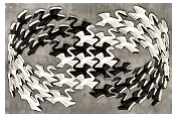


Visitor

Oh, It seems we
have a **visitor**.



The IT Crowd 1x01 – Yesterday's Jam



Visitor

■ Známý jako návštěvník

- výstižnější analogie je audit, nebo návštěva tchýně

■ Účel

- umožňuje přidat nové operace do existující hierarchie bez její modifikace

■ Motivační příklad

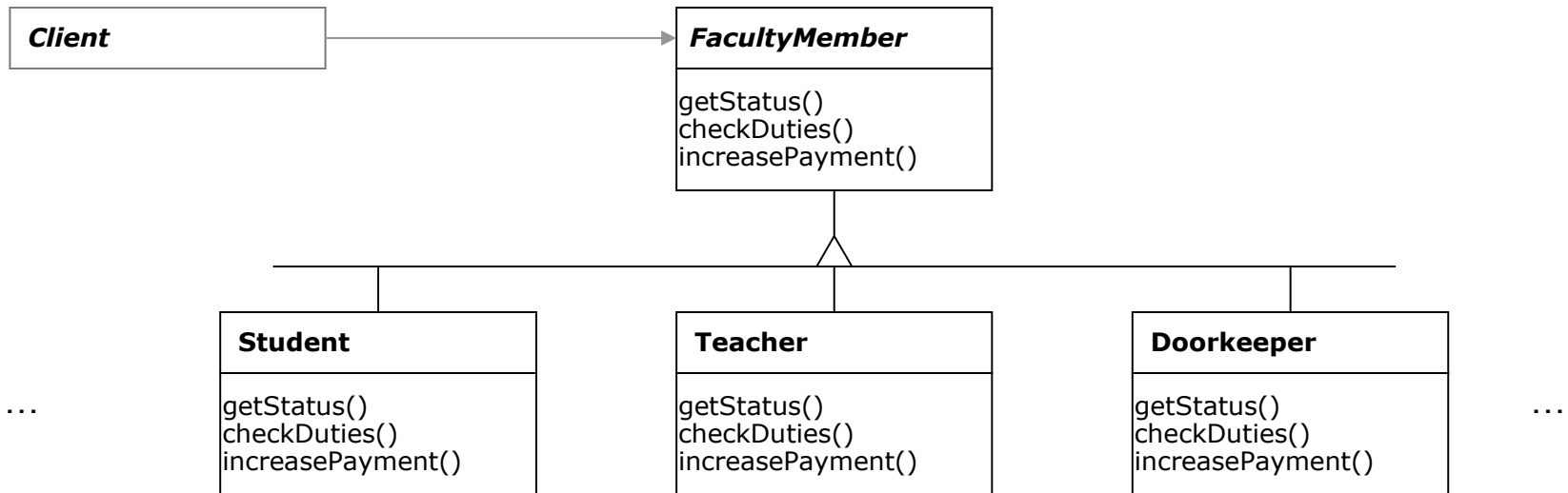
- máme složitou strukturu objektů (např. IS pro školu...)
 - vyučující, studenti, administrativní pracovníci ...
- chceme na ní provádět nejrůznější operace
 - výkazy činností, kontroly splněných povinností ...
- počet typů objektů je velký, ale neměnný
- ministerstvo/rektorát neustále vydávají nová nařízení
 - často se mění (a přidávají operace)



Visitor – motivační příklad

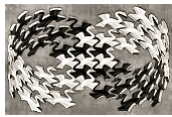
■ Klasické řešení

- umístíme operace do jednotlivých tříd



■ Nevýhody

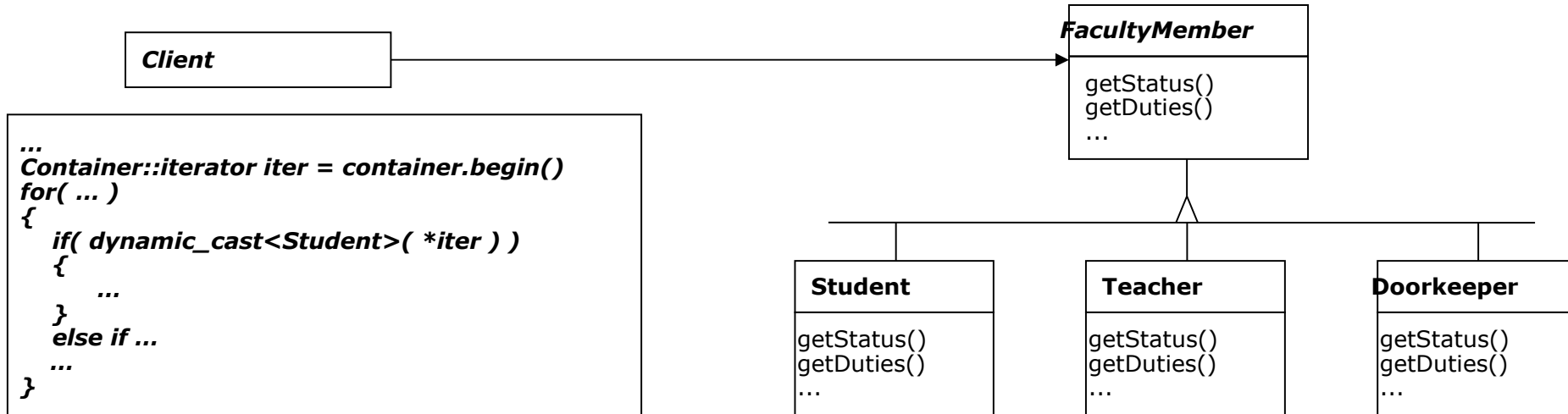
- struktura objektů je méně přehledná a hůře se udržuje
- algoritmus každé operace je rozdělen v několika třídách
- přidání operace vyžaduje změnu všech tříd



Visitor – motivační příklad

■ Druhý nápad !!!!!

- umístíme operace někam do klienta, a budeme je zpracovávat podle typů



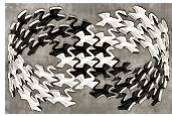
■ Výhody

- struktura objektů je „víc“ přehledná

■ Nevýhody

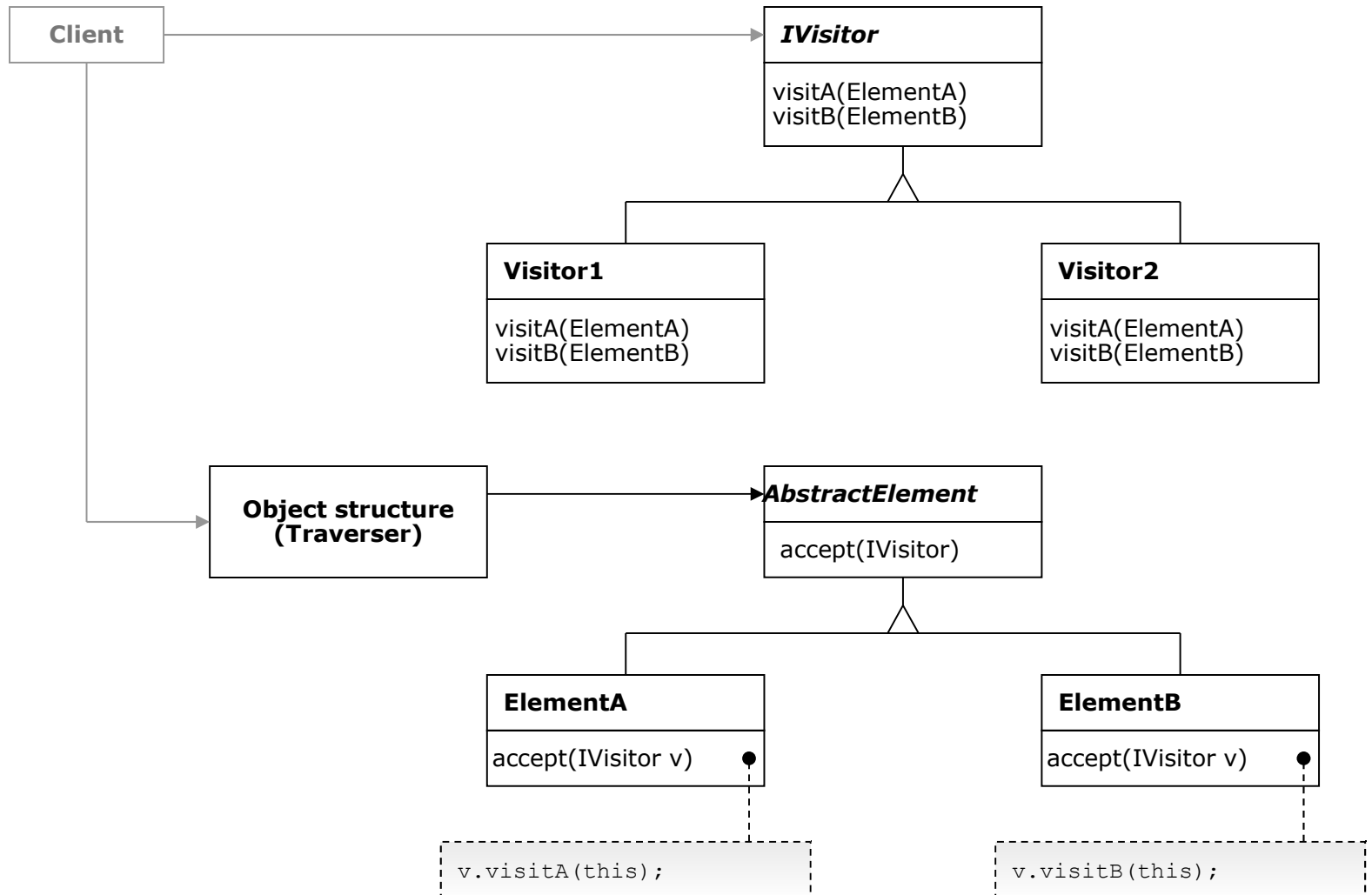
- pro každou operaci máme jeden switch blok
- zrádný a nepřehledný switch blok

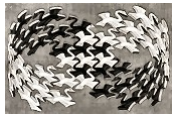
■ Lepší řešení – použijeme Visitora ...



Visitor - 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)

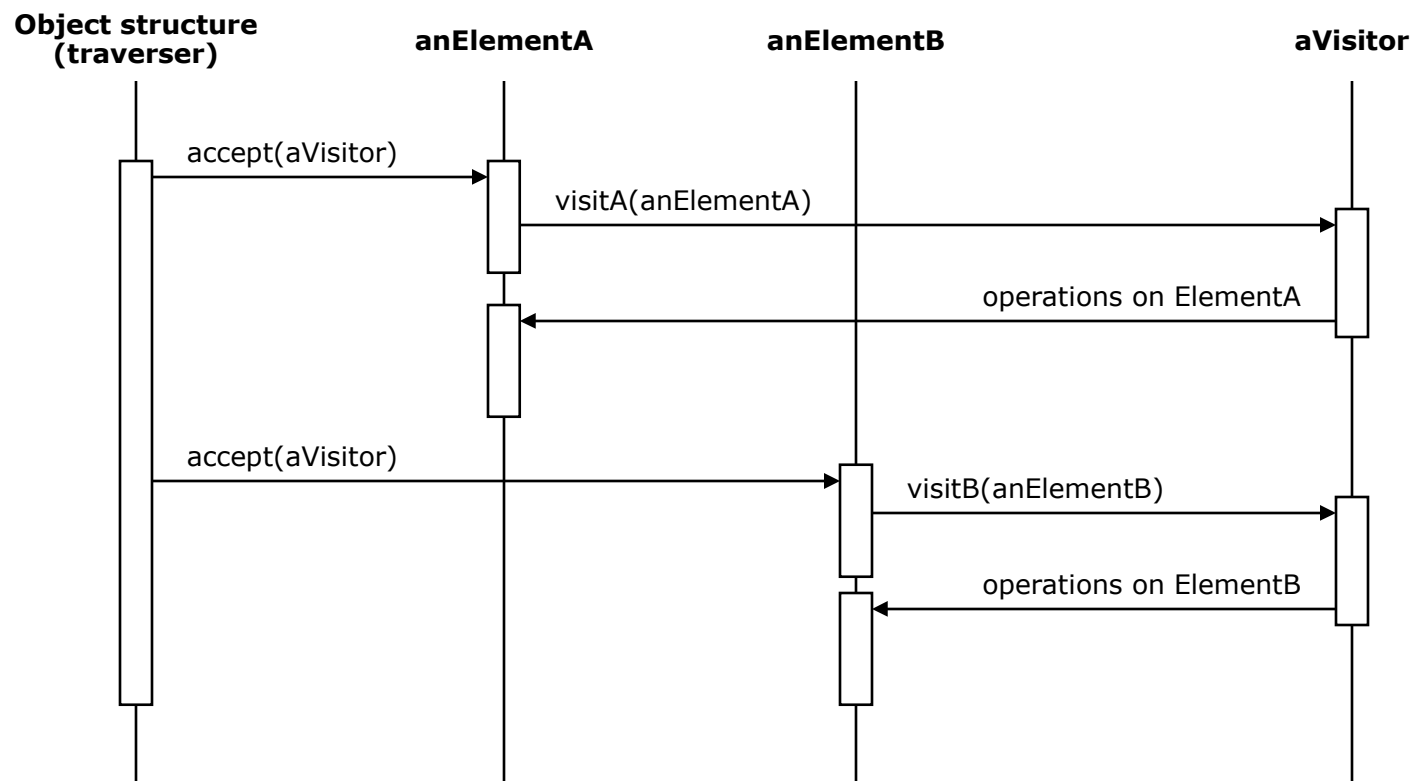
□ Object structure (traverser)

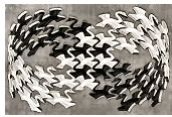
- umí procházet strukturu elementů
- na každém elementu zavolá metodu visit



Visitor - spolupráce

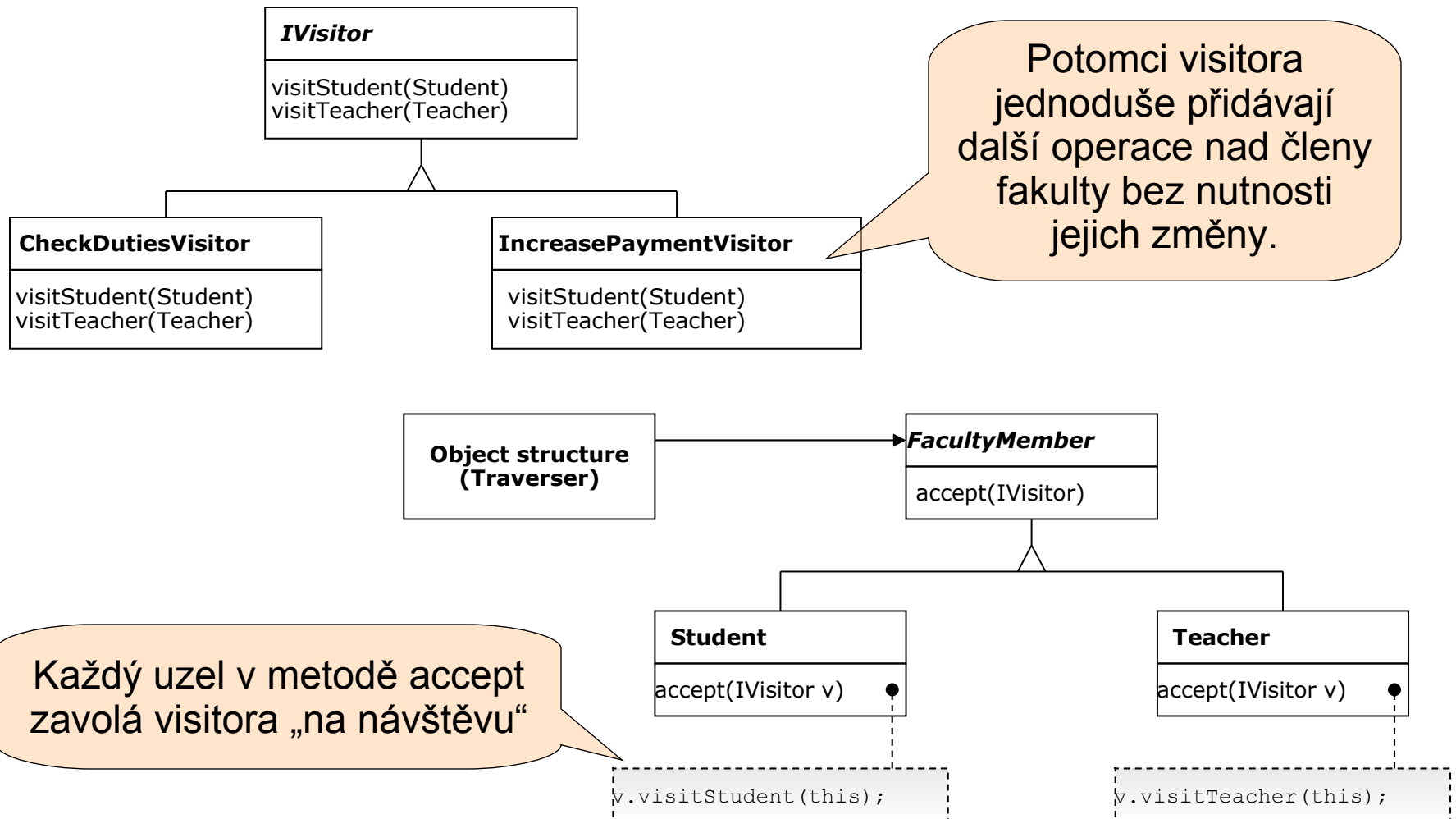
■ Spolupráce účastníků

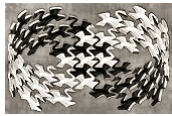




Visitor – řešení motivačního příkladu

■ Řešení motivačního příkladu s použitím Visitoru

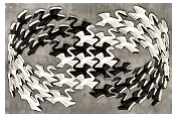




Visitor – důsledky a souvislosti

■ 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 specifikum 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`)



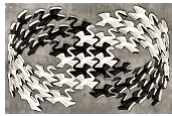
Visitor - implementace

■ 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

- pštroší přístup (více práce pro klienta)
- klient má možnost plně řídit, koho Visitor navštíví

□ struktura

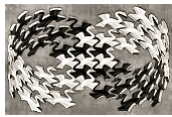
- rekurzivní volání metody `accept` na potomky
- při použití vzoru Composite

□ 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

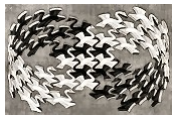
- obslužná operace je vybrána na základě typu požadavku a příjemce
 - odpovídá přímému zavolání virtuální metody na objektu (příjemci)
 - nevyhovuje principu volání Visitoru
 - Vektor funkcí

FunctionX	ElementA	ElementB	ElementC
-----------	----------	----------	----------

■ double dispatch

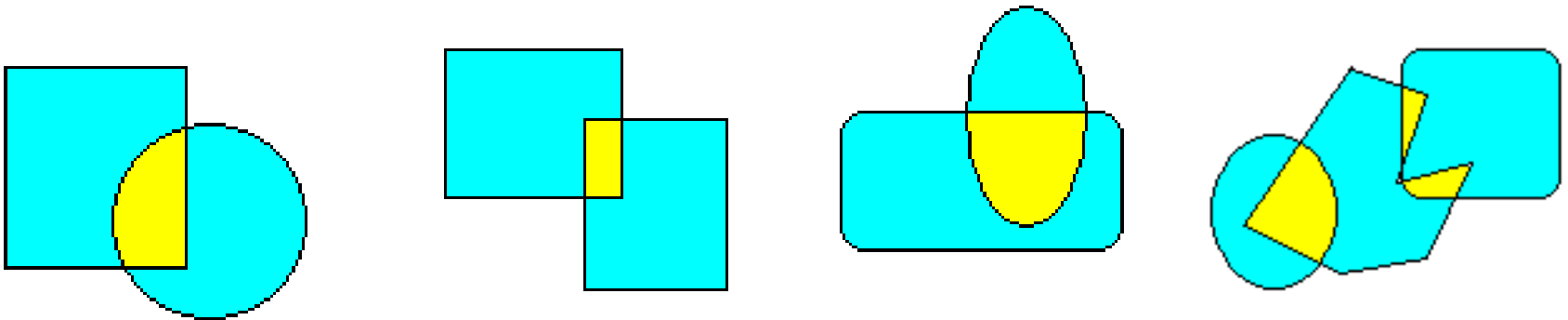
- obslužná operace je vybrána na základě typu požadavku a **dvou** příjemců
 - odpovídá volání metody `accept(IVisitor)` na objektu, který má být navštíven
 - dělají se dvě vyhledání (dispatch)
 - 1. dispatch: při volání `accept` – dynamický výběr typu operace (volání s pozdní vazbou)
 - 2. dispatch: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

	ElementA	ElementB	ElementC
Visitor1			
Visitor2			



Visitor – single vs. double dispatch

- Příklad využití double dispatch (kromě Visitoru)
 - mějme grafický editor
 - objekty jsou odvozené od abstraktní třídy Shape
 - chceme, aby se průniky objektů kreslily jinou barvou
 - některé průniky se počítají hůře, některé lépe (obdélník-obdélník, polygon-kružnice)
 - potřebujeme se rozhodnout podle dvou objektů (symetrie)



- double dispatch je speciální případ multiple dispatch
- existují jazyky, které tuhle techniku podporují přímo
 - CLOS, Perl, R, ...



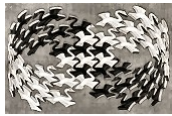
Visitor - příklad

```
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



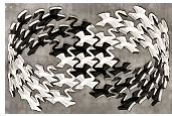
Visitor - příklad

```
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_;
    unsigned int wordCount_;
};
```



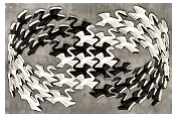

Visitor - příklad

```
...  
//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 statisticVisitor;  
  
doc->Accept( statisticVisitor );  
  
cout << "Count of pages: " << statisticVisitor.GetPagesCount()  
      << endl;  
cout << "Count of words: " << statisticVisitor.GetWordsCount()  
      << endl;  
cout << "Count of chars: " << statisticVisitor.GetCharsCount()  
      << endl;
```

vytvoření
dokumentu

spočtení
statistik

výpis
statistik



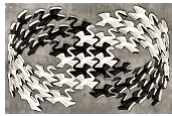
Visitor – potencionální potíže

■ 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 našli 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
- ...