
  **Design Patterns**
  Elements of Reusable Object-Oriented Software
  1995
  *Grada 2003:*
  *Návrh programů podle vzorů*

- J. Vlissides:
  Pattern Hatching - Design Patterns Applied





# Literatura

- Design Patterns in C#, Java, VisualBasic, PHP, ..., Learning Design Patterns, ...
- Freeman, Freeman, Sierra, Bates: Head First Design Patterns

- A. Alexandrescu:
  Modern C++ Design - Generic Programming and Design Patterns Applied
  (C++ in Depth)
  *Moderní programování v C++*

- Brown, Malveau, McCormick, Mowbray
  AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis

- http://hillside.net
- google: design patterns
- http://www.mcdonaldland.info/2007/11/

# 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
- **'Revoční' 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

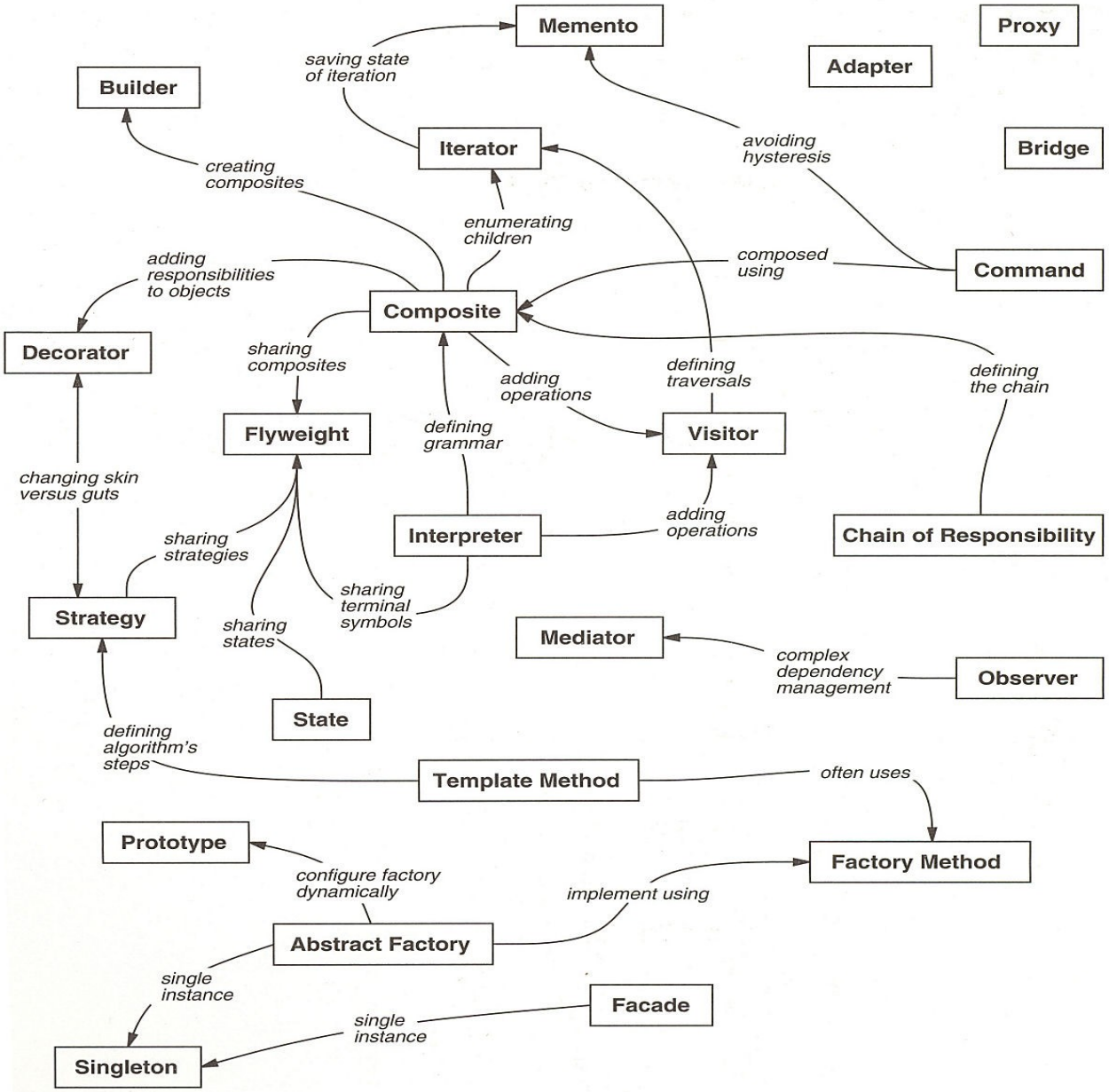
| | **Creational**<br>*Tvořivé vzory* | **Structural**<br>*Strukturální vzory* | **Behavioral**<br>*Vzory chování* |
|---|---|---|---|
| **Třída** | **Factory Method** | **Adapter** | **Interpreter**<br>**Template Method** |
| **Objekt** | **Abstract Factory**<br>**Builder**<br>**Prototype**<br>**Singleton**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

vytváření objektů

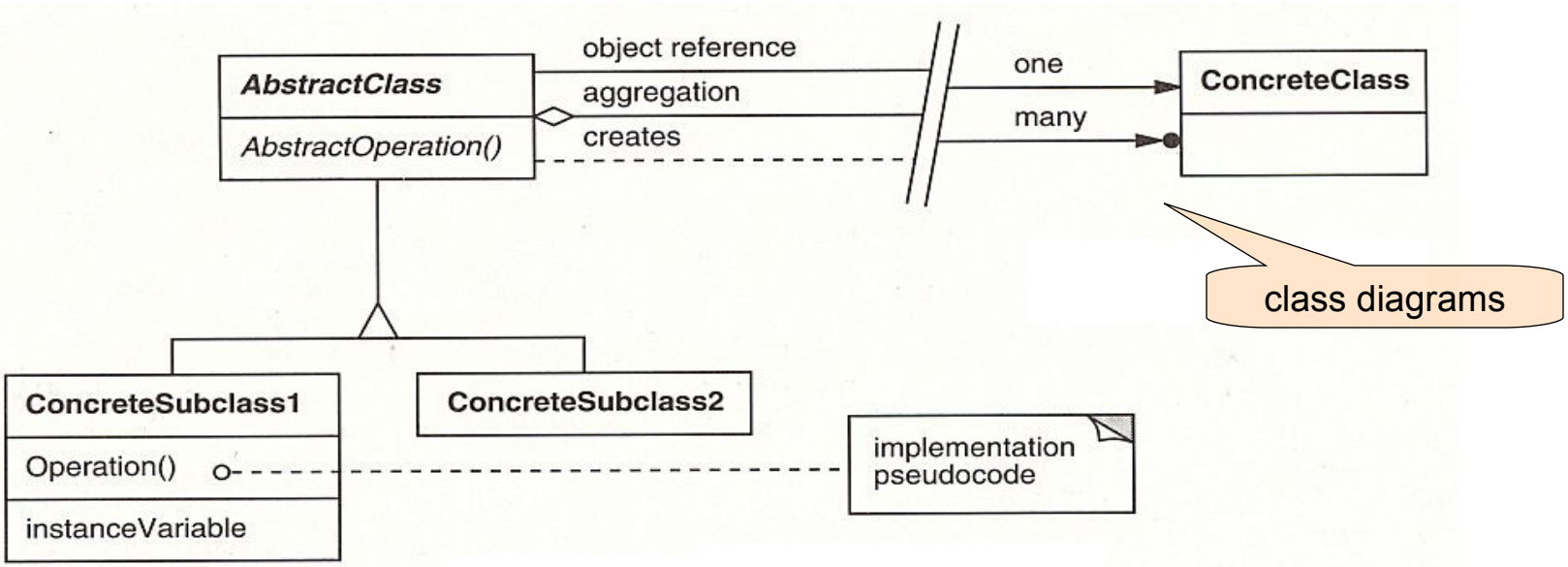
uspořádání tříd a objektů

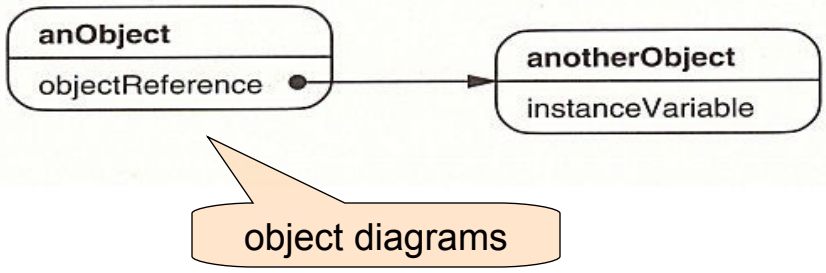
chování a interakce objektů a tříd

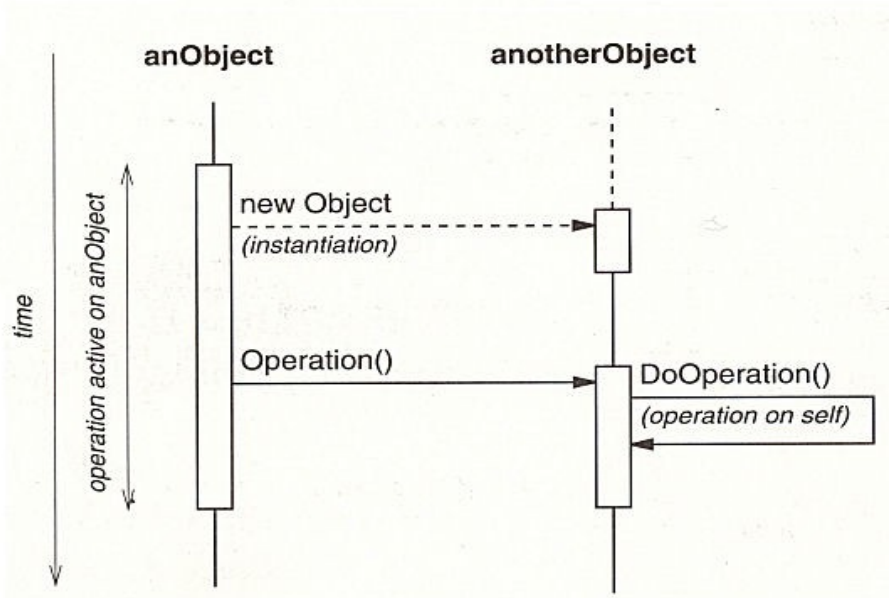
# Vztahy mezi NV



# Značení – Object Modeling Technique



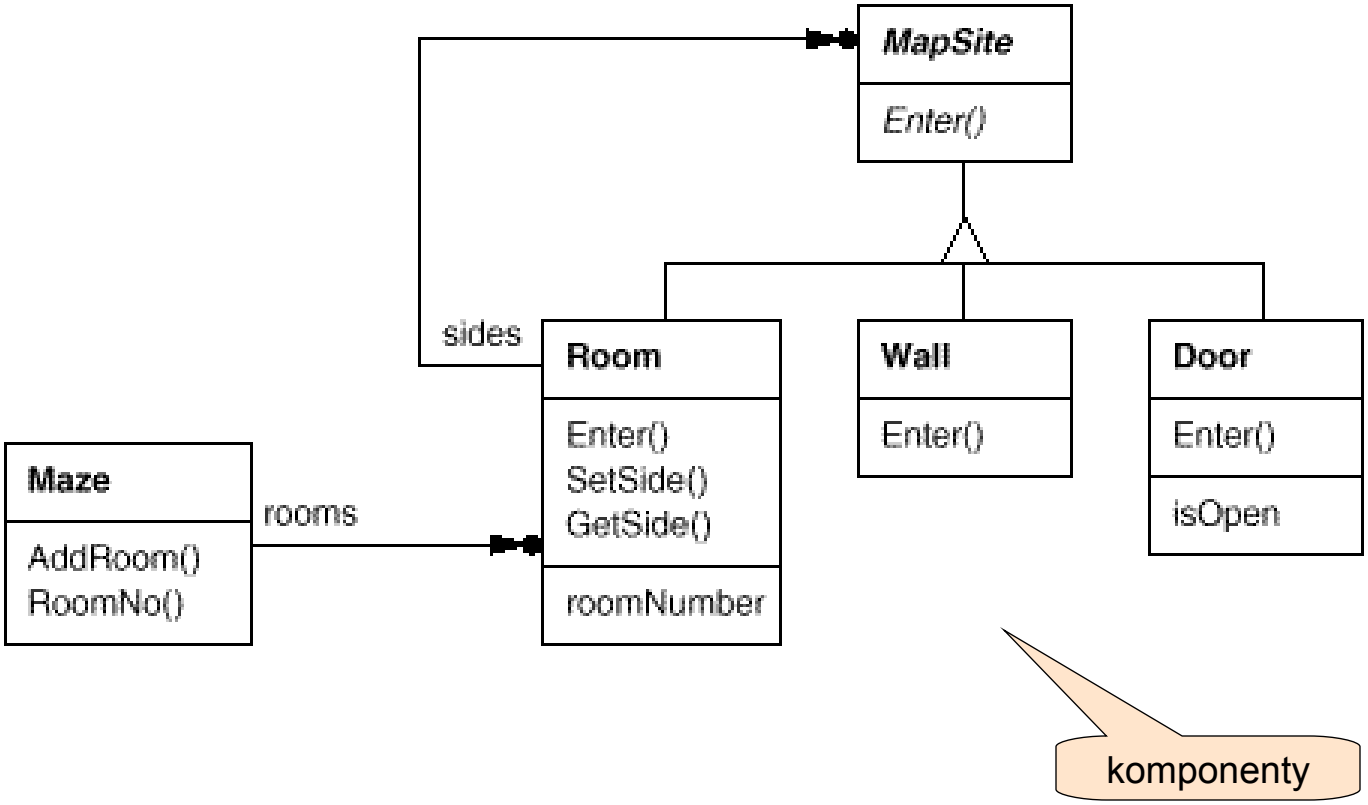



# Tvořivé NV

- **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

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

class MapSite {
public:
  virtual void Enter() = 0;
};

class Room : public MapSite {
public:
  Room(int roomNo);

  MapSite* GetSide(Direction) const;
  void SetSide(Direction, MapSite*);

  virtual void Enter();

private:
  MapSite* _sides[4];
  int _roomNumber;
};
```

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

```
class Wall : public MapSite {
public:
  Wall();
  virtual void Enter();
};

class Door : public MapSite {
public:
  Door(Room* = 0, Room* = 0);

  virtual void Enter();
  Room* OtherSideFrom(Room*);

private:
  Room* _room1;
  Room* _room2;
  bool _isOpen;
};

class Maze {
public:
  Maze();

  void AddRoom(Room*);
  Room* RoomNo(int) const;
private: // ...
};
```

# Bludiště - vytvoření

```
Maze* MazeGame::CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor = new Door(r1, r2);

  aMaze->AddRoom(r1);
  aMaze->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 aMaze;
}
```

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 instanciovaných tříd předáním jiného parametru
- **Builder**
  - `CreateMaze` dostane parametr objekt s operacemi pro přidávání komponent
  - pomocí dědičnost lze lzměnit 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/ConcreteCreartor - 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* aMaze = MakeMaze();
  Room* r1 = MakeRoom(1);
  Room* r2 = MakeRoom(2);
  Door* theDoor = MakeDoor(r1, r2);
```

```
Maze* CreateMaze( MazeFactory *f) {
  Maze* aMaze = f->MakeMaze();
  Room* r1 = f->MakeRoom(1);
  Room* r2 = f->MakeRoom(2);
  Door* door = f->MakeDoor(r1,r2);
```

Factory Method

Abstract Factory

```
Maze* CreateMaze( MazeBuilder& builder) {
  builder.BuildMaze();
  builder.BuildRoom(1);
  builder.BuildRoom(2);
  builder.BuildDoor(1, 2);
  return builder.GetMaze();
```

Builder

```
MazeProtoFactory simpleMazeFactory(
  new Maze, new Wall, new Room, new Door);
Maze* aMaze = game.CreateMaze( simpleMazeFactory);
```

```
Wall* MazeProtoFactory::MakeWall () const {
  return _protoWall->Clone();
}

Door* MazeProtoFactory::MakeDoor (Room* r1, Room *r2) {
  Door* door = _protoDoor->Clone();
  door->Initialize(r1, r2);
  return door;
}
```

Prototype

# Strukturální NV

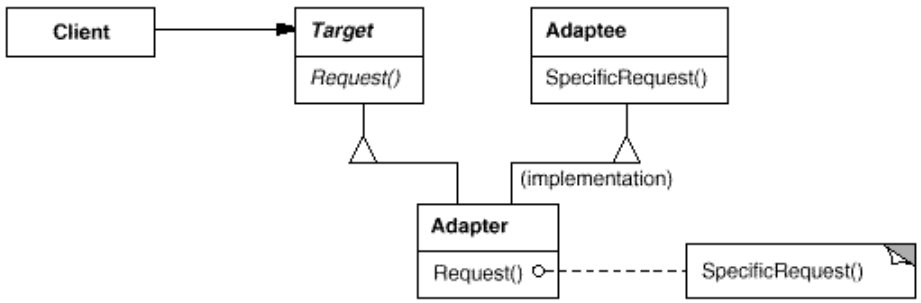
- **Structural Patterns**
  - jak jsou třídy a objekty složeny do větších struktur
- **Strukturální NV tříd**
  - dědičnost pro skládání rozhraní nebo implementací
  - **Adapter** - přizpůsobení rozhraní třídy jiným rozhraním
- **Strukturální NV objektů**
  - skládání objektů pro dosažení nové funkcionality
  - 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

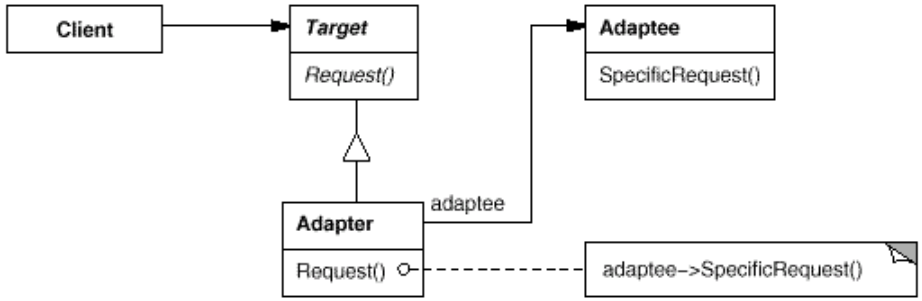
**Na samostatných slajdech**

- **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

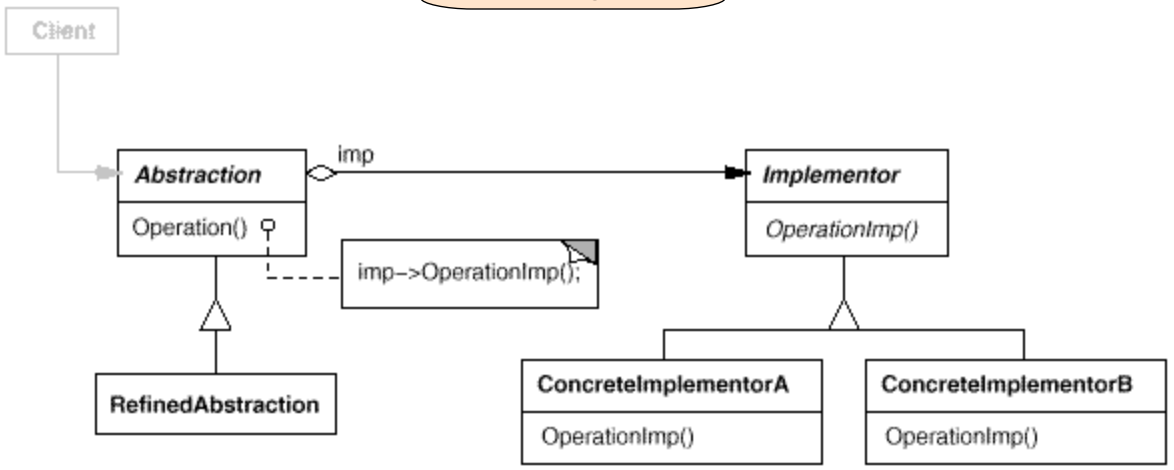



Class Adapter




Object Adapter

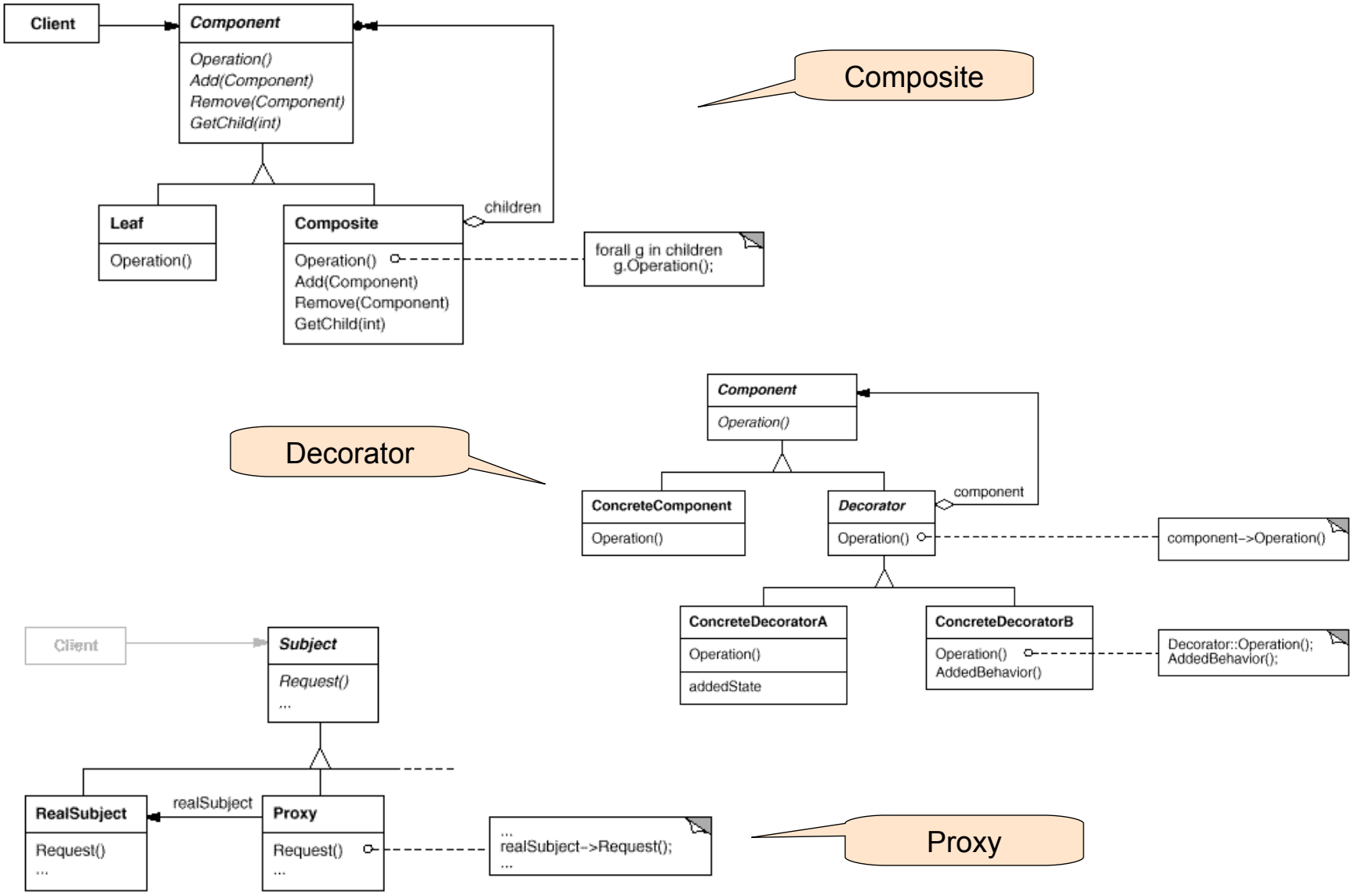
Bridge



# 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

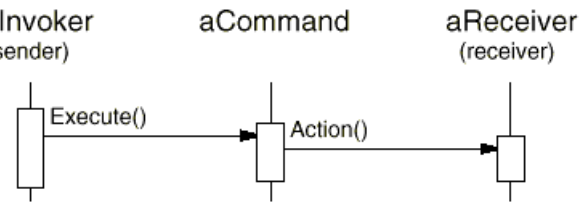
- **Behavioral design patterns**
  - rozdělení funkčnosti a zodpovědnosti mezi objekty
  - komunikace mezi objekty
  - složitější struktura provádění kódu
  - umožňuje zaměřit se při návrhu na propojení tříd, ne na běhové technické detaily
  - dynamické vztahy - RT vlastnosti
  - vzájemná provázanost

- **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

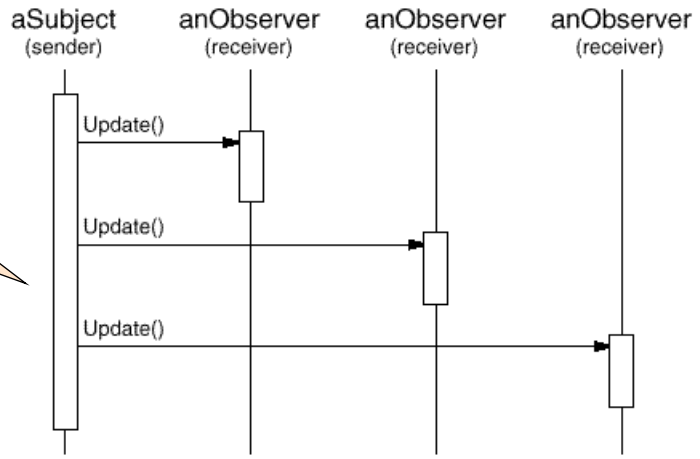
- **Behavioral object patterns**
  - spolupráce mezi skupinami objektů pro dosažení funkčnosti
- **Objektová kompozice místo dědičnosti**
  - **Mediator**
    - odstraňuje nutnost referencí na všechny spolupracující objekty
  - **Chain of Responsibility**
    - zasílání zpráv neznámým objektům přes zřetězené 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 agragovaných objektů

# Vztahy mezi odesílateli a příjemci zpráv

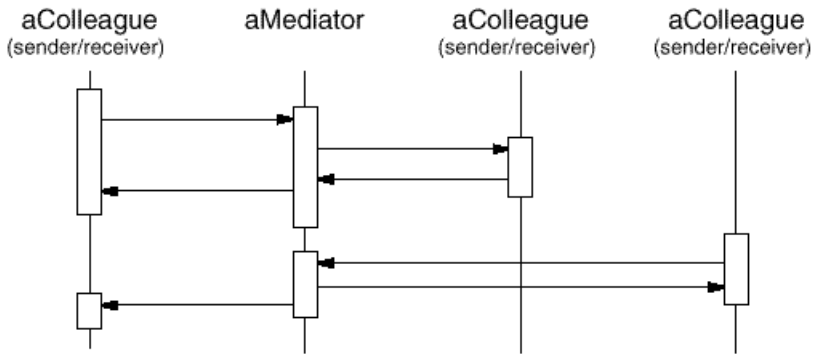



Comand
Separace příjemce (a parametrů) od okamžiku volání

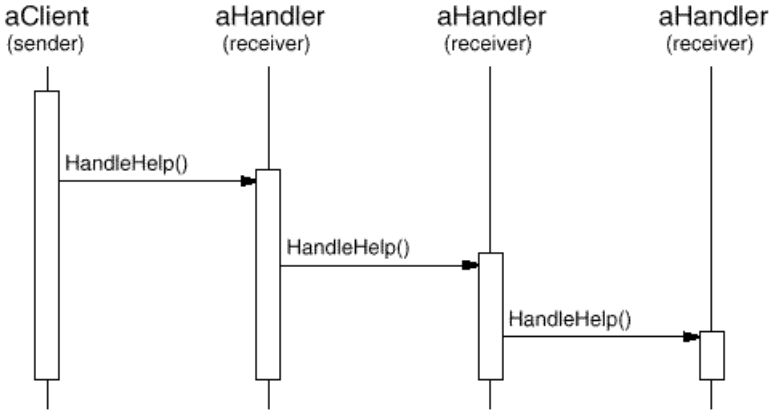
Observer
neznámý počet příjemců




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 vymýšlení kola
    - ... a *'bez chyb'*
  - Kompozice NV
  - Generické implementace
  - Mnoho dalších rozšiřujících vzorů
    - *často cíleně zaměřených*