PROGRAMOVÁNÍ BEZ CHYB

A ČITELNĚ

KAPITOLA I

REVIZE K'DU

REVIZE K'DU

Je snadné udělat chybu

```
Control c = getControl();
if (c == null && c.isDisposed())
    return;
```

Je nesnadné ji odhalit

MOTIVACE

- **Inspekce je levnější než testování**
- **Čtení kódu odhalí více chyb než vlastní test**

(Basili and Selby 1987).

- **6x vyšší fin nároky na testy v porovnání s inspekcí k'du**

(Ackerman, Buchwald, and Lewski 1989).

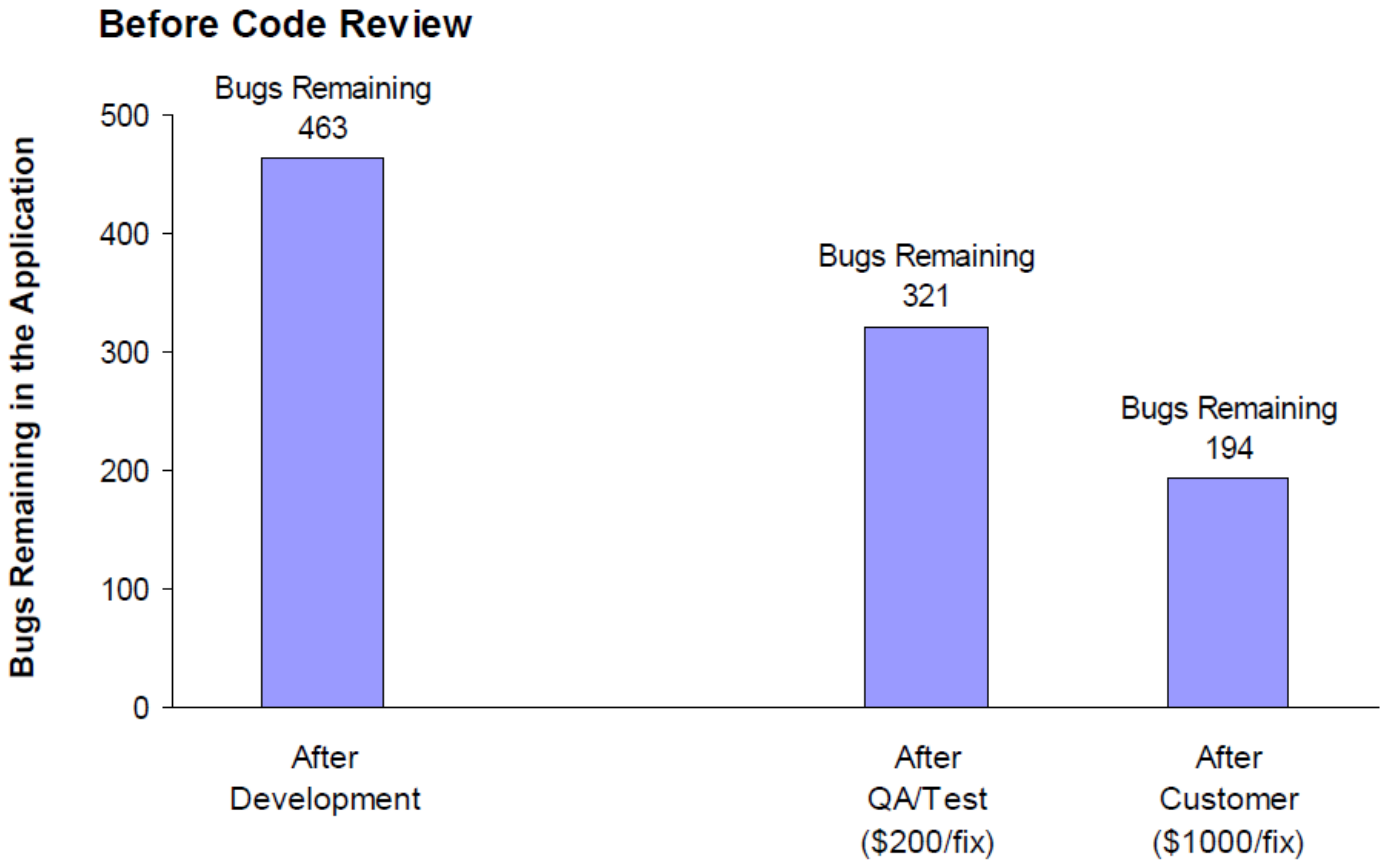
- **IBM studie :** nalezení jedné chyby
  - ~ 3.5 hod při inspekci k'du
  - ~ 15-25 hod při testování

(Kaplan 1995)

MOTIVACE

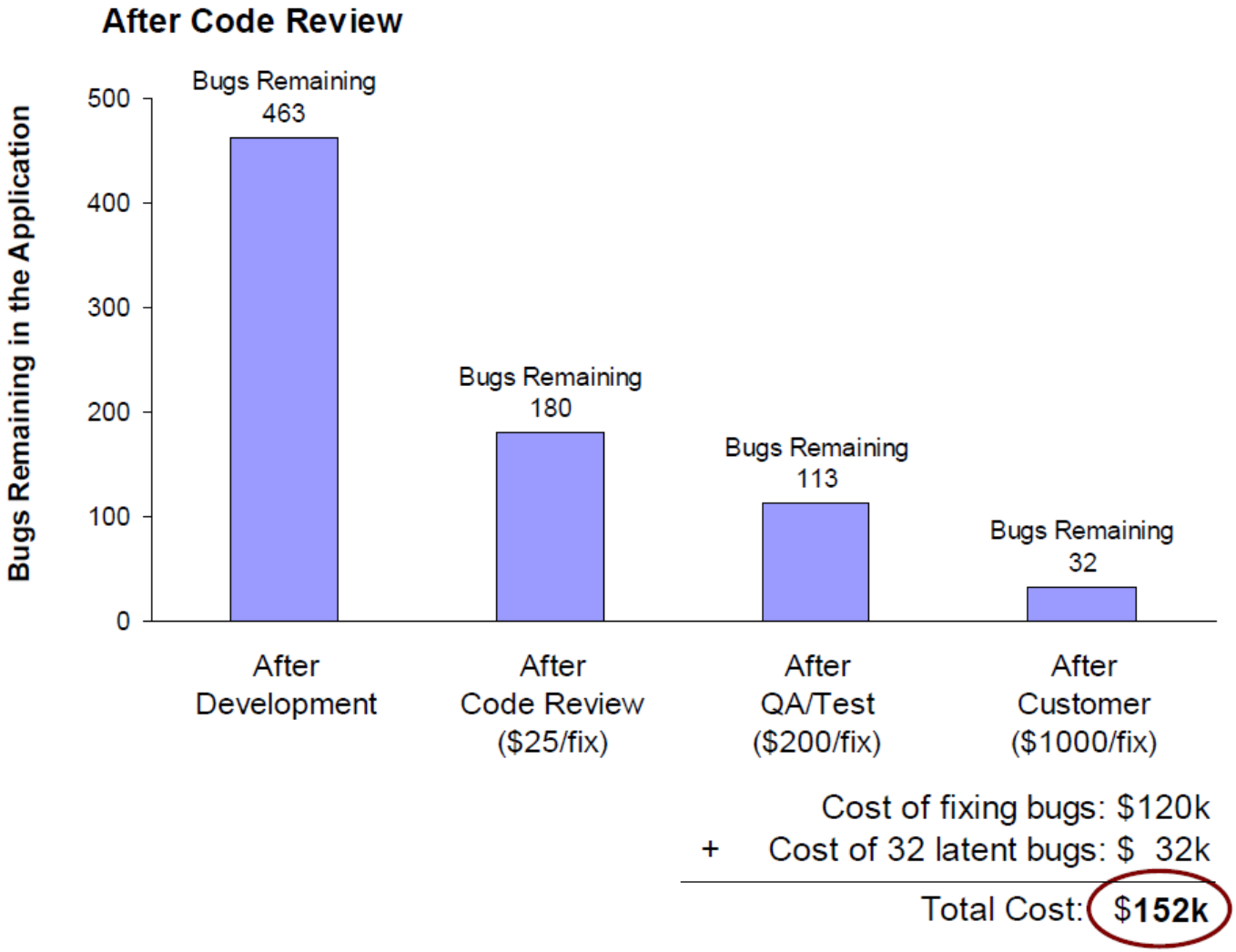
- **Kombinace návrhu a inspekce k'du odstraní 70-85% chyb (Jones 1996)**
- **Inspekce zvýší produktivitu o 20% (Wiegers 2002)**
- **Inspekce sníží 10-15% nákladů na projekt**

STUDIE



Cost of fixing bugs: $174k
\+ Cost of 194 latent bugs: $194k
Total Cost: **$368k**

STUDIE


After Code Review
Bugs Remaining in the Application
500
400
300
200
100
0
Bugs Remaining
463
Bugs Remaining
180
Bugs Remaining
113
Bugs Remaining
32
After Development
After Code Review ($25/fix)
After QA/Test ($200/fix)
After Customer ($1000/fix)
Cost of fixing bugs: $120k
+ Cost of 32 latent bugs: $ 32k
Total Cost: $152k

REVIZE K'DU

Revize jinou osobou/nástrojem

- **Crucible**
- **CodeCollaborator**
- **Review Board™**

Časové náročné

Otravné

Nudné

Občas i motivuje k vraždě..

REVIZE K'DU

Revize jinou osobou/nástrojem

- **Crucible**
- **CodeCollaborator**
- **Review Board™**

Časové náročné

Otravné

Nudné

Občas i motivuje k vraždě..

STATICKÁ ANALÝZA ZDROJOVÉHO K'DU

Revize nástrojem

- **FindBugs**
- **PMD**
- **CheckStyle**
- **Jdepend**
- **Ckjm**
- **Cpd**
- **IntelliJ**

Okamžitá odpověď

FINDBUGS

Automatická statická analýza

FindBugs

because its easy

an open source project

http://findbugs.sourceforge.net/

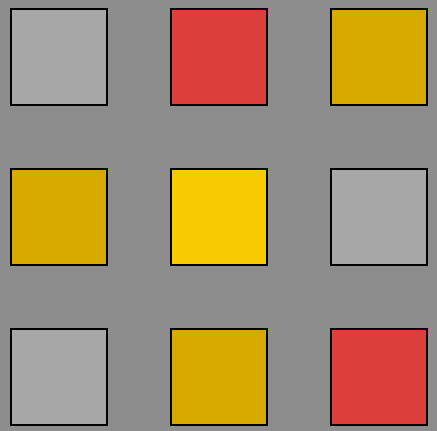
FINDBUGS

Automatická statická analýza

INPUT

- Set of ".class" files containing byte-code
- Configurations

PROCESSING



- Detectors

OUTPUT

- Bug Pattern Code
- Sou... Line Nu...
- De... Me...

FINDBUGS

- Not concerned by formatting or coding standards
- Focus on detecting potential bugs and performance issues
- Can detect many types of common, hard-to-find bugs

NullPointerException

```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
 ...
}
```

Uninitialized field

```
public class ShoppingCart {
    private List items;
    public addItem(Item item) {
        items.add(item);
   }
}
```

FINDBUGS

Over 200 rules divided into different categories:

- *Correctness*

  E.g. infinite recursive loop, reads a field that is never written
- *Bad practice*

  E.g. code that drops exceptions or fails to close file
- *Performance*
- *Multithreaded correctness*
- *Dodgy*
  - E.g. unused local variables or unchecked casts
- *Malicious code vulnerability*
- *Security*
- *..*

FINDBUGS

Demo

KAPITOLA II

HLADÍTKA

MONITOROVÁNÍ JVM

1. Monitorování JVM
2. Profilování
3. Ladění

- Vývojáři, architekti, admini

- Paměť, CPU, file handly, čas, škálování, odezva, propostnost

MONITOROVÁNÍ JVM

Nástroje

- **JConsole**
- **VisualVM**

- **Profilování**
  - Vysoké CPU, okupovaná paměť, mem-leak, resource leak, lock

MONITOROVÁNÍ JVM

Demo

- **JConsole**
- **JVisialVM**

MONITOROVÁNÍ

- **jps**: list the instrumented HotSpot VM's for the current user in the target system.
- **jinfo**: prints system properties and command-line flags used in a JVM
- **jmap**: prints memory related statistics for a running VM or core file
- **jhat**: parses a heap dump in binary format and starts an HTTP server to browse different queries (JDK 6)
- **jstat**: provides information on performance and resource consumption of running applications using the built-in instrumentation in the HotSpot VM.
- **jstack**: prints the stack traces of all the threads attached to the JVM, including Java threads and VM internal threads. Also performs deadlock detection
- **jconsole**: uses the build-in JMX instrumentation in the JVM to provide information on performance and resource consumption of running applications
- **jvisualvm**: useful to troubleshoot applications and to monitor and improve application's performance. It helps ot generate and analyze heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweigth memory and CPU profiling.

MONITOROVÁNÍ

- **Unix tools**
  - **top –** CPU Mem usage
  - **sysstat –** system performance monitoring
  - **du –** disk usage
  - **lsof | wc –l – open files**
  - **lsof –a –p <pid> | wc –l**

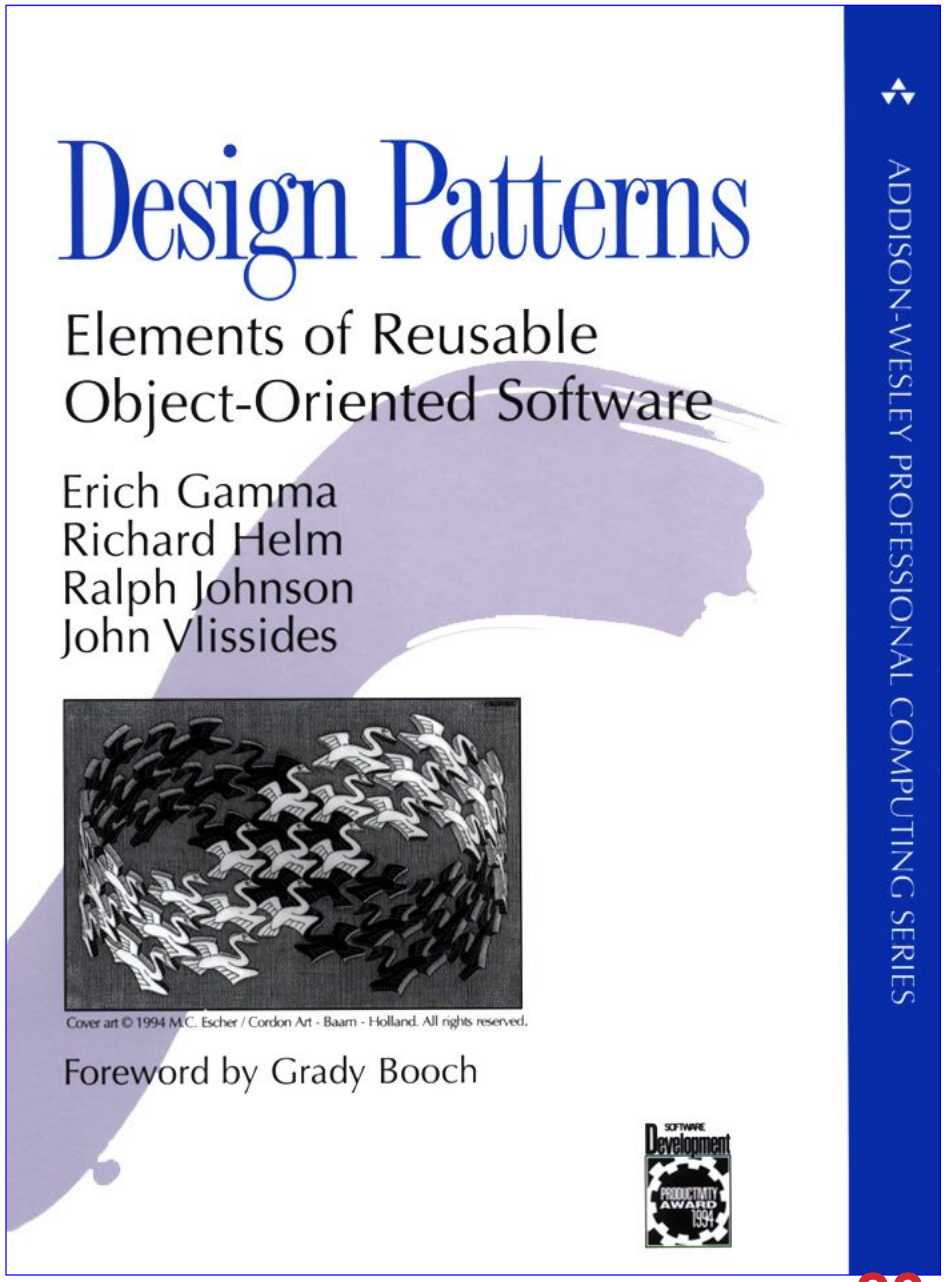
  - **Jmeter -** performance

KAPITOLA III

NÁVRHOVÉ VZORY

- E. Gamma, R. Helm, R. Johnson, J. Vlissides
- **The Gang of Four (GoF)**

Design Patterns
Elements of Reusable Object-Oriented Software (1994)



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

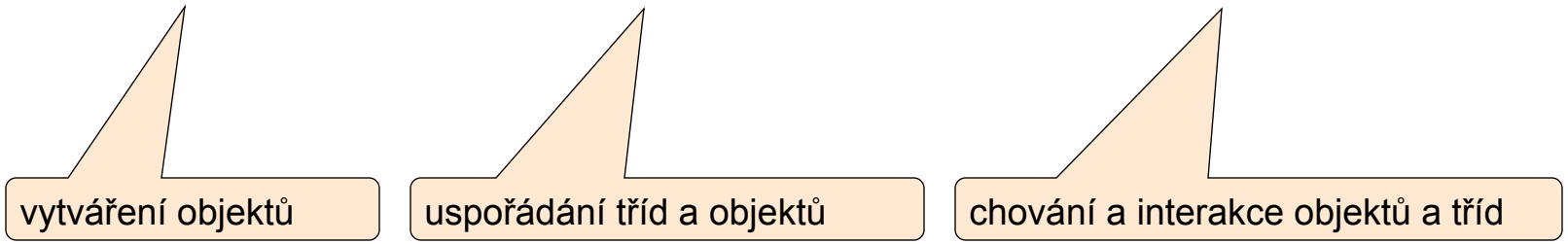
- **Relativní komplexnost a obecnost**
  - pro rozsáhlejší systémy
    - předpoklad dlouhé životnosti, údržby a rozšiřování
  - při návrhu nových systémů
  - při rozsáhlých úpravách
- **Inženýrský přístup**
  - přehled o existenci a typickém použití
  - při návrhu hledat uplatnění
- **'Revoční' myšlenka 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 *Tvořivé vzory*	**Structural** *Strukturální vzory*	**Behavioral** *Vzory chování*
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**



Tvořivé Návrhové Vzory

- **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

Strukturální Návrhové Vzory

- **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

Návrhové Vzory Chování

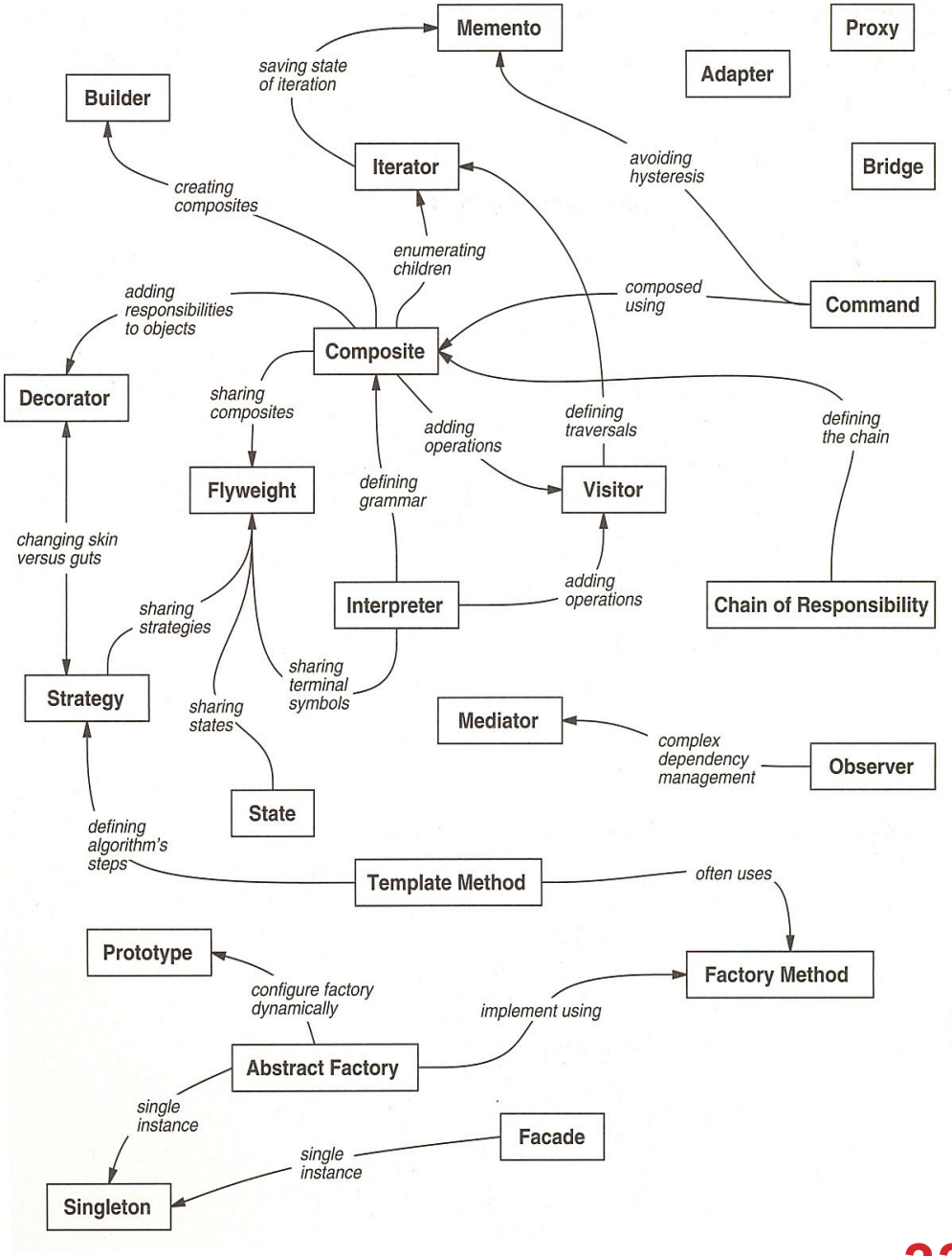
- **Behavioral design patterns**
  - rozdělení funkčnosti a zodpovědnosti mezi objekty
  - komunikace mezi objekty
  - složitější struktura provádění kódu
  - umožňuje zaměřit se při návrhu na propojení tříd, ne na běhové technické detaily
  - dynamické vztahy - RT vlastnosti
  - vzájemná provázanost

- **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

Návrhové Vzory 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

- **Shrnutí**
  - 'Žádné velké moudro'
    - ... jak pro koho
  - Slovník!
  - Implementace bez vymýšlení kola
    - ... a *'bez chyb'*
  - Mnoho dalších rozšiřujících vzorů
    - *často cíleně zaměřených*



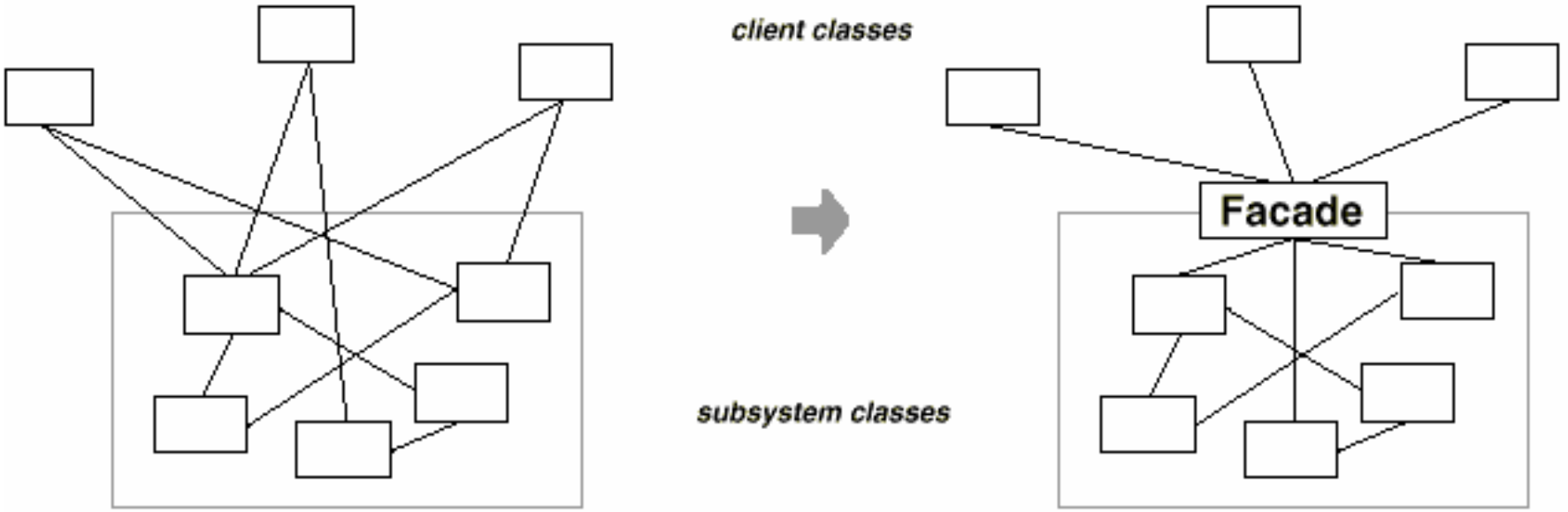
Facade

Facade

- **Účel**
  - sjednocené high-level rozhraní pro subsystém
  - zjednodušuje použití subsystému
  - zapouzdření – skrytí návrhu před uživateli

- **Motivace**
  - zjednodušit komunikaci mezi subsystémy
  - zredukovat závislosti mezi subsystémy
  - neznemožnit používání low-level interfaců
    - pro použití „na míru“

Facade - motivace



Facade – motivace

Puštění filmu:

- zatáhnutí žaluzií
- zapnutí projektoru
- zapnutí DVD přehrávače
- zapnutí ozvučení

Ukončení přehrávání:

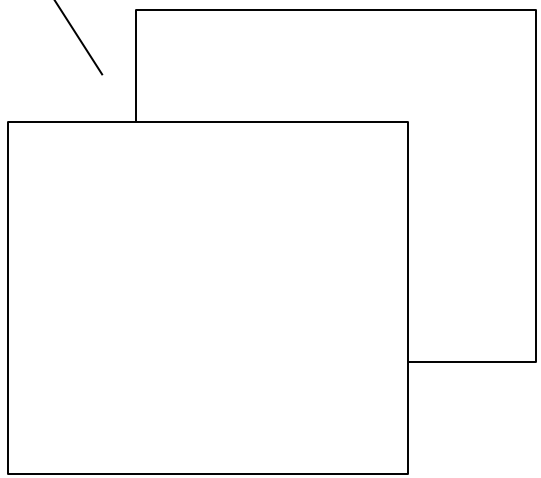
- vypnout DVD přehrávač
- vypnout ozvučení
- vypnout projektor
- vytáhnout žaluzie

Facade – motivace

Řešení:

Univerzální ovladač s funkcemi:

- Přehrát film
- Ukončit film
- Přehrát hudbu
- Vypnout hudbu
- …

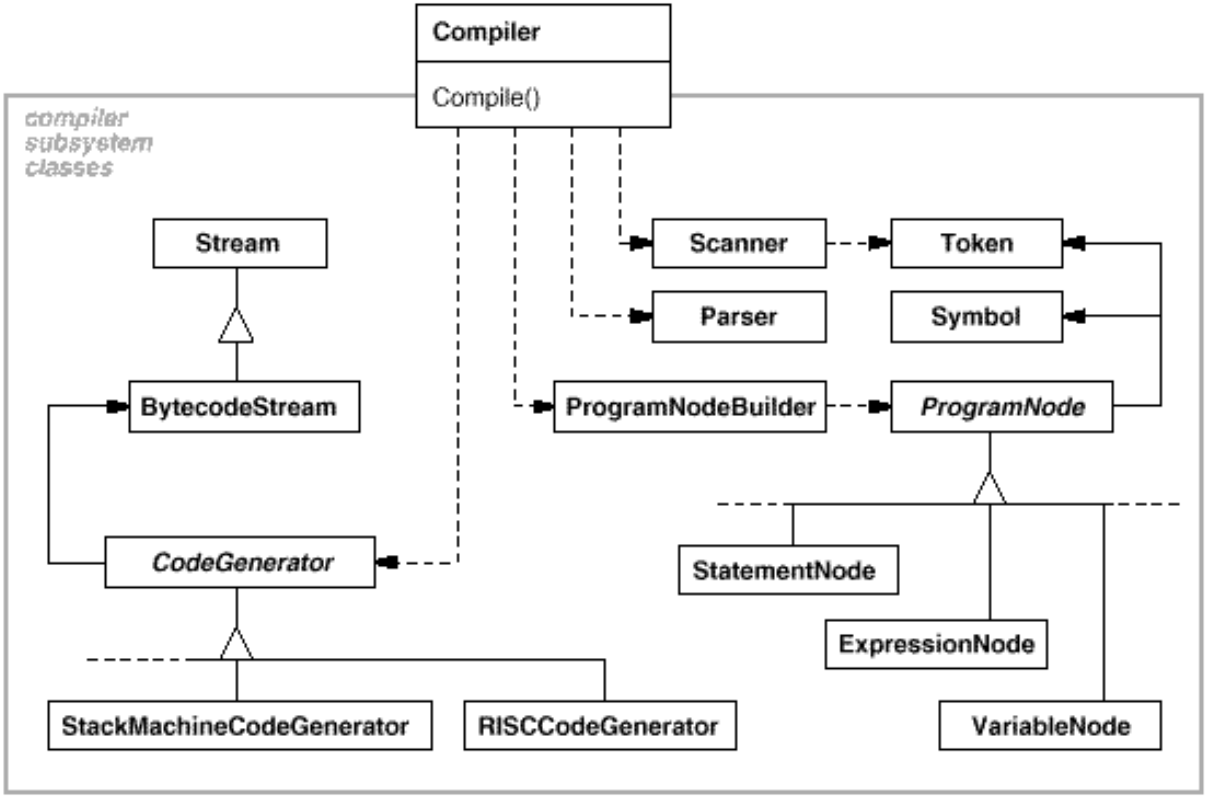


- zatáhnout žaluzie
- zapnout projektor
- zapnout DVD přehrávač
- zapnout ozvučení

- vypnout DVD přehrávač
- vypnout ozvučení
- vypnout projektor
- vytáhnout žaluzie

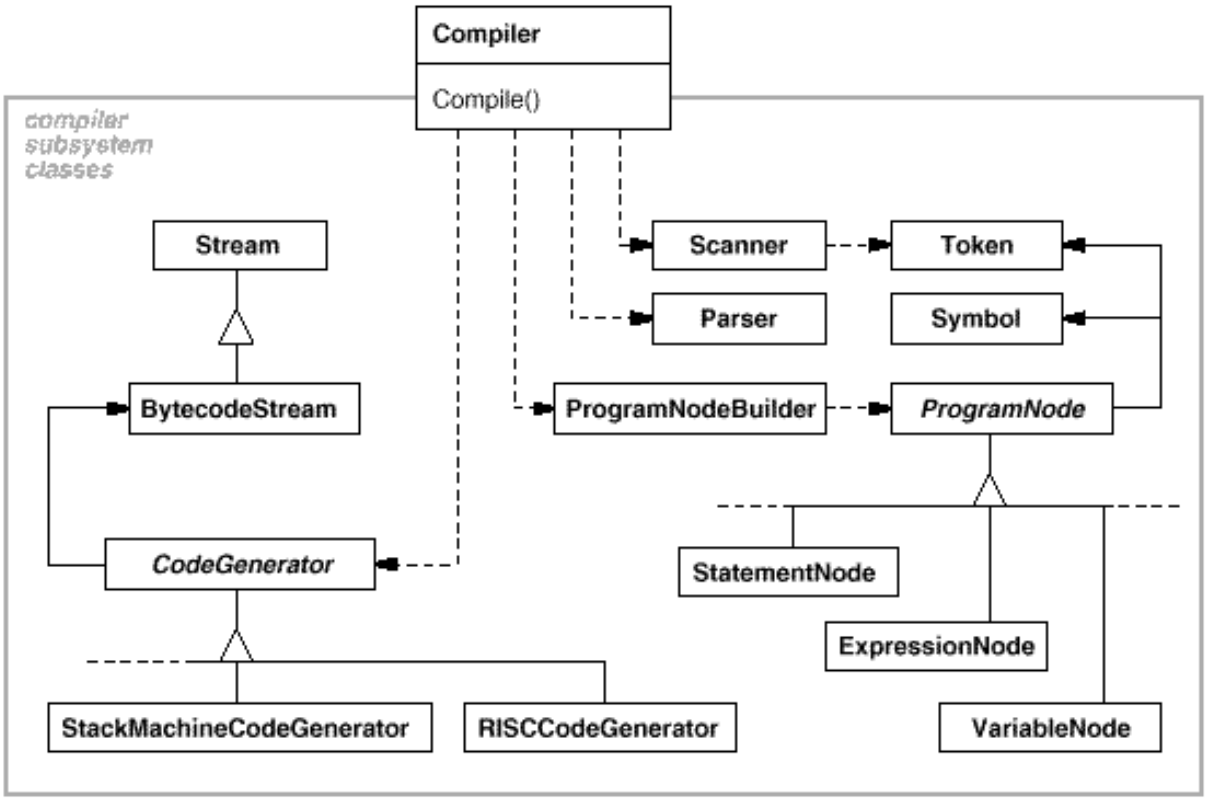
Facade – motivace

- **Příklad**
  - *Compiler*
  - třídy *Scanner*, *Parser*, *ProgramNodeBuilder*, ..
  - Facade poskytuje rozhraní pro práci s kompilátorem na high-level úrovni



Facade – motivace

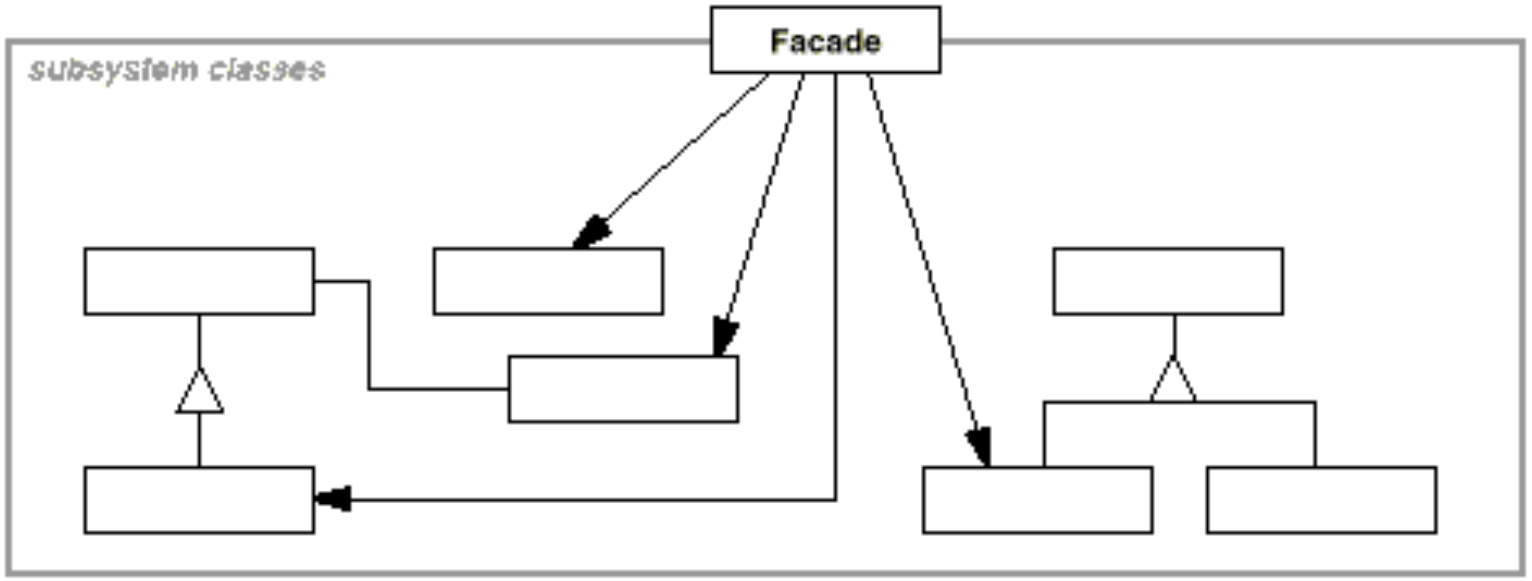
- **Příklad**
  - *transparentnost* – třídy subsystému o Facade nevědí
  - *svoboda volby*- neskrývá třídy subsystému (Parser, Scanner)



Facade - použití

- **Použití**
  - když je subsystém složitý na běžné použití
  - když existuje mnoho závislostí vně subsystému
  - při vytváření vstupních bodů vrstveného systému

- **Struktura**



Facade - použití

- **Účastníci**
  - Facade (kompilátor)
    - zná třídy uvnitř subsystému
    - deleguje požadavky
  - třídy subsystému (Scanner, Parser,...)
    - nevědí o existenci Facade
    - implementují funkčnost

- **Souvislosti**
  - klient se subsystémem komunikuje přes Facade
    - Facade předává požadavky dál
    - může obsahovat vlastní logiku – překlad požadavků na třídy subsystému
  - klienti nemusí používat podsystém přímo

Facade - důsledky

- **Výhody použití**
  - redukuje počet objektů, se kterými klienti komunikují
    - snadnější použití subsystému
  - zmenšuje počet závislostí mezi klienty a subsystémem
    - odstraňuje komplexní a kruhové závislosti
    - méně kompilačních závislostí
  - neskrývá třídy podsytému
    - klient si může vybrat jednoduchost nebo použití na „míru“
  - umožňuje rozšířit stávající funkcionalitu
    - kombinací podsystémů a jejich metod
    - monitorování systému – přístupy, využívání jednotlivých metod

- **Kdy se vyplatí facade do systému implementovat**
  - má cenu o ní uvažovat pouze v případě, kdy je cena za vytvoření fasády menší, než je nastudování systému (podsystémů) uživateli

Facade - příklad

```
class Scanner {
public:
        …
        virtual Token& Scan();
        …
};
```

Třída subsystému

```
class Parser {
public:
  …
  virtual void Parse(Scanner&,ProgramNodeBuilder&);
  …
};
```

Třída subsystému

```
class ProgramNodeBuilder {
public:
  ProgramNodeBuilder();
  virtual ProgramNode* NewVariable(…);
  virtual ProgramNode* NewAssignment(…);
  virtual ProgramNode* NewReturnStatement(…);
  virtual ProgramNode* NewCondition(…) const;
  …
  ProgramNode* GetRootNode();
  …
};
```

Třída subsystému

Facade - příklad

Třída subsystému

```
class ProgramNode {
public:
        // program node manipulation
        virtual void GetSourcePosition(int& line, int& index);
  …

        // child manipulation
        virtual void Add(ProgramNode*);
        virtual void Remove(ProgramNode*);
  …

        virtual void Traverse(CodeGenerator&);
protected: ProgramNode();
};
```

Třída subsystému

```
class CodeGenerator {
public:
  virtual void Visit(StatementNode*);
  virtual void Visit(ExpressionNode*);
  …
protected:
  CodeGenerator(BytecodeStream&);
protected:
  BytecodeStream& _output;
};
```

Facade - příklad

```
void ExpressionNode::Traverse (CodeGenerator& cg)
{
        cg.Visit(this);
        ListIterator i(_children);
        for (i.First(); !i.IsDone(); i.Next()) {
                i.CurrentItem()->Traverse(cg);
        }
}
```

Facade

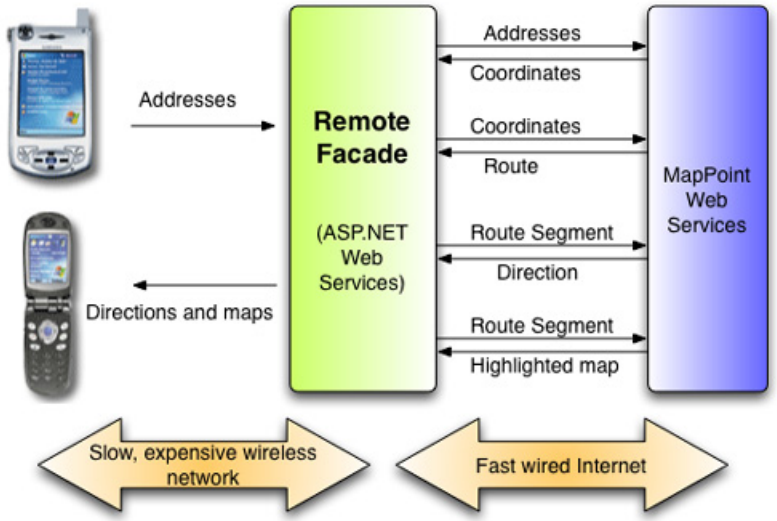
```
class Compiler {
public:
  Compiler();
  virtual void Compile(istream&,BytecodeStream&);
};

void Compiler::Compile ( istream& input, BytecodeStream& output )
{
  Scanner scanner(input);
  ProgramNodeBuilder builder;
  Parser parser;
  parser.Parse(scanner, builder);
  RISCCodeGenerator generator(output);                // potomek CodeGenerator
  ProgramNode* parseTree = builder.GetRootNode();
  parseTree->Traverse(generator);
}
```

Facade - implementace

- **Konfigurace fasády**
  - Facade jako abstraktní třída
    - konkrétní implementace podsystému je jejím potomkem
    - klienti komunikují s podsystémem přes rozhraní abstraktní třídy
      - klient neví, která implementace podsystému je použita
      - lze zcela změnit způsob implementace, flexibilita podsystému
  - Facade jako jedna konfigurovatelná třída
    - slabší alternativa předchozího
    - výměna komponent podsystému
- **Viditelnost komponent podsystému**
  - je vhodné určit viditelné a skryté komponenty podsystému
    - analogie private a public metod třídy
    - malá podpora v objektových jazycích

Facade – reálné použití

- **V mobilních aplikacích**

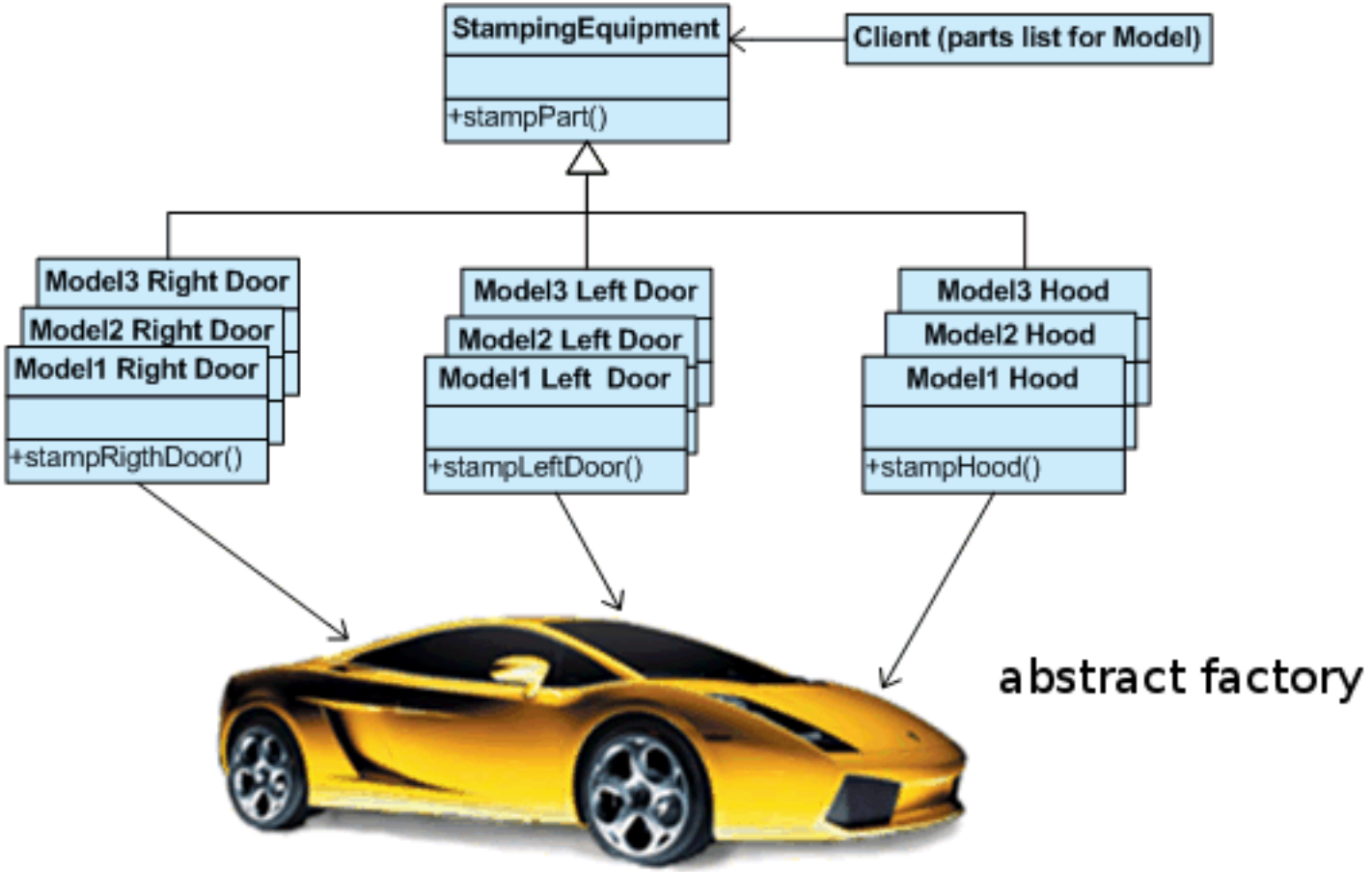


- **Session Facade v J2EE**
  - Fasáda pro webové služby
- **JOptionPane ve Swingu**
  - vytváří různé typy základních dialogových oken a zobrazuje je
  - zjednodušuje používání této rozsáhlé knihovny

Facade – související vzory

- **Abstract Factory**
  - Facade může poskytovat interface pro tvorbu objektů
- **Singleton**
  - Facade jako Singleton - jen jeden vstupní bod do systému
- **Mediator**
  - Mediator také snižuje závislosti
  - na rozdíl od fasády snižuje závislosti mezi komponentami subsystému

FACTORY METHOD



ÚVOD - PROBLÉM

- Mějme obchod s auty:

Při přidání nového modelu je nutné upravit

```
public class OrderCars {
    public Car orderCar(String model) {
        Car car = null;

        if(model.equals("Mark IV"))       car = new Mercury(model);
        else if(model.equals("Corvette")) car = new Chevrolet(model);
        else if(model.equals("Fusion"))   car = new Ford(model);
        else if(model.equals("Enzo"))     car = new Ferrari(model);
        else if(model.equals("Fabia"))    car = new Skoda(model);

        car.buildCar();
        car.testCar();
        car.shipCar();
    }
}
```

Kód, který se nebude často měnit

- **Problém – míchání relativně stálého kódu s kódem, který se bude měnit při každém přidání nové třídy**
- **Problém – míchání úrovní abstrakce – auto X Fabia**

ŘEŠENÍ PŘEDCHOZÍHO PROBLÉMU

```
public abstract class CarFactory {
    abstract Car createCar(String model);

    public Car orderCar(String model) {
        Car car = createCar(model);
        car.buildCar();
        car.testCar();
        car.shipCar();
        return car;
    }
}
```

Vytvoření objektu – využívá polymorfismu

Business logika

```
public class FordFactory extends CarFactory {
    Car createCar(String model) {
        if(model.equals("Fusion")) return new Ford(model);
        else if(model.equals("Mark IV")) return new Mercury(model);
        else return null;
    }
}
```

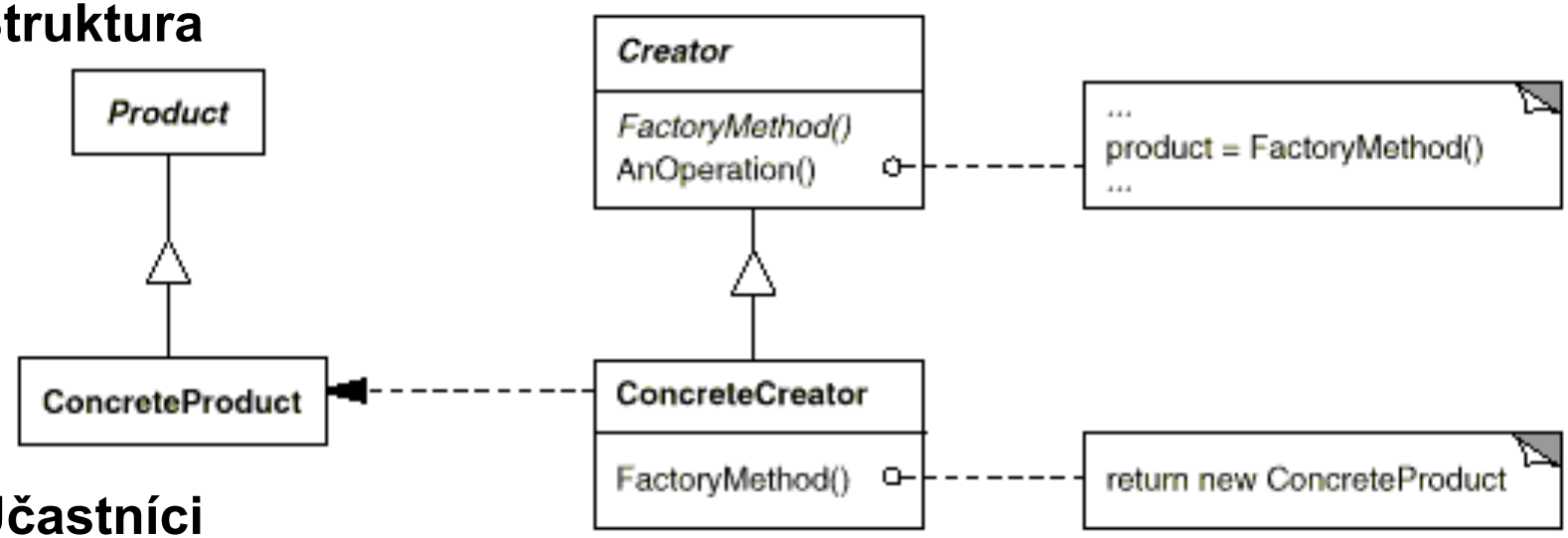
- Vytvoření „frameworku“ pro objednávání aut
- Při přidání nového auta => pouze změna `FordFactory.createCar`

FACTORY METHOD

- **Známý jako**
  - Tovární metoda
  - Virtual Constructor
- **Účel**
  - Definuje rozhraní pro vytvoření objektu
  - Potomci sami rozhodují kterou třídu instanciovat
  - Umožňuje odložit instanciaci z předka na potomka
- **Kategorie**
  - Class Creational (Tvořivé návrhové vzory)
- **Využití**
  - Vytváření frameworků
    - Třída nezná objekty, které má vytvářet
    - Potomci specifikují vytvářené objekty
      - záleží na programátorech, kolik konkrétních potomků vytvoří
  - Oddělení kódu, který se mění „často“, od „neměnného“

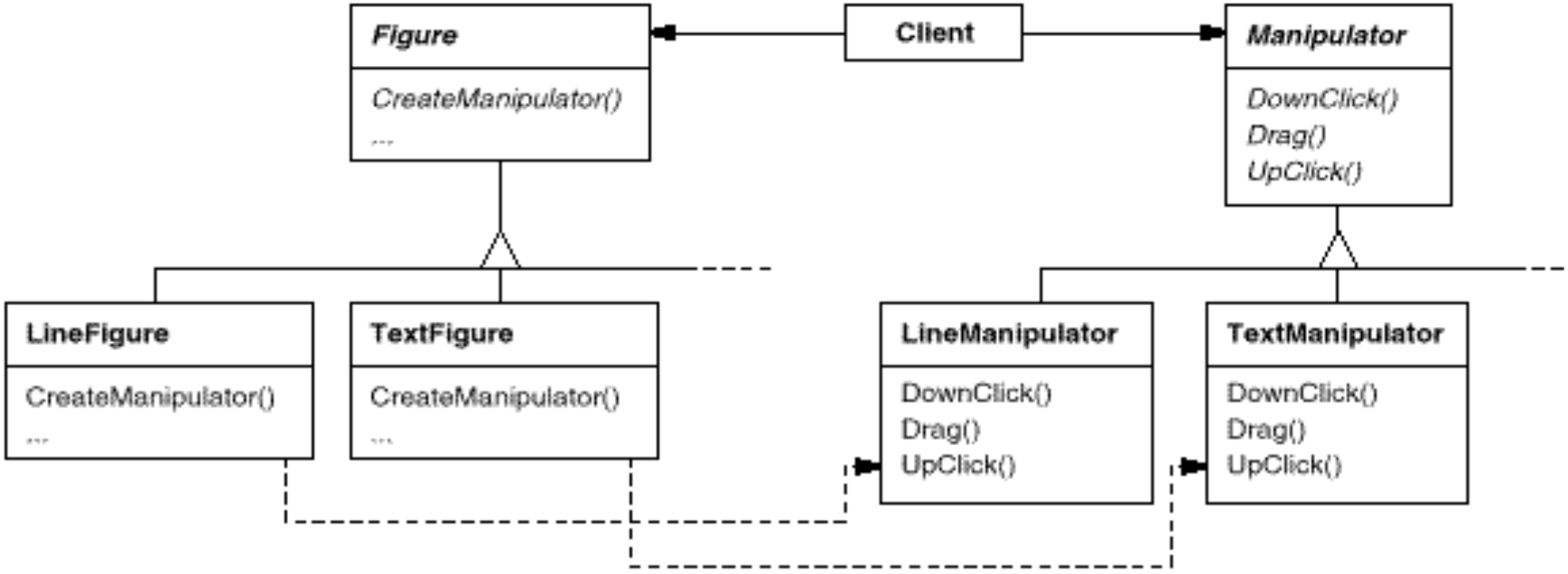
FACTORY METHOD - STRUKTURA

- **Struktura**



- **Účastníci**
  - Product (`Car`)
    - definuje rozhraní objektů vytvářených tovární metodou
  - ConcreteProduct (`Fabia`)
    - implementuje rozhraní Productu
  - Creator (`CarFactory`)
    - deklaruje tovární metodu vracející objekt typu Product
    - může definovat defaultní implementaci vracející defaultní objekt ConcreteProduct
  - ConcreteCreator (`FordFactory`)
    - implementuje tovární metodu vracející instanci ConcreteProductu

FACTORY METHOD - PARALELNÍ HIERARCHIE



- **Propojení paralelních hierarchií tříd**
  - třída deleguje některé akce na jinou třídu
  - příklad: manipulovatelné grafické objekty (úsečka, obdélník, kruh, ...)
    - stav manipulace je nutné udržovat pouze během manipulace - ne u grafického objektu
    - manipulace různých objektů uchovávají jiný stav (koncový bod, rozteč řádků, ...)
    - objekt `Manipulator`, konkrétní grafické objekty mají vlastní manipulátory
    - paralelní hierarchie
    - `Figure` deklaruje `CreateManipulator`, potomci vytvářejí vlastní manipulátory
    - částečně paralelní hierarchie - dědění defaultního manipulátoru

FACTORY METHOD - IMPLEMENTACE

- **Implementace**
  - dvě hlavní varianty
    - 1. Creator je abstraktní třída, nemá vlastní implementaci tovární metody
      - nutnost dědičnosti a vlastní implementace
    - 2. Creator je konkrétní třída s defaultní implementací
      - flexibilita
      - pravidlo: Vytvářet objekty v separátní metodě => potomci mohou ovlivnit (polymorfismus), které třídy se budou instanciovat.
  - 3. parametrizované tovární metody (viz níže)
    - variace - více druhů produktů
    - tovární metoda dostane parametr identifikující druh objektu (ProdId)
    - všechny objekty sdílejí rozhraní Productu

```
class Creator {
public:
  virtual Prod* Create(ProdId);
};
Prod* Creator::Create (ProdId id) {
  if(id==MINE)  return new MyProd;
  if(id==YOURS) return new YourProd;
  return 0;
}
```

```
Prod* MyCreator::Create (ProdId id) {
  if(id==MINE)   return new MyProd2;
  if(id==THEIRS) return new TheirProd;
  return Creator::Create(id);
}
```

Obsluha ostatních id se deleguje na předka

FACTORY METHOD - IMPLEMENTACE

- Pozdní inicializace (Lazy load)
  - tovární metody jsou vždy virtuální, často čistě virtuální - pozor na volání z konstruktorů
  - lazy initialization - přístup pouze přes přístupovou metodu (accessor)
  - konstruktor Creatoru inicializuje ukazatel na 0
  - v případě potřeby se produkt vytvoří 'na žádost'

```
class Creator {
public:
  Product* GetProduct();
protected:
  virtual Product* CreateProduct();
private:
  Product* _product;
};

Product* Creator::GetProduct () {
  if (_product == 0) {
     _product = CreateProduct();
  }
  return _product;
}
```

accessor

inicializace na žádost

FACTORY METHOD - IMPLEMENTACE

- použití šablon
  - někdy zbytečně pracné vytvářet konkrétní potomky
  - šablona parametrizovaná Produktem
  - není zapotřebí vytvářet potomky Creatora

```
class Creator {
public:
  virtual Product* CreateProduct() = 0;
};

template <class T>
class StdCreator: public Creator {
public:
  virtual Product* CreateProduct();
};

template <class T>
Product* StdCreator<T>::CreateProduct() {
  return new T;
}

class MyProduct : public Product {
public:
  MyProduct();
  // ...
};

StdCreator<MyProduct> myCreator;
```

vrací konkrétního potomka abstraktního produktu

v Javě není možné, ale lze přes reflexe ☺

FACTORY METHOD – PŘÍKLADY POUŽITÍ

- **Qt gui framework (C++)**

```
virtual Qmenu* QmainWindow::createPopupMenu()
```

- **Java API – XML parsery, DateFormat třída**
  - DocumentBuilderFactory, DocumentBuilder
  - SAXParserFactory, SAXParser

Factory method

```
DateFormat df1 = DateFormat.getDateInstance(DateFormat.FULL, Locale.FRANCE);
```

- **Zend webový aplikační framework (PHP)**
  - Třída Zend_Db – connector k různým typům databází

```
// require_once 'Zend/Db/Adapter/Pdo/Mysql.php';
// Automatically load class Zend_Db_Adapter_Pdo_Mysql
// and create an instance of it.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

Factory method

Pdo_Sqlite, Pdo_Mssql, Oracle, DB2, ...

FACTORY METHOD – PŘÍKLADY POUŽITÍ

```
public static function factory($adapter, $config = array()) {
        // obtaining  adapter class name
        $adapterName = strtolower($adapterNamespace . '_' . $adapter);

        // Load the adapter class.  This throws an exception
        // if the specified class cannot be loaded.
        @Zend_Loader::loadClass($adapterName);

        // Create an instance of the adapter class.
        // Pass the config to the adapter class constructor.
        $dbAdapter = new $adapterName($config);

        // Verify that the object created is
        // a descendent of the abstract adapter type.
        if (! $dbAdapter instanceof Zend_Db_Adapter_Abstract) {
            // throwing Zend_Db_Exception
            require_once 'Zend/Db/Exception.php';
            throw new Zend_Db_Exception("Adapter class '$adapterName' does not "
                                        "extend Zend_Db_Adapter_Abstract");
        }
        return $dbAdapter;
}
```

Factory method

Concrete Product

test of Product base class

FACTORY METHOD - SOUVISEJÍCÍ NÁVRHOVÉ VZORY

- **Shrnutí**
  - flexibilita za relativně malou cenu
  - obvyklá základní metoda zvýšení flexibility
    - virtual constructor
  - časté využití jinými vzory
- **Abstract Factory**
  - často implementovaná pomocí továrních metod
- **Template Method**
  - často volány tovární metody
- **Prototype**
  - nevyžaduje dědění Creatora
  - ... avšak vyžaduje navíc inicializaci produktové třídy

CO DÁL

A7B36ASS - NÁVRHOVÉ VZORY (STM)

A4M36JEE - VE SPOLUPRÁCI S FIRMOU RED HAT