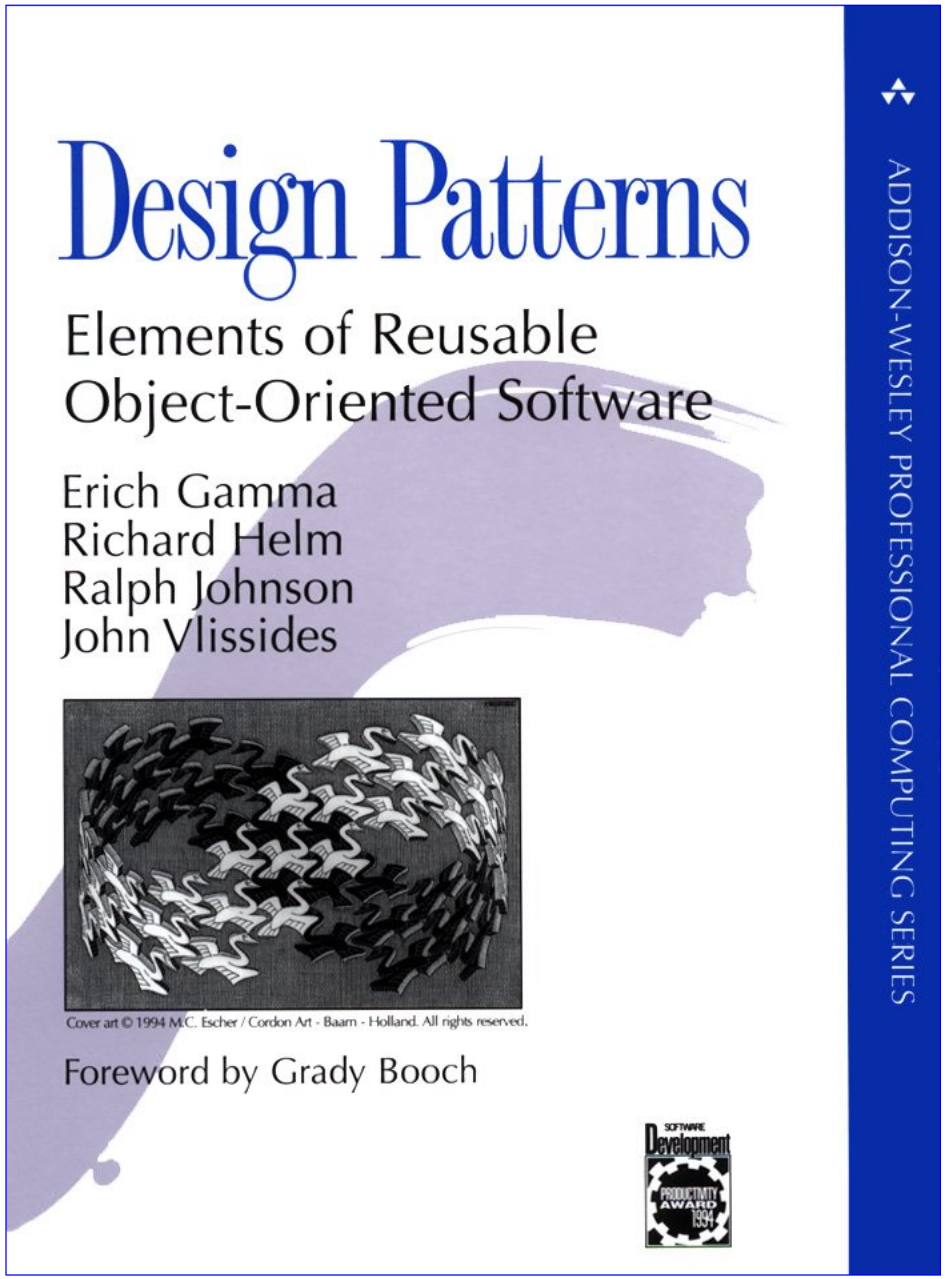
Návrhové vzory

– motivace pro 36ASS

Literatura

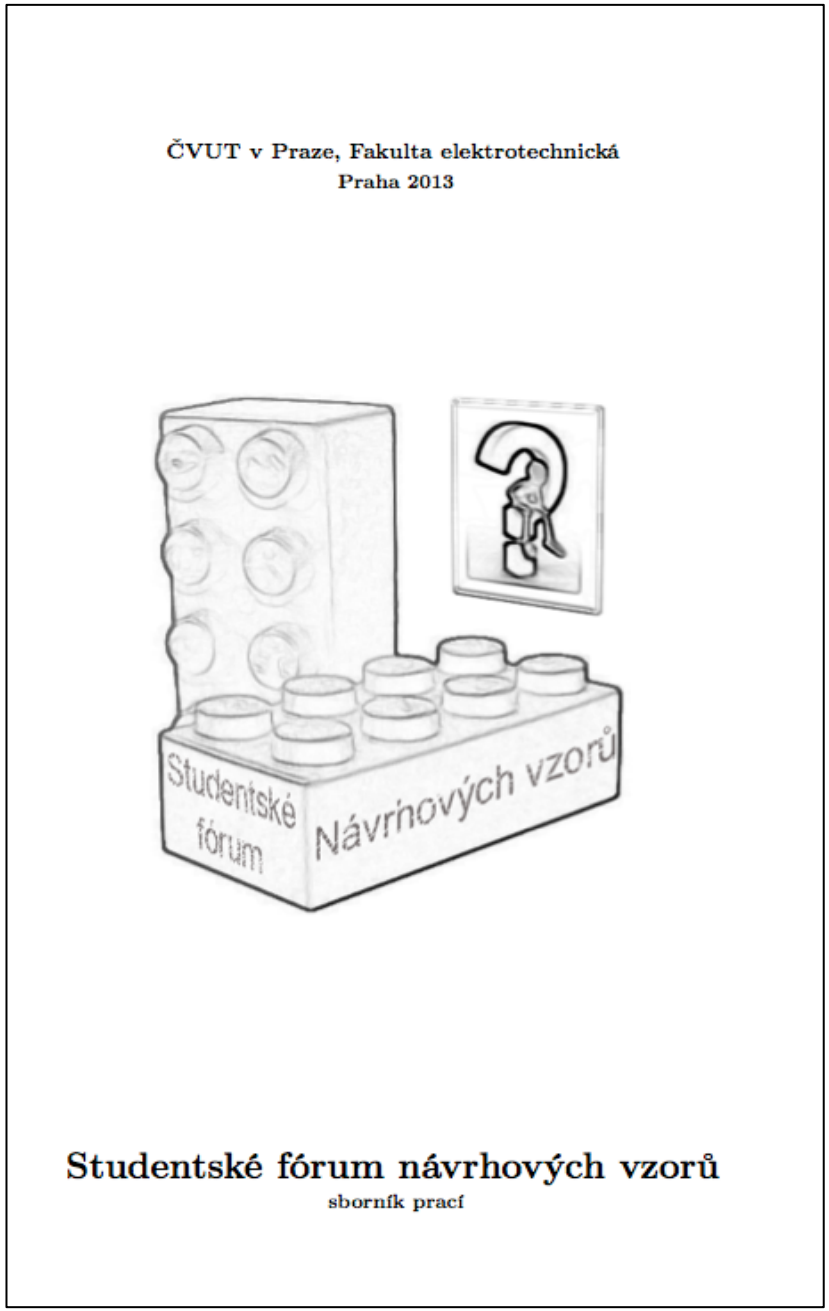
- **E. Gamma, R. Helm, R. Johnson, J. Vlissides**
  The Gang of Four (GoF)
  Design Patterns
  Elements of Reusable Object-Oriented Software
  1995
  Grada 2003:
  Návrh programů podle vzorů

- J. Vlissides:
  Pattern Hatching - Design Patterns Applied



Literatura

- **Černý a studenti, 2013**
- Free
- http://wiki.cs.czacm.org/xwiki/bin/download/36ASS/osnova/sbornik2-online.pdf



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

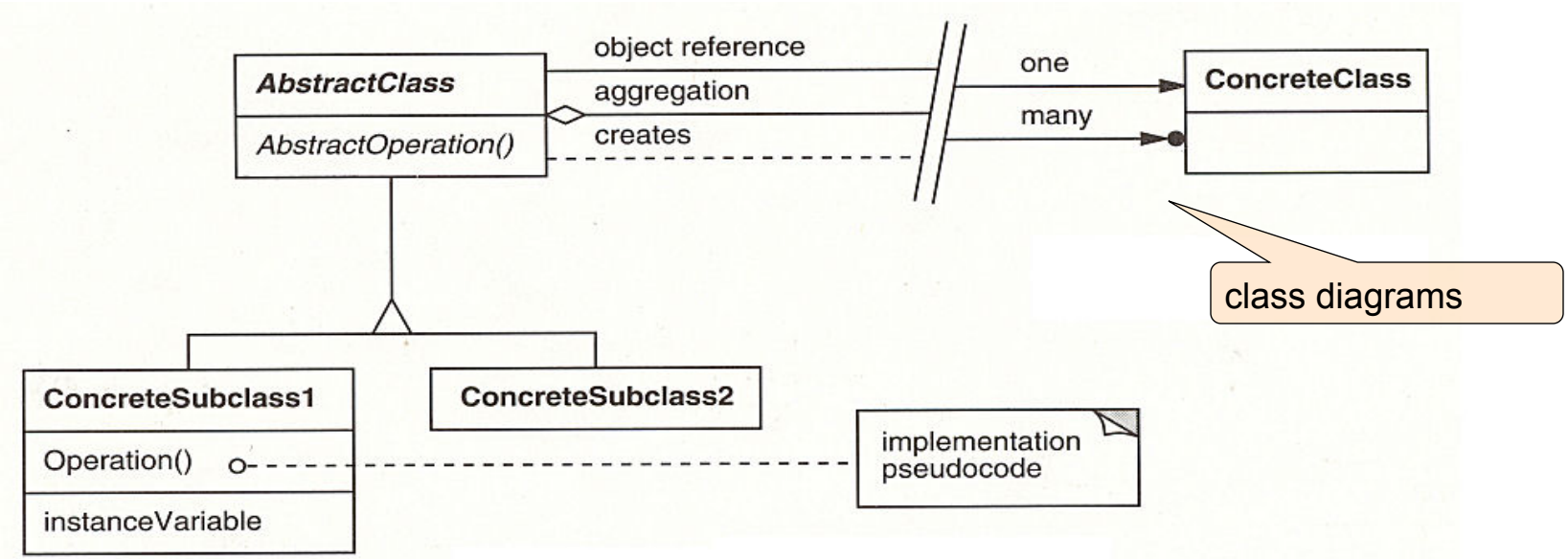
	Creational *Tvořivé vzory*	**Structural** *Strukturální vzory*	**Behavioral** *Vzory chování*
Třída	**Factory Method**	**Adapter**	**Interpreter** **Template Method**
Objekt	**Abstract Factory** **Builder** **Prototype** **Singleton**	**Bridge** **Composite** **Decorator** **Facade** **Proxy** **Flyweight**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

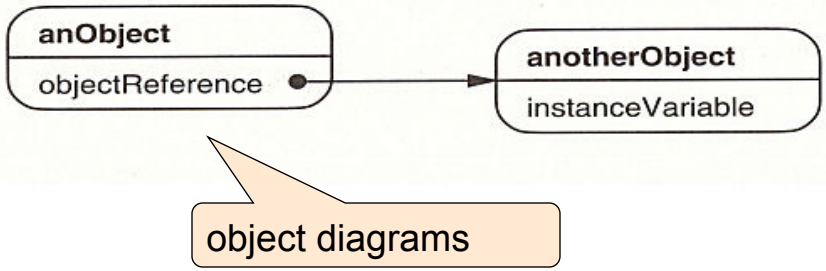
vytváření objektů

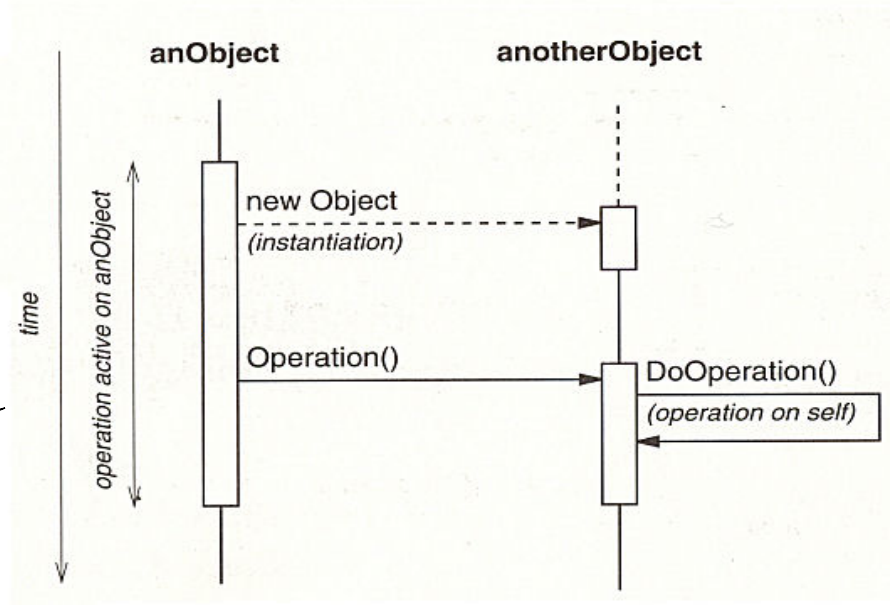
uspořádání tříd a objektů

chování a interakce objektů a tříd

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

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

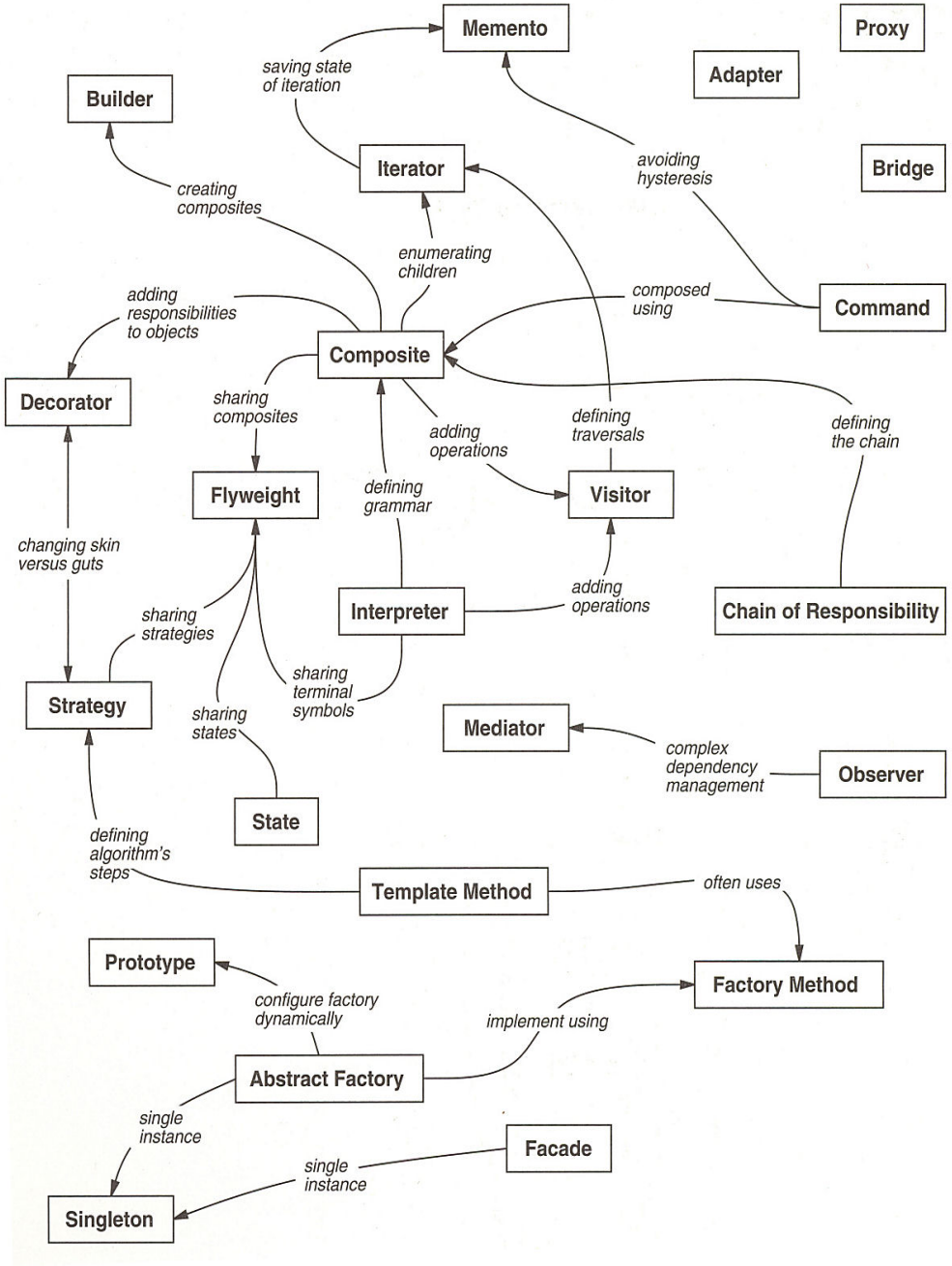
NV chování

- **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

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*

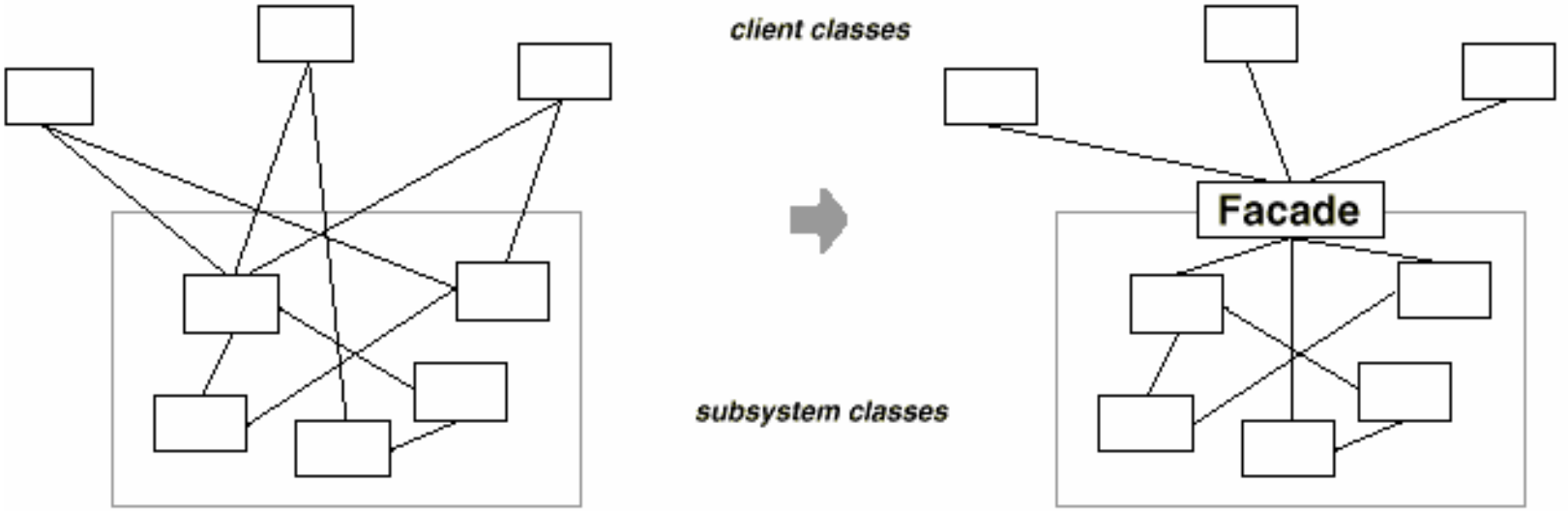


Facade

Facade

- **Známý jako**
  - Facade, Fasáda
- **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:

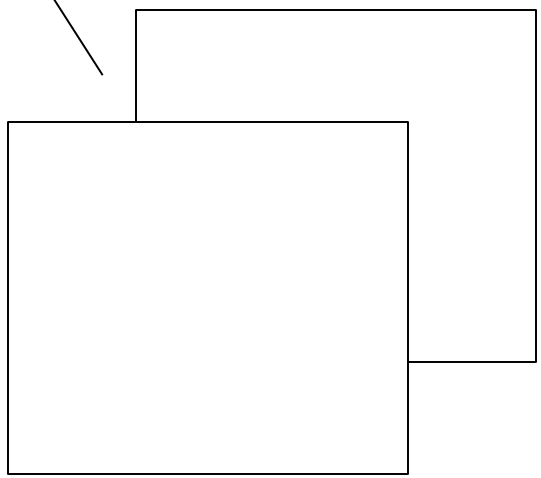
- vypnout DVD přehrávač
- vypnout ozvučení
- vypnout projektor
- vytáhnout žaluzie

Facade – motivace

Řešení:

Univerzální ovladač s funkcemi:

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

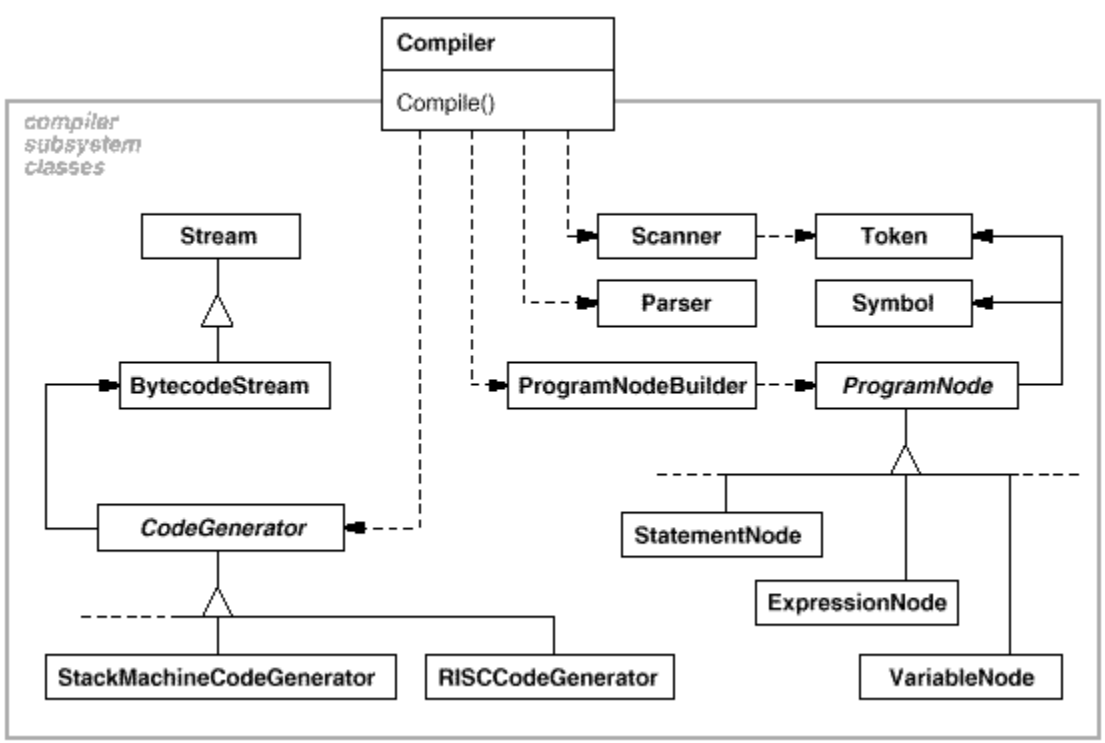


- zatáhnout žaluzie
- zapnout projektor
- zapnout DVD přehrávač
- zapnout ozvučení

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

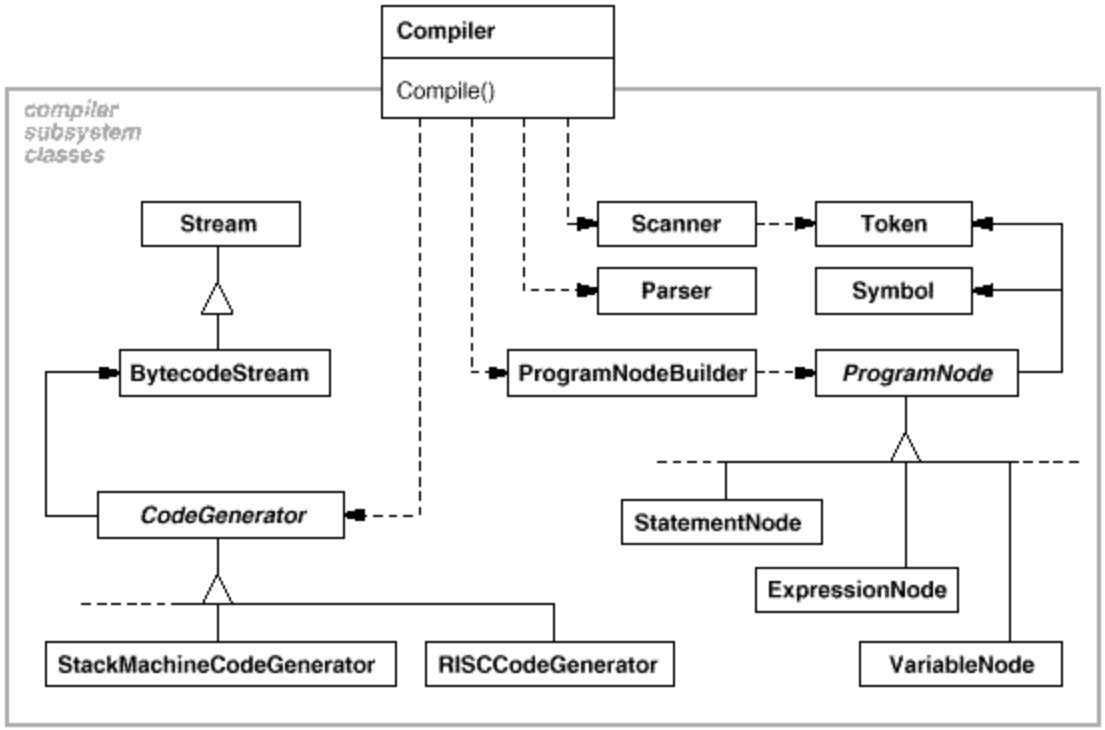
Facade - motivace

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



Facade - motivace

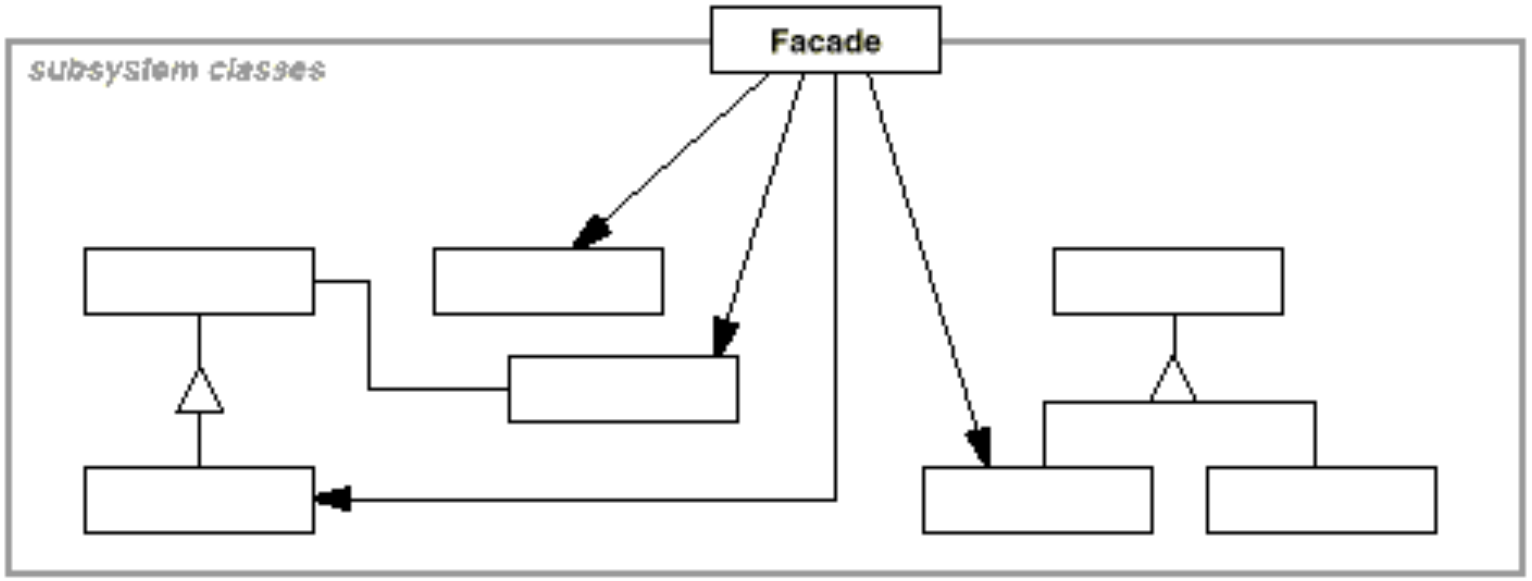
- **Příklad**
  - *transparentnost* – třídy subsystému o Facade nevědí
  - *svoboda volby*- neskrývá třídy subsystému (Parser, Scanner)



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

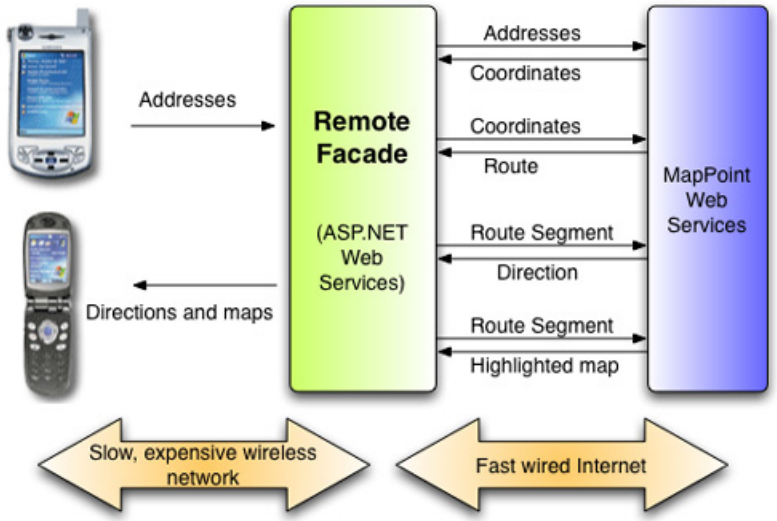
```
class Compiler {
public:
  Compiler();
  virtual void Compile(istream&,BytecodeStream&);
};

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

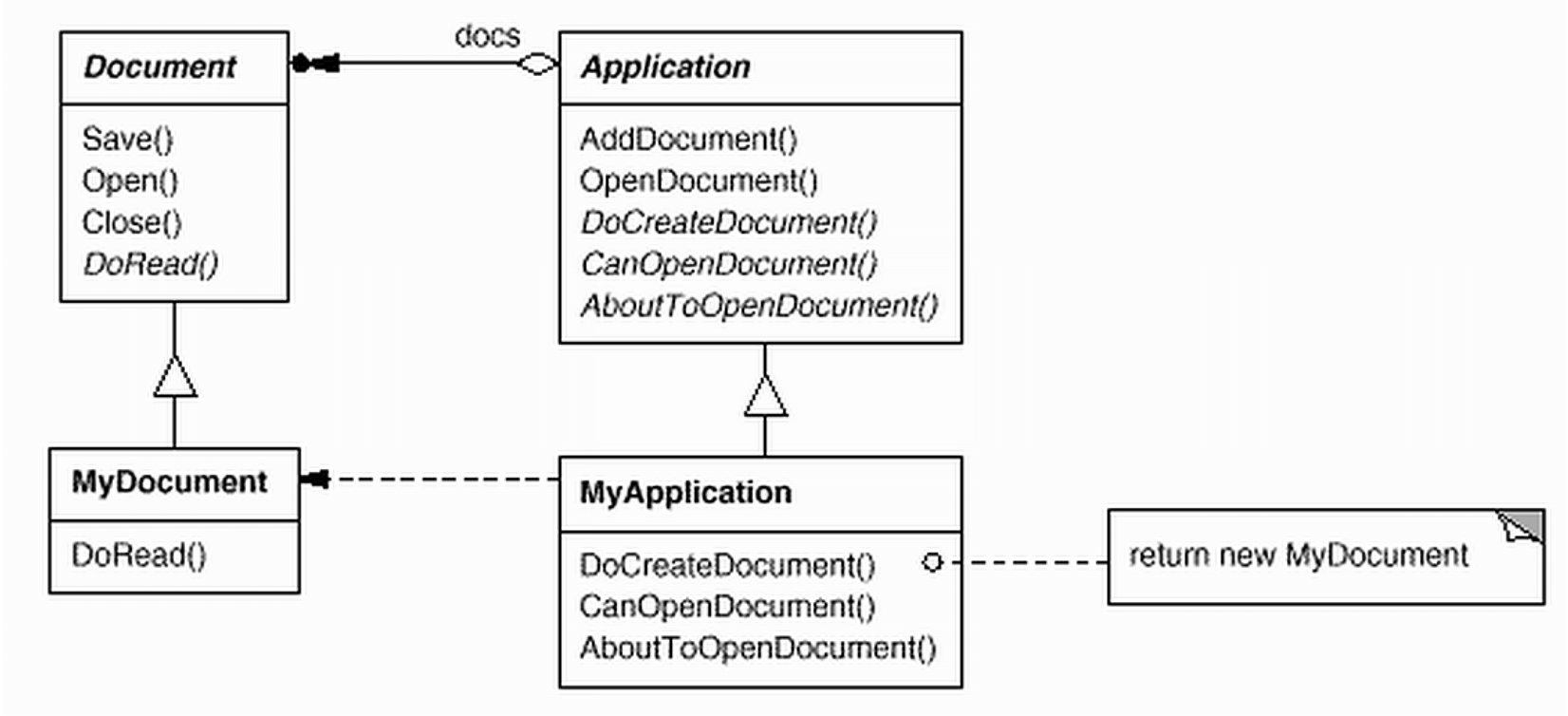
Template method

Motivační příklad – reálný svět

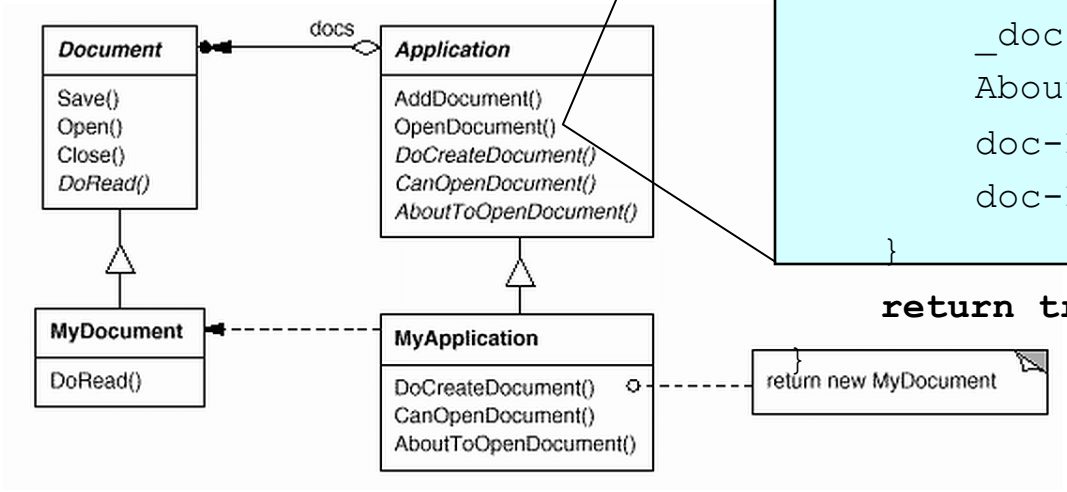
- **Pásová výroba (assembly line)**

Motivační příklad – SW inženýrství

- **Otevírání souborů**
- **Třídy `Application` a `Document` (+potomci)**



Motivační příklad – SW inženýrství



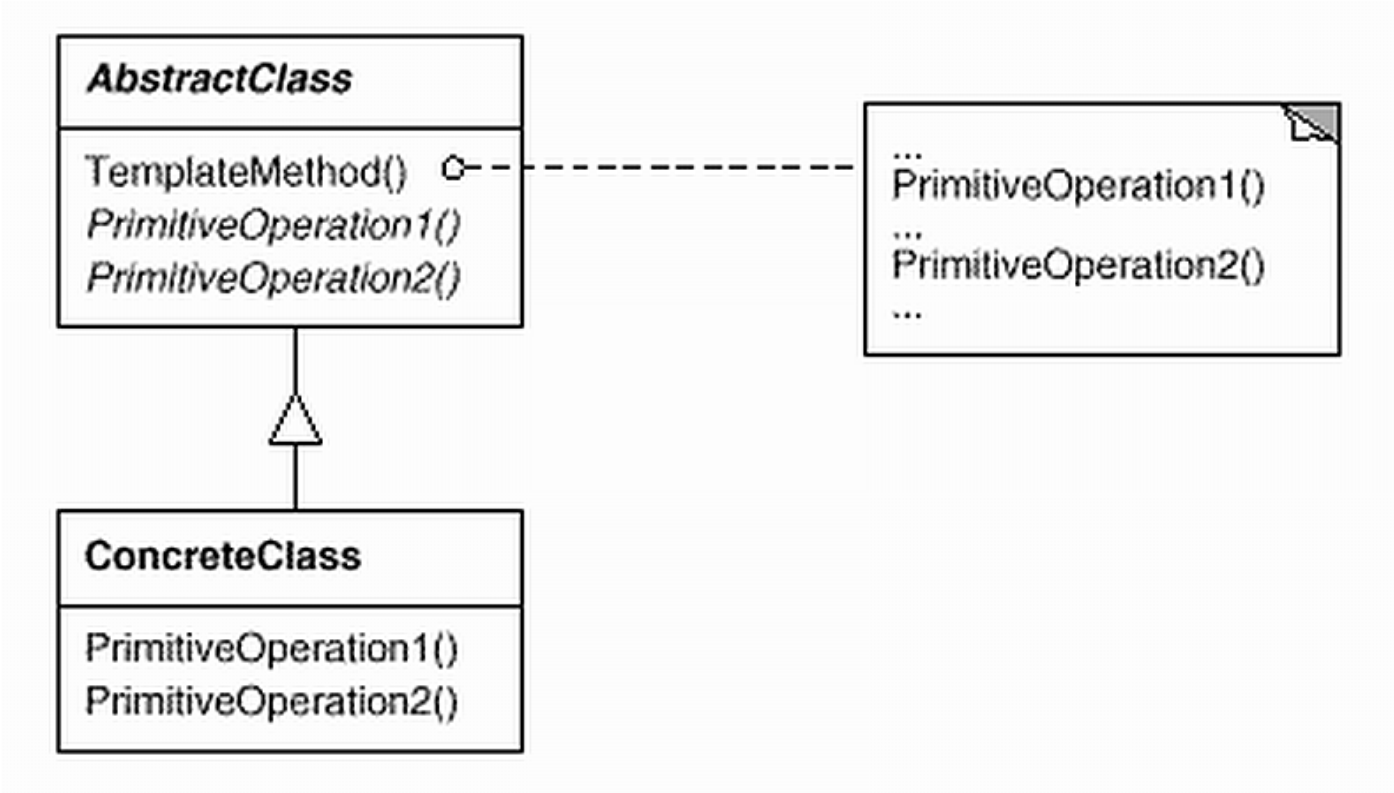

```
bool Application::OpenDocument() {
    if (!CanOpenDocument(name)){
        return false;
    }
    Document *doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open(name);
        doc->DoRead();
    }
    return true;
}
```

- `OpenDocument()` – Template method
- `DoCreateDocument()`, `CanOpenDocument()` – primitivní operace

Účel

- **Definice skeletu algoritmu**
  - Pořadí operací
  - Invarianty
- **Agregace totožného kódu množiny tříd**
  - Template method u rodiče
  - Potomkové definují lišící se části
  - „Refactoring to generalize“
- **Řízená dědičnost**
  - Hook operations v Template method

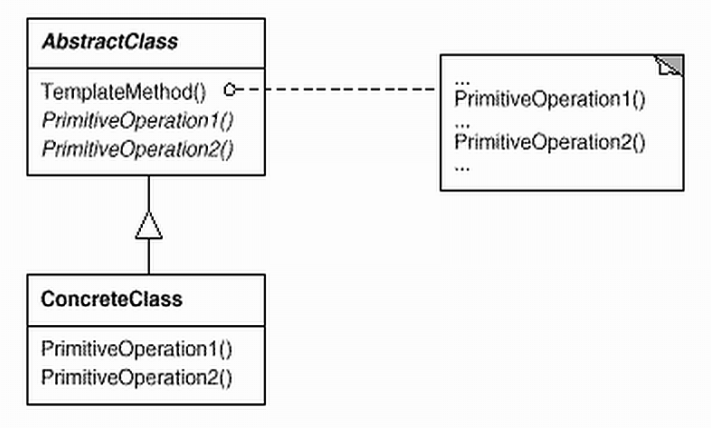
Struktura



- Inverzní princip (Hollywood principle)
  - „Don't call us, we'll call you"

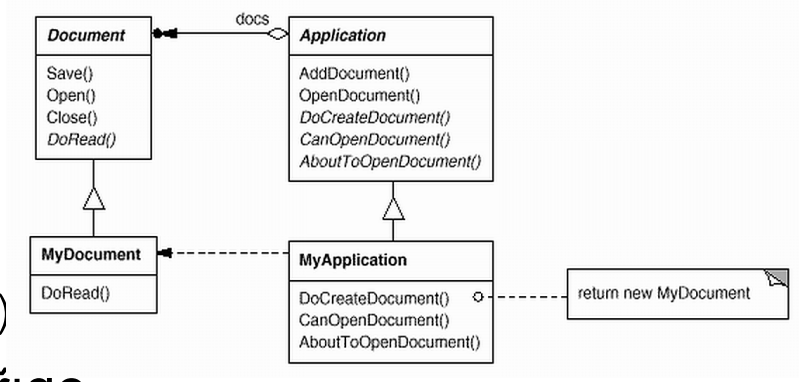
Účastníci

- **`AbstractClass`**
  - Definuje abstraktní primitivní operace k dědění
  - Implemntuje template metodu a definuje skeleton algortimu
  - Template metoda volá primitivní operace ale i další definované v Abstr. classs
- **`ConcreteClass`**
  - Implementuje primitivní operace pro změnu algoritmu



Template Metoda volá:

- **Konkrétní operace v `ConcreteClass`**
- **Konkrétní operace v `AbstractClass`**
  - Invarianty
  - `AddDocument()`
- **Primitivní (abstraktní) operace**
  - `CanOpenDocument()`, `DoRead()`
  - Implementovány pak v konkrétní třídě
- **Factory method-y**
  - `DoCreateDocument()`
- **Hook operations**
  - Nepovinné – řízená extenze třídy
  - `AboutToOpenDocument()`



Hook operations

- **Specifická funkcionalita `DerivedClass`**
- **Nemusí být implementovány**

```
void ParentClass::Operation() {
        // ParentClass behaviour
        HookOperation();
}
void ParentClass::HookOperation() {
}

…

void DerivedClass::HookOperation() {
        // DerivedClass extension
}
```

Implementace

- **Přístupová práva a modifikátory OO jazyků**
  - Primitivní operace – `protected`, čistě virtuální
  - Template method – `public`, nevirtuální
- **Minimalizace počtu primitivních operací**
  - Protože
  - to
  - komplikuje
  - implementaci
- **Pojmenovávací konvence**
  - Primitivní operace - prefix „`Do-`“ (MacApp framework)
  - Hook operations – prefix „`Hook-`“

Příklad – NeXT AppKit

Příklad – NeXT AppKit

- Třída `View` – vykreslování do grafického kontextu
- Před samotným vykreslením potřebuje focus

```
void View::Display() {
        SetFocus();
        DoDisplay();
        ResetFocus();
}
void View::DoDisplay() {
}

…

void CustomView::DoDisplay {
        // render something awesome
}
```

Příklad – JUnit testy

- **Při každém automatickém testu potřeba spustit několik akcí**
  - `setUp()` – připraví zdroje
  - `runTest()` – vykoná test
  - `tearDown()` – uvolní zdroje
  - `runBare()` – Template method

JUnit.org

```
public abstract class TestCase {
public void runBare() throws Throwable {
setUp();
try {
runTest();
}
finally {
tearDown();
}
}

protected void setUp() throws Exception {
}

protected void tearDown() throws Exception {
}

protected void runTest() throws Throwable {
}
}
```

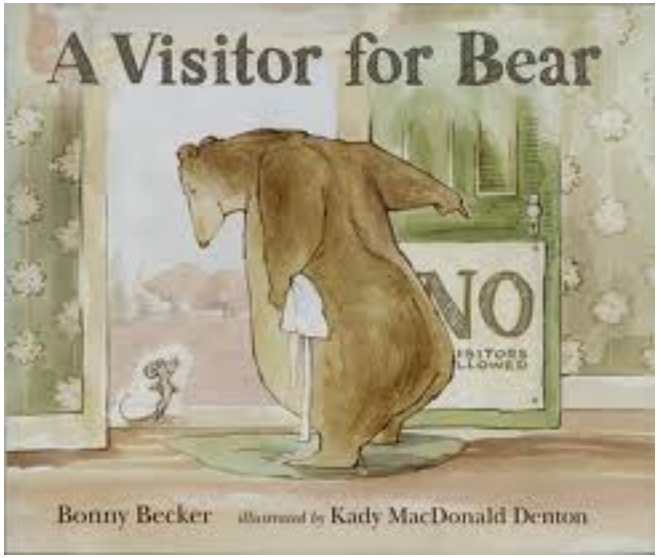
Související NV

- **Factory method**
  - e.g. `DoCreateDocument()` volána pomocí template metody `OpenDocument()`
  - Primitivní operace
  - Často volána pomocí template metody
- **Strategy**
  - Změna celého algoritmu delegací
  - Zdědění template metody k částečné změně algoritmu

Template method - Shrnutí

- **Název**
  - Lehce zavádějící
- **Problém**
  - Definice obecného algoritmu, jehož struktura zůstane zachována i v jeho konkrétních implementacích
- **Řešení**
  - Mateřská třída implementuje kostru algoritmu
  - Podtřída implementuje jeho jednotlivé kroky
- **Implementace**
  - Využití přístupových práv a virtuality metod
- **Příklady**
  - Obecně každý framework/knihovna

VISITOR

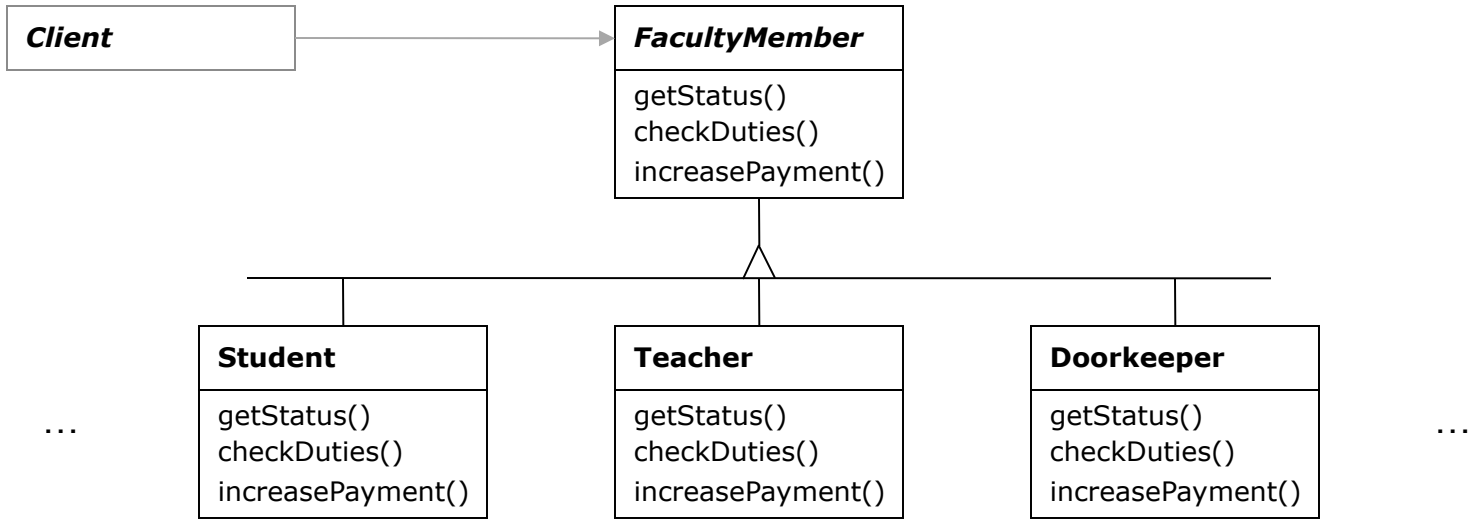

A Visitor for Bear
NO
Bonny Becker
illustrated by Kady MacDonald Denton

VISITOR

- **Známý jako návštěvník**
  - výstižnější analogie je audit, nebo návštěva tchýně
    - ovšem za podmínky, že často měníte manželku
- **Účel**
  - umožňuje přidat nové operace do existující hierarchie bez její modifikace
- **Motivační příklad**
  - máme složitou strukturu objektů (např. IS pro školu…)
    - vyučující, studenti, administrativní pracovníci …
  - chceme na ní provádět nejrůznější operace
    - výkazy činností, kontroly splněných povinností …
  - počet typů objektů je velký, ale neměnný
  - ministerstvo/rektorát neustále vydávají nová nařízení
    - často se mění (a přidávají operace)

VISITOR – MOTIVAČNÍ PŘÍKLAD

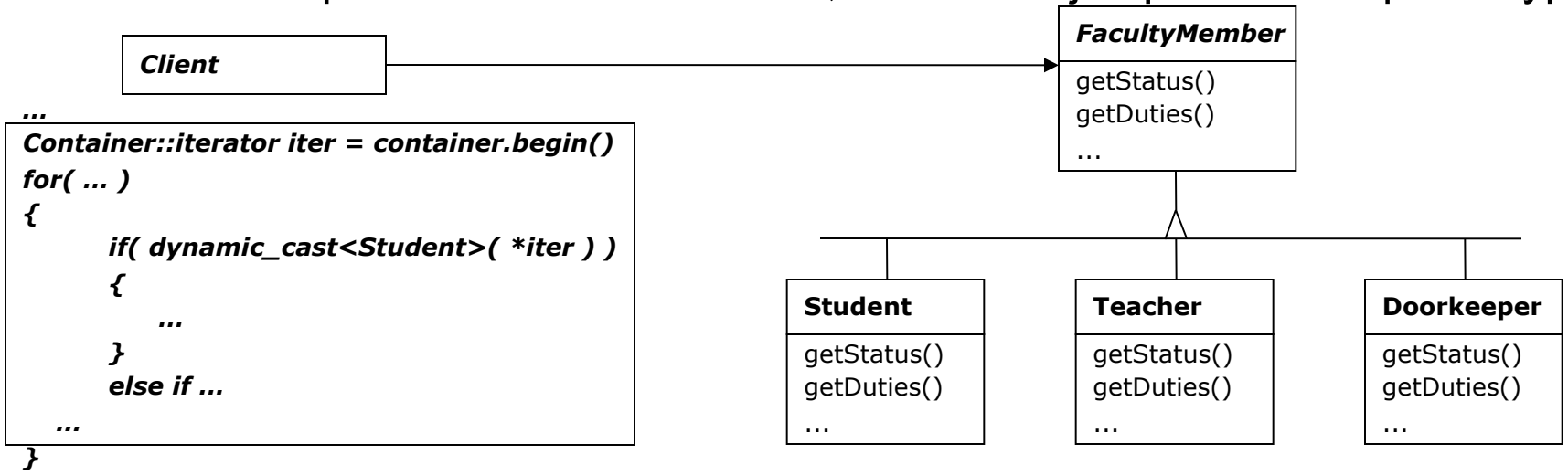
- **Klasické řešení**
  - umístíme operace do jednotlivých tříd



- **Nevýhody**
  - struktura objektů je méně přehledná a hůře se udržuje
  - algoritmus každé operace je rozdělen v několika třídách
  - přidání operace vyžaduje změnu všech tříd

VISITOR – MOTIVAČNÍ PŘÍKLAD

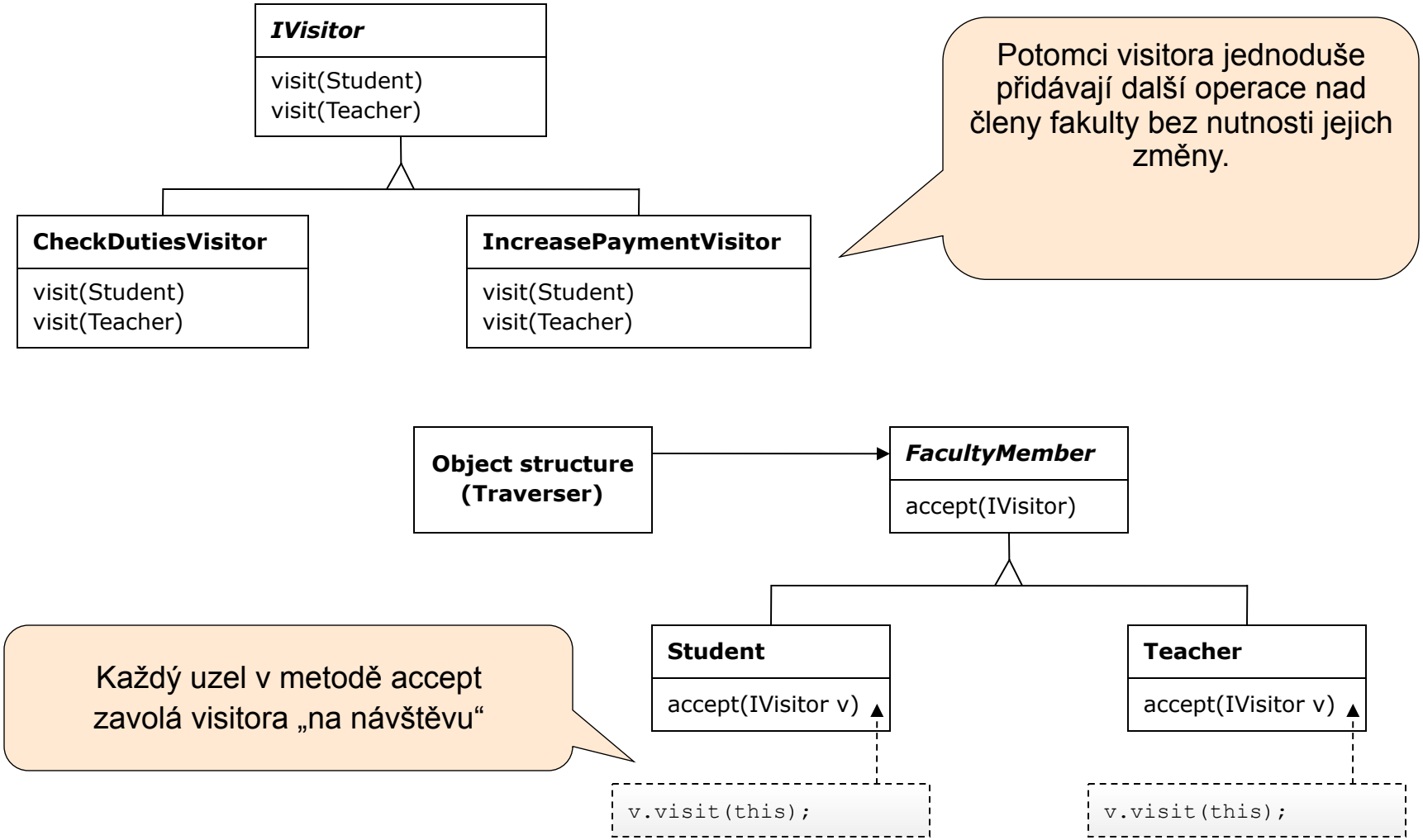
- **Druhý nápad !!!!!**
  - umístníme operace někam do clienta, a budeme je zpracovávat podle typů



- Výhody
  - struktura objektů je „víc“ přehledná
- Nevýhody
  - pro každou operaci máme jeden switch blok
  - zrádný a nepřehledný switch blok
- Lepší řešení – použijeme Visitora …

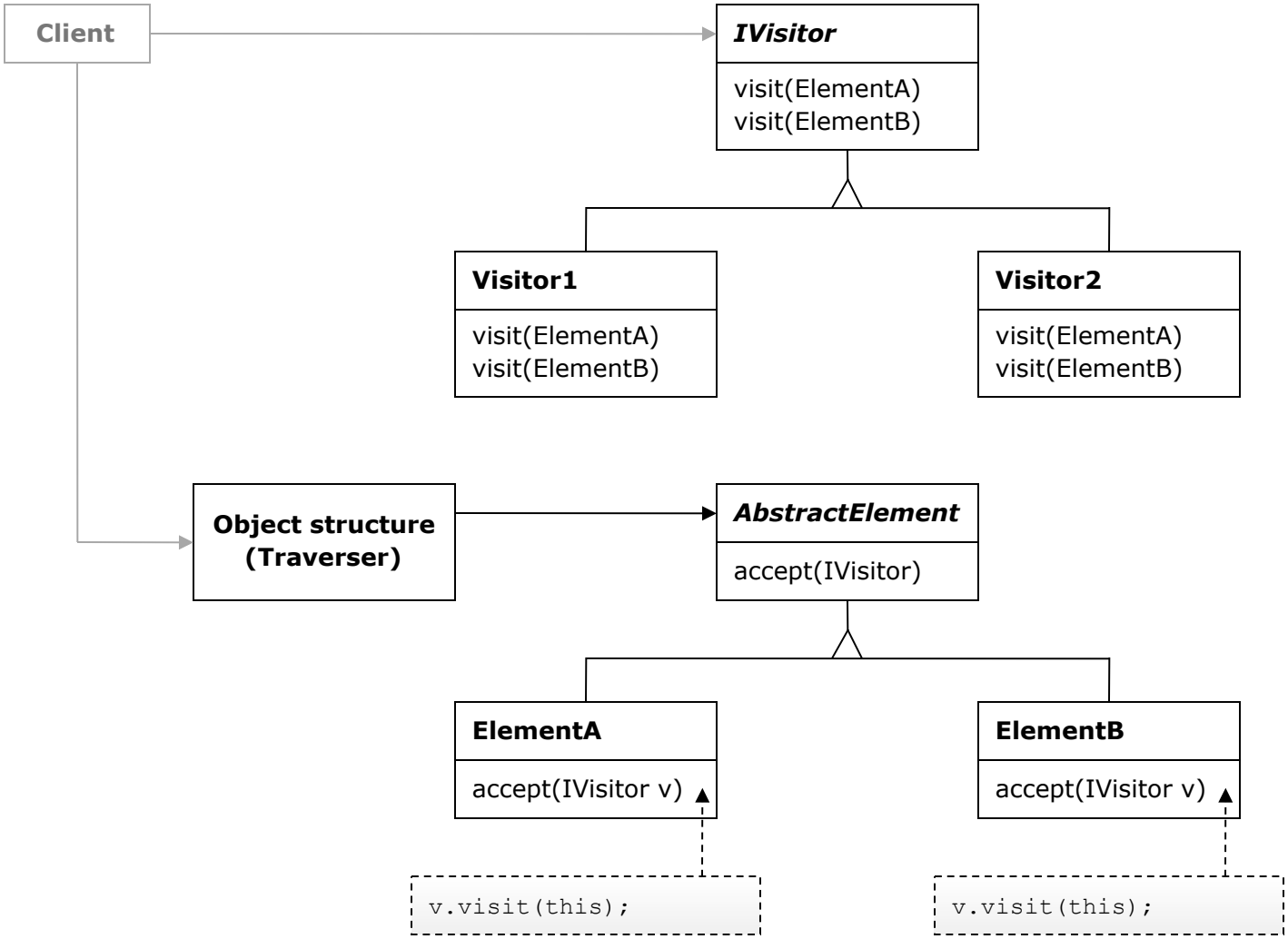
VISITOR – ŘEŠENÍ MOTIVAČNÍHO PŘÍKLADU

- **Řešení motivačního příkladu s použitím Visitoru**



VISITOR - STRUKTURA

- **Struktura**



VISITOR – ÚČASTNÍCI

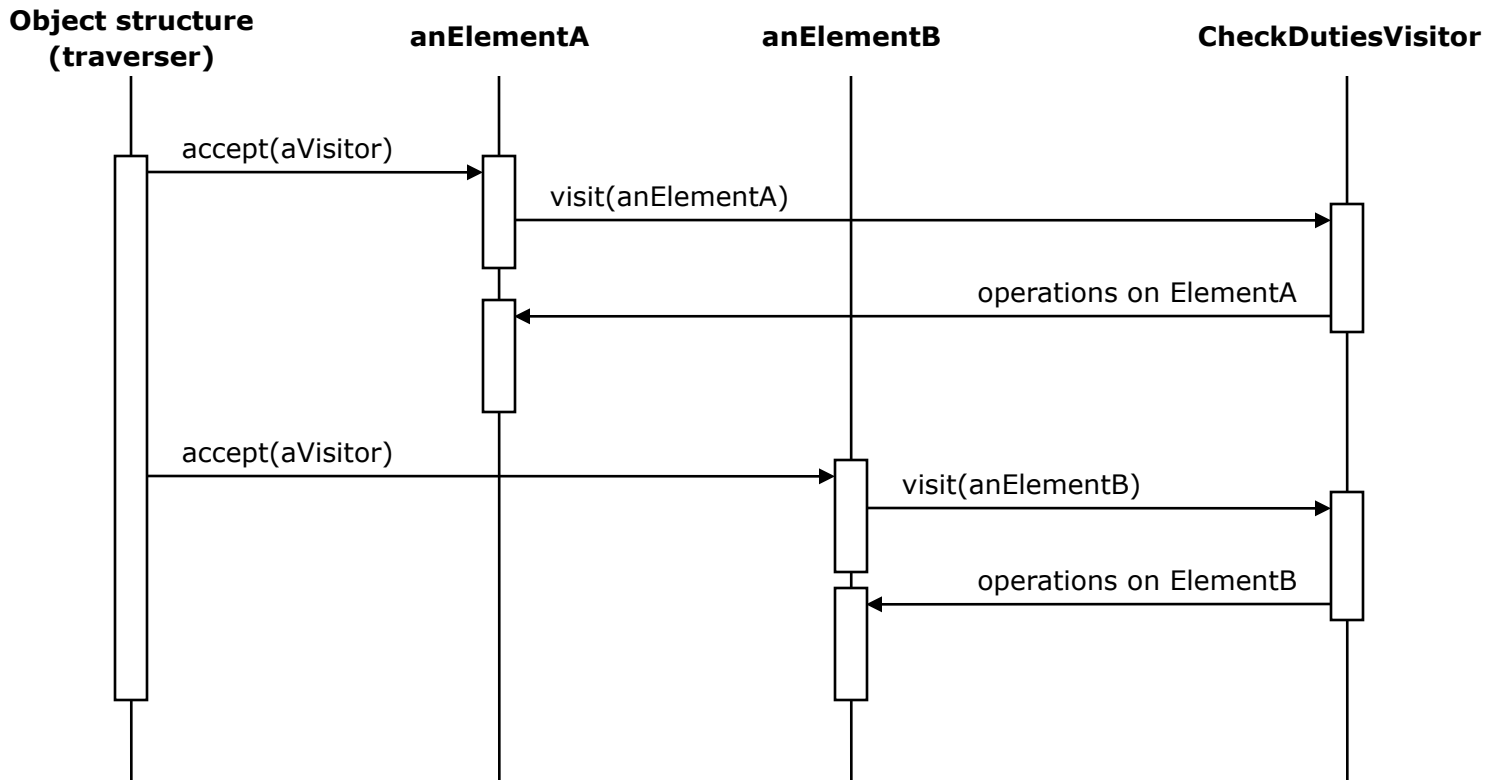
- **Přehled účastníků**
  - IVisitor
    - interface (nebo abstraktní třída), který musí implementovat konkrétní Visitory
    - definuje metody visit pro všechny typy elementů
      - může využívat i overloading funkcí:

```
visit( ElementA& );
visit( ElementB& );
```

  - Visitor1, Visitor2
    - konkrétní Visitory (implementují rozhraní IVisitor)
    - přidávají novou funkcionalitu do existující struktury
  - AbstractElement
    - abstraktní třída pro všechny typy, které mohou být navštíveny Visitorem
    - definuje abstraktní metodu accept
  - ElementA, ElementB
    - konkrétní elementy odvozené od AbstractElement
    - implementují metodu accept (uvnitř které pozve předaného Visitora na návštěvu)
  - Object structure (traverser)
    - umí procházet strukturu elementů
    - na každém elementu zavolá metodu visit

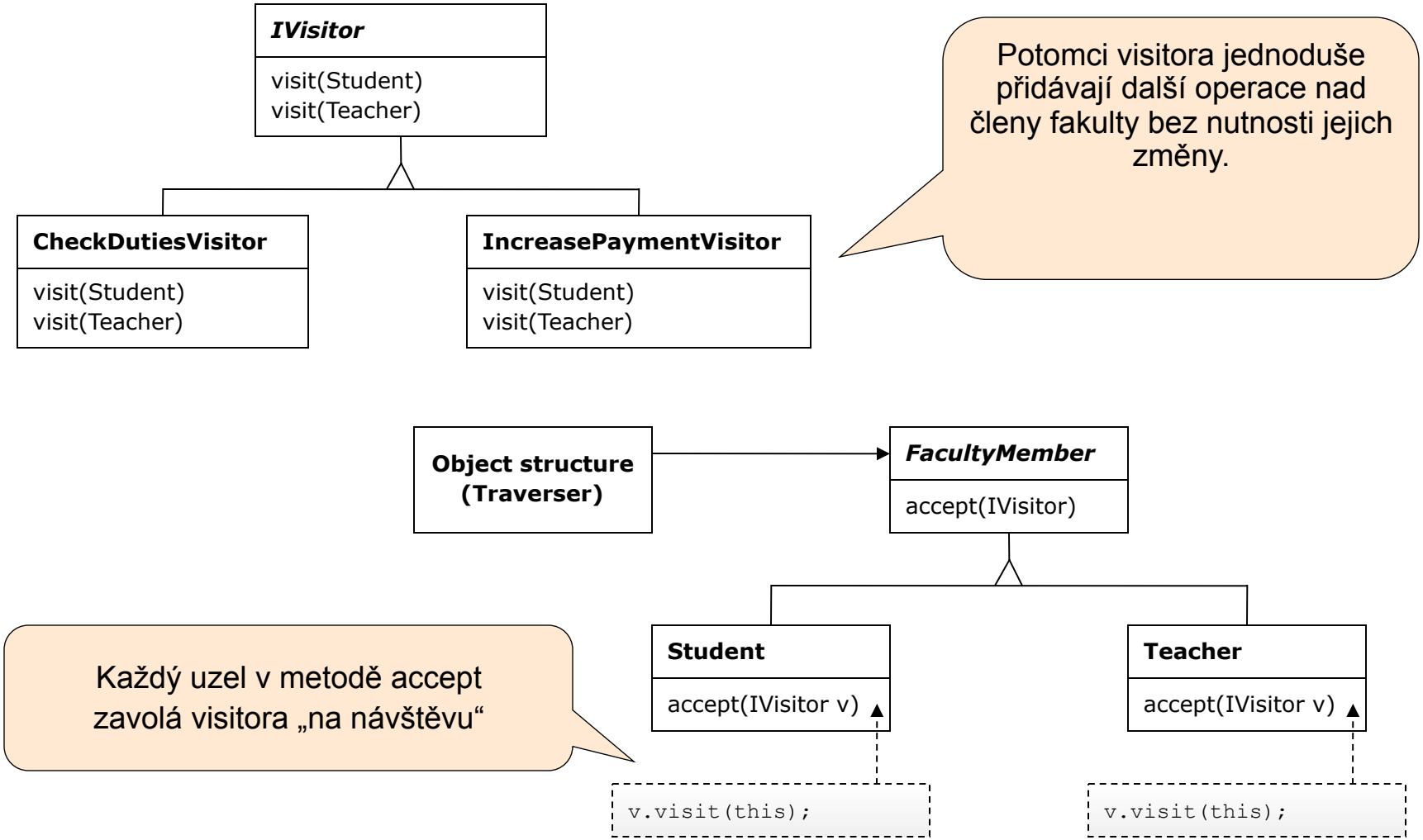
VISITOR - SPOLUPRÁCE

- **Spolupráce účastníků**



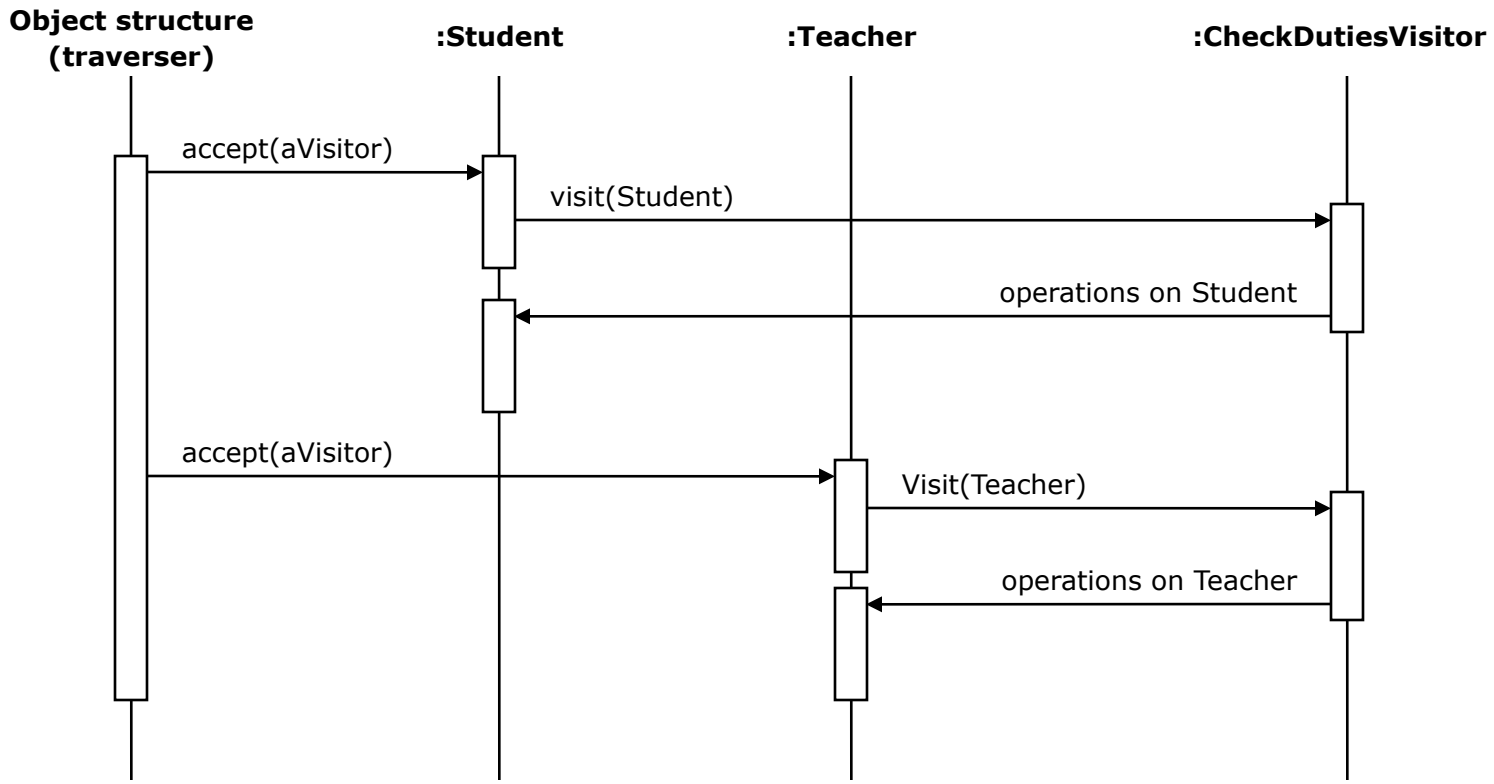
VISITOR – ŘEŠENÍ MOTIVAČNÍHO PŘÍKLADU

- **Řešení motivačního příkladu s použitím Visitoru**



VISITOR - SPOLUPRÁCE

- **Spolupráce účastníků**



VISITOR – DŮSLEDKY A SOUVISLOSTI

- **Důsledky a souvislosti**
  - snadné přidávání nových operací
    - není třeba měnit a rekompilovat objekty, nad kterými se operace provádí
    - třídy objektů zůstávají přehledné a snadno udržovatelné
    - Visitory mohou být implementovány jako zásuvné moduly
  - zapouzdření souvisejících operací
    - Visitor skrývá specifika algoritmu
  - udržování kontextu při průchodu strukturou objektů uvnitř Visitoru
    - Visitor může mít vnitřní stav (data)
      - nemusí se předávat parametrem, nebo v globálních hodnotách
      - vnitřní stav může ovlivnit prováděné operace
  - přidání nového typu objektu většinou znamená přepsání všech Visitorů
    - nasazení při častěji měnící se struktuře je nevhodné
  - porušení zapouzdření objektů, nad nimiž se operuje
    - Visitor může potřebovat pracovat s interním stavem
  - Visitor vytváří cyklickou závislost mezi Elementy a Visitorem
    - řešení: Acyklický Visitor (dvě hierarchie tříd, dynamic_cast)

VISITOR - IMPLEMENTACE

- **Implementace Visitoru**
  - každá struktura objektů má přiřazenu vlastní hierarchii Visitorů
  - Visitor musí implementovat daný interface
    - interface je pevně svázán se strukturou, kterou Visitor rozšiřuje
    - žádné další požadavky na něj kladeny nejsou
  - objekty, které Visitor navštěvuje mohou mít společného předka, ale nemusí
  - Visitor jako Singleton
    - pokud nemá žádné vnitřní stavy
  - Visitor jako FlyWeight
    - pokud má vnitřní parametry (read only) a používá se často
  - v ostatních případech se Visitor vytváří účelově pro jedno volání
  - v některých případech nechceme ve Visitoru metodu pro každý objekt hierarchie
    - řešení: vše zachytávající funkce – funguje pro hierarchii se společným předkem Visit(FacultyMember&);

VISITOR – PROCHÁZENÍ STRUKTURY OBJEKTŮ

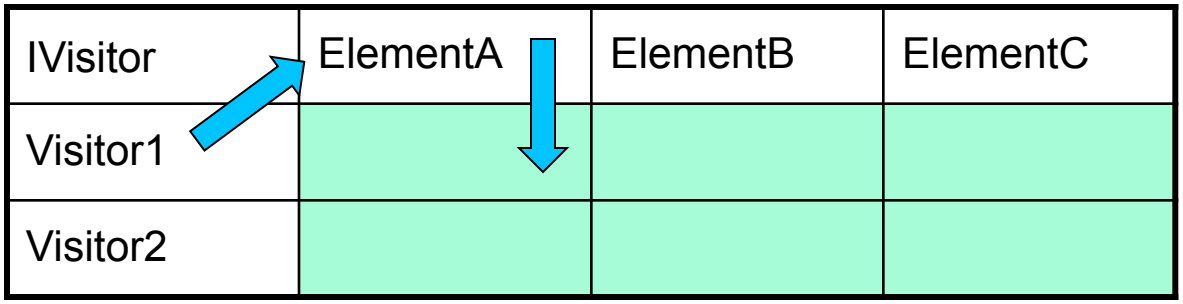
- **Kdo je zodpovědný za procházení struktury objektů?**
  - klient
    - pštrosí přístup (více práce pro klienta)
    - klient má možnost plně řídit, koho Visitor navštíví
  - struktura
    - rekurzivní volání metody `accept` na potomky
    - při použití vzoru Composite
  - Iterator
    - nelze použít, pokud objekty nemají společného předka
    - kód na procházení je na jediném místě
  - Visitor
    - elementy struktury nemusí mít společného předka
    - komplexnější algoritmy průchodu strukturou závislé na výsledcích operací nad prvky struktury
    - duplikace kódu na procházení v každém Visitoru

VISITOR – SINGLE VS. DOUBLE DISPATCH

- **single dispatch**
  - obslužná operace je vybrána na základě typu požadavku a příjemce
    - odpovídá přímému zavolání virtuální metody na objektu (příjemci)
    - nevyhovuje principu volání Visitoru
    - Vektor funkcí

FunctionX	ElementA	ElementB	ElementC

- **double dispatch**
  - obslužná operace je vybrána na základě typu požadavku a **dvou** příjemců
    - odpovídá volání metody `accept(IVisitor)` na objektu, který má být navštíven
    - dělají se dvě vyhledání (dispatch)
      - 1. dispatch: při volání accept – dynamický výběr typu operace (volání s pozdní vazbou)
      - 2. dispatch: při volání visit (uvnitř accept) – vybírá operaci staticky (známe při překladu)
    - Matice funkcí



VISITOR - PŘÍKLAD

```
class DocElement {
public:
  virtual ~DocElement();

  virtual unsigned int GetCharCount() = 0;
  virtual unsigned int GetWordCount() = 0;
  ...

  virtual void Accept(DocElementVisitor&) = 0;
protected:
    DocElement();
};
```

abstraktní předek

```
class DocElementVisitor {
public:
  virtual ~DocElementVisitor();

  virtual void Visit(DocElement&) = 0;
  virtual void Visit(Page&) = 0;
  virtual void Visit(Paragraph&) = 0;

  ...
protected:
    DocElementVisitor();
};
```

vše zachytávající funkce

VISITOR - PŘÍKLAD

```
class Paragraph: public DocElement {
public:
  ...
  virtual void Accept(DocElementVisitor& v)
  {
     v.Visit( *this );
  }
  ...
};
```

definice `Accept`
na jednoduchém objektu

```
class Page: public DocElement {
public:
  ...
  virtual void Accept(DocElementVisitor& v)
  {
     ListIterator i = list.begin();
     for( ; i != list.end(); ++i ) {
          i->Accept( v );
     }
     v.Visit( *this );
  }
private:
  List< DocElement* > list;
};
```

definice `Accept`
na složeném objektu

VISITOR - PŘÍKLAD

```
class StatisticVisitor : public DocElementVisitor {
public:
  StatisticVisitor();

  virtual void Visit(DocElement& e) {
     _ASSERT( !"Forget on Accept" );
  }

  virtual void Visit(Page& e){
    ++pageCount_;
  }

  virtual void Visit(Paragraph& e) {
      charCount_ += e.GetCharCount();
     wordCount_ += e.GetWordCount();
  }

  ...
private:
  unsigned int pageCount_;
  unsigned int charCount_;
  unsgined int wordCount_;
};
```

VISITOR - PŘÍKLAD

```
...
//doc – objekt pro celý dokument

Page* page = new Page();
doc->Add( page );

Paragraph* para1 = new Paragraph();
Paragraph* para2 = new Paragraph();
page->Add( para1 );
page->Add( para2 );

...

StatisticVisitor statVisitor;

doc->Accept( statisticVisitor );

cout << "Count of pages: " << statVisitor.GetPagesCount() << endl;
cout << "Count of words: " << statVisitor.GetWordsCount() << endl;
cout << "Count of chars: " << statVisitor.GetCharsCount() << endl;
```

vytvoření dokumentu

spočtení statistik

výpis statistik

VISITOR – POTENCIONÁLNÍ POTÍŽE

- **Porušení zapouzdření objektů ve struktuře**
  - Visitor potřebuje přistupovat k privátním položkám objektů ve struktuře
  - řešení:
    - položky objektů ve struktuře se zpřístupní jako public
    - definujeme rozhraní (metody) pro přístup k privátním položkám
      - obecné rozhraní se špatně navrhuje
      - nemusí vyhovovat všem potencionálním Visitorům
    - použijeme mechanismus reflexe
      - jazyk jej musí podporovat (C#, Java, …) a většinou nebývá rychlý

- **Použití Visitoru s již existující strukturou**
  - v již existující struktuře chybí metoda `accept` – nemůžeme použít double dispatch
  - řešení:
    - single dispatch ☹
    - použijeme Decorator a všechny třídy ve struktuře rozšíříme o metodu `accept`
      - pracné
    - použijeme reflexi, abychom nalezli správnou metodu visit podle typu cíle
      - podpora jazyka, pomalé

VISITOR – SOUVISEJÍCÍ NV

- **Známé použití**
  - operace na kolekcích
    - Funktor – jednodušší varianta Visitoru
  - operace na stromových strukturách
    - práce s jednoduchými jazyky (s gramatickými stromy)
    - XML - operace nad DOM reprezentací
  - operace GUI

- **Související NV**
  - Composite
    - operace aplikovatelné na objekty struktury jsou zapouzdřeny do Visitorů
  - Interpreter
    - na interpretaci se může použít Visitor
  - Command
    - Command může používat (případně sám být) Visitor
  - …