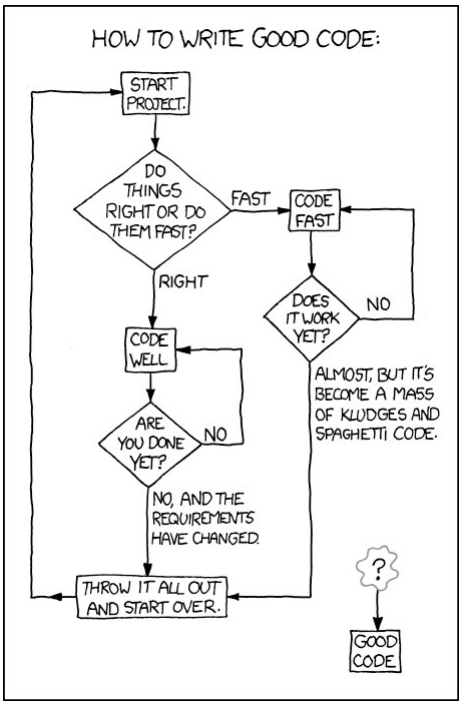


OBJECT-ORIENTED PROGRAMMING

Behavioral Design Patterns

Lecture #12

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2023/2024

Návrhové vzory správania sa

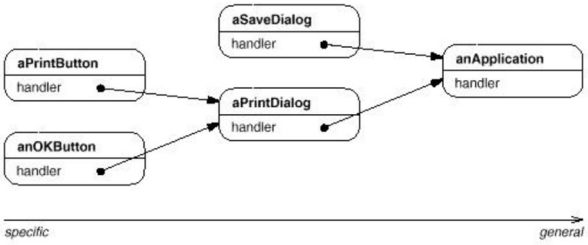
- Zaoberajú sa dynamickými interakciami medzi triedami a objektmi
- Dnes si ukážeme
  - Chain of Responsibility
  - Iterator
  - Observer
  - Strategy
  - Visitor

Chain of Responsibility

- Zámer použitia
  - Zabrániť priamej väzbe medzi odosielateľom požiadavky a príjemcom, tým že vytvoríme reťazec objektov spracúvajúcich požiadavku
- Motivácia
  - Funkcionalita kontextového pomocníka v GUI
  - Objekt, ktorý realizuje pomocníka pre konkrétny objekt (napr. tlačidlo) nie je priamo známy
  - Požiadavku necháme prejsť celým reťazcom objektov, z ktorých sa určí ten správny a vykoná príslušnú funkciu
  - Každý objekt v reťazci má jednotné rozhranie pre prijatie požiadavky a referenciu na nasledujúci objekt v reťazci

Chain of Responsibility

- Motivácia

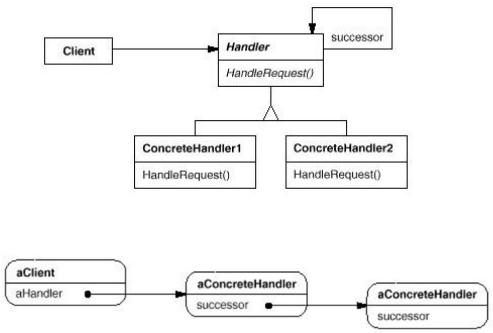


Chain of Responsibility

- Použitie
  - Pokiaľ viac ako jeden objekt môže obslúžiť požiadavku, pričom nie je známe, ktorý konkrétny objekt to je
  - Ak požiadavky vychádzajú z modelu „obslúž alebo pošli ďalej“, kde je požiadavka buď obslúžená priamo v objekte, alebo je potrebné ju poslať inému objektu
- Dôsledky
  - Zabránením priamej väzby medzi odosielateľom a príjemcom požiadavky umožňuje realizovať nezávislé riešenia
  - Odozva nie je garantovaná – požiadavka sa môže dostať na koniec reťazca bez obslúženia
  - Reťazec obslužných objektov môže byť vytváraný dynamicky

Chain of Responsibility

- Štruktúra



Chain of Responsibility (príklad 1)

- Zadanie
  - Softvérový systém pre bezpečnostné monitorovanie. Systém má viacero senzorov (dym, oheň, pohyb, atď.), ktoré prenášajú svoj stav do centrálneho počítača
  - Pre každý fyzický senzor vytvoríme jednu inštanciu objektu senzora
  - Každý objekt vie, že určitá hodnota vytvorená senzorom predstavuje nejakú akciu, ale samotná akcia je závislá aj od iných údajov ako od senzora (napr. umiestnenie senzora, údaje z iných zariadení lokalizovaných v mieste senzora, atď.)
  - Chceme škálovateľné riešenie, ktoré je možné nasadiť do ľubovoľného prostredia

Chain of Responsibility (príklad 1)

- Riešenie
  - Použijeme vzor Chain of Responsibility
  - Vytvoríme hierarchickú kompozíciu objektov senzorov, ktorá sleduje konkrétne zabezpečené prostredie
  - Definujeme objekty prostredia (stena, miestnosť, podlaha, budova) ako súčasť tejto hierarchickej kompozície
  - Poplach generovaný senzorom odošleme na vrchole tejto hierarchie a príslušný komponovaný objekt vykoná akciu

Chain of Responsibility (príklad 2)

- Základná štruktúra vzoru je, že rozhranie `Handler` má jednu metódu `handleRequest()`

```
public interface Handler {
  void handleRequest();
}
```

- Čo ak chceme obslúžiť viacero typov požiadaviek?

Chain of Responsibility (príklad 2)

- Rešenie #1
  - Rozhranie `Handler` rozšírime o viacero metód podporujúcich obsluhu rôznych typov požiadaviek

```
public interface Handler {
  void handleHelp();
  void handlePrint();
  void handleFormat();
}
```

  - V tomto prípade každá konkrétna obslužná trieda musí implementovať všetky obslužné metódy

Chain of Responsibility (príklad 2)

- Príklad konkrétnej obslužnej triedy

```
public class ConcreteHandler implements Handler {
  private Handler successor;
  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }
  @Override
  public void handleHelp() {
    // Požiadavka na pomocníka je obslúžená na tomto mieste
  }
  @Override
  public void handlePrint() {
    this.successor.handlePrint();
  }
  @Override
  public void handleFormat() {
    this.successor.handleFormat();
  }
}
```

- Ak po čase potrebujeme pridať ďalšiu obsluhu musíme meniť aj už existujúce konkrétne triedy!

Chain of Responsibility (príklad 2)

- Riešenie #2
  - Iné riešenie je mať pre každý typ požiadavky osobitné rozhranie

```
public interface HelpHandler {
  void handleHelp();
}
public interface PrintHandler {
  void handlePrint();
}
public interface FormatHandler {
  void handleFormat();
}
```

  - Teraz konkrétna trieda môže implementovať jeden alebo viac rozhraní podľa potreby a musí mať objekt nasledujúceho objektu v reťazci pre každý typ požiadavky

Chain of Responsibility (príklad 2)

```
public class ConcreteHandler implements HelpHandler, PrintHandler, FormatHandler {

  private HelpHandler helpSuccessor;
  private PrintHandler printSuccessor;
  private FormatHandler formatSuccessor;

  public ConcreteHandler(HelpHandler helpSuccessor, PrintHandler printSuccessor,
   FormatHandler formatSuccessor) {
    this.helpSuccessor = helpSuccessor;
    this.printSuccessor = printSuccessor;
    this.formatSuccessor = formatSuccessor;
  }

  @Override
  public void handleHelp() {
    // Požiadavka na pomocníka je obslúžená na tomto mieste
  }
  @Override
  public void handlePrint() { this.printSuccessor.handlePrint(); }
  @Override
  public void handleFormat() { this.formatSuccessor.handleFormat(); }
}
```

Chain of Responsibility (príklad 2)

- Riešenie #3
  - Ďalšie možné riešenie je mať iba jednu metódu rozhrania obsluhy, ktorá ako parameter prijíma typ obsluhovanej požiadavky

```
public interface Handler {
  void handleRequest(String request);
}
```


Chain of Responsibility (príklad 2)

- Konkrétna obslužná trieda vyzerá nasledovne

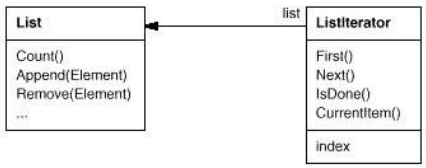
```
public class ConcreteHandler implements Handler {

  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  @Override
  public void handleRequest(String request) {
    if (request.equals("Help")) {
      // Požiadavka na pomocníka je obslúžená na tomto mieste
    }
    else {
      // Pošli ďalej
      this.successor.handle(request);
    }
  }
}
```

Iterator

- Zámer použitia
  - Poskytnutie sekvenčného prístupu (kurzora) k agregovaným objektom (kolekcia, zoznam) bez znalosti ich reprezentácie
  - Tiež nazývaný Cursor
- Motivácia
  - Agregovaný objekt napr. zoznam by mal poskytovať spôsob prístupu k jednotlivým obsiahnutým prvkom bez toho, aby sprístupnil svoju štruktúru
  - Mal by poskytovať rôzne metódy prístupu
  - Mal by poskytovať viacnásobný prístup pre paralelné spracovanie
  - Nechceme definovať tieto funkcie priamo v agregovanom objekte

Iterator

- Motivačný príklad #1
  - Zoznam s kurzorom

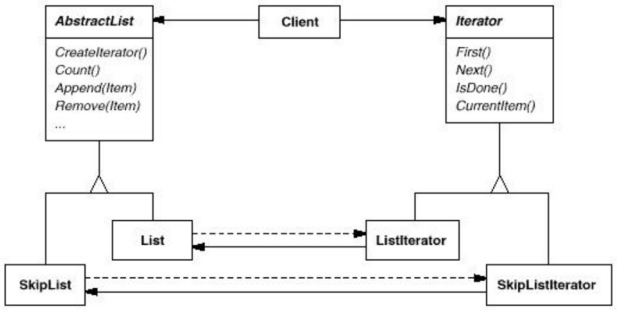


  - Typický kód klienta

```
...
List list = new List();
...
ListIterator iterator = new ListIterator(list);
iterator.First();
while (!iterator.IsDone()) {
  Object item = iterator.CurrentItem();
  // Kód pre prácu s vybraným prvkom
  iterator.Next();
}
...
```

Iterator

- Motivačný príklad #2
  - Polymorfický kurzor



Iterator

- Typický kód klienta

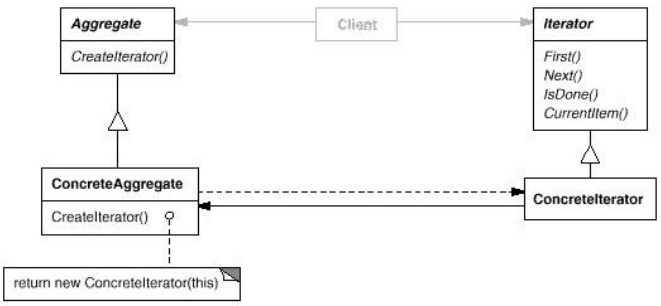
```
List list = new List();
SkipList skipList = new SkipList();
Iterator listIterator = list.CreateIterator();
Iterator skipListIterator = skipList.CreateIterator();
handleList(listIterator);
handleList(skipListIterator);
...
public void handleList(Iterator iterator) {
  iterator.First();
  while (!iterator.IsDone()) {
    Object item = iterator.CurrentItem();
    // Kód pre prácu s vybraným prvkom
    iterator.Next();
  }
}
```

Iterator

- Použitie
  - Prístup k prvkom agregovaného objektu bez toho, aby sme použili jeho skutočnú štruktúru
  - Potreba viacnásobného (paralelného) prístupu k prvkom agregovaného objektu
  - Poskytnúť jednotné rozhranie pre prístup k prvkom (tzn. polymorfický kurzor)

Iterator

- Štruktúra



Iterator

- Zúčastnené prvky
  - `Iterator` – definuje rozhranie pre sekvenčný prístup k prvkom
  - `ConcreteIterator` – implementuje rozhranie `Iterator`, udržiava stav prechodu agregovaným objektom
  - `Aggregate` – definuje rozhranie pre vytváranie objektu kurzora (použitie Factory Method)
  - `ConcreteAggregate` – implementuje rozhranie `Aggregate` a vytvára objekt konkrétneho kurzora

Iterator

- Výhody
  - Zjednodušuje rozhranie agregovaného objektu, tým že neobsahuje metódy pre prístup k jeho prvkom
  - Podporuje viacnásobný prístup
  - Podporuje rôzne varianty kurzorov

Iterator

- Implementácia
  - Kto kontroluje prechod kurzora?
    - Klient – viac flexibilné, nazývame tiež ako „externý kurzor“
    - Samotný kurzor – nazývame „interný kurzor“
  - Kto definuje algoritmus prechodu prvkami?
    - Kurzor – všeobecnejšie, jednoduchšie ak chceme mať rôzne varianty kurzorov
    - Agregovaný objekt – kurzor si uchováva iba stav prechodu prvkami
  - Môže byť agregovaný objekt zmenený počas používania kurzora?
    - Ak áno, nazývame ho tiež „robustný kurzor“
  - Implementácia ďalších operácií ako napr. `Previous()`?

Iterator (príklady)

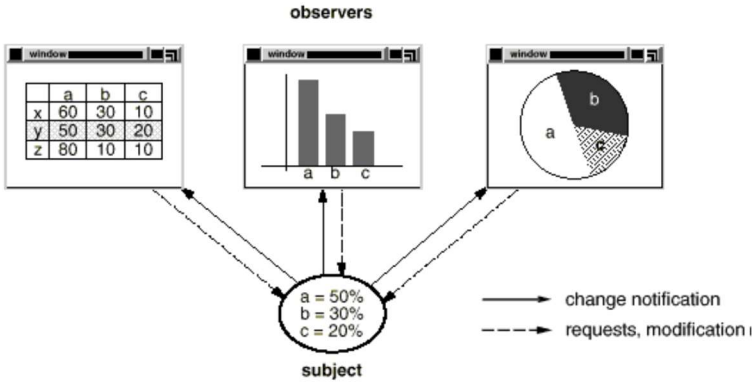
- Príklady – pozri Java
  - `java.util.Enumeration`
  - `java.util.Hashtable`
  - `java.util.Collection`
  - `java.util.Iterator`
  - `java.util.LinkedList`
  - `java.util.List`
  - `java.util.ListIterator`
  - `java.util.Vector`
  - ...

Observer

- Zámer použitia
  - Definícia závislosti 1:N medzi objektmi, ak sa zmení stav objektu, všetky závisle objekty sú o tom informované
- Motivácia
  - Potreba konzistencie medzi závislými objektmi, bez potreby úzkej väzby medzi nimi

Observer

- Motivácia

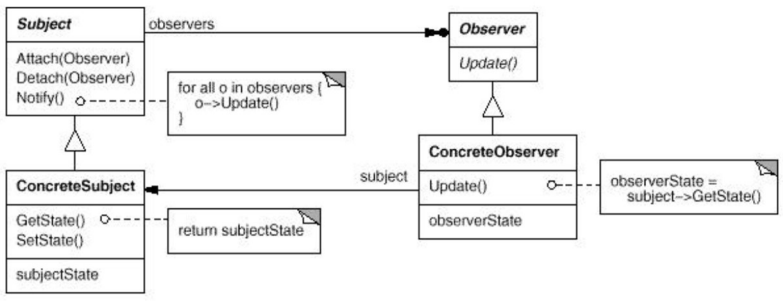


Observer

- Použitie
  - Flexibilná implementácia závislosti medzi objektmi na ich stave
  - Keď zmena jedného objektu má spôsobiť zmenu iného objektu
  - Keď objekt má informovať iné objekty o svojom stave, bez toho aby tieto objekty poznal

Observer

- Štruktúra

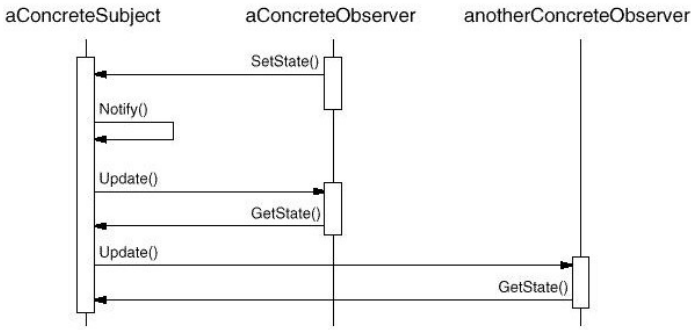


Observer

- Zúčastnené prvky
  - `Subject` – udržiava informáciu o závislých objektoch, poskytuje rozhranie pre vkladanie a vyberanie závislých objektov
  - `Observer` – rozhranie pre implementáciu upozornenia o zmene stavu
  - `ConcreteSubject` – konkrétna trieda sledovaného objektu, implementuje `Subject` a uchováva referencie na `ConcreteObservers`, ktoré upozorňuje na zmenu cez rozhranie `Observer`
  - `ConcreteObserver` – konkrétna trieda sledujúceho (závislého) objektu, jeho stav je konzistentný so stavom sledovaného objektu, implementuje `Observer` a vykonáva zmenu svojho stavu po vykonaní upozornenia zo strany `ConcreteSubject`

Observer

- Spolupráca



Observer

- Výhody
  - Minimálna väzba medzi sledovaným a závislým objektom
    - Možnosť rozširovania sledovaného objektu bez nutnosti zmeny závislých objektov
    - Pridávanie závislých objektov bez nutnosti zmeny sledovaného objektu
    - Sledovaný objekt pozná iba zoznam závislých objektov (nie konkrétne objekty) a ich rozhranie pre vykonanie upozornenia
    - Sledovaný objekt a závislé objekty môžu patriť do rozdielnych abstrakcií
  - Podpora rozosielania udalostí
    - Sledovaný objekt rozošle upozornenie (udalosť) registrovaným závislým objektom
    - Závislé objekty sa môžu ľubovoľne registrovať a odhlasovať v sledovanom objekte

Observer

- Nevýhody
  - Možnosť kaskádového upozorňovania objektov
    - Závislé objekty o sebe spravidla nevedia a vykonávanie ich akcií nemusí byť konzistentné s inými závislými objektmi
  - Jednoduché rozhranie upozorňovania môže komplikovať závislým objektom rozlišovanie, ktorý sledovaný objekt sa vlastne zmenil

Observer

- Implementácia
  - Ako udržuje sledovaný objekt množinu závislých objektov?
    - Pole, spojkový zoznam, atď.
  - Čo ak závislý objekt chce sledovať viac subjektov?
    - Sledovaný objekt musí identifikovať samého seba pri upozornení závislého objektu
  - Kto vykonáva zmenu stavu závislého objektu?
    - Sledovaný objekt pri každom upozornení
    - Samotný závislý objekt napr. pri zmene viacerých stavov
    - Iný objekt mimo danej štruktúry
  - Predtým než sledovaný objekt vykoná upozornenie **musí** vykonať zmenu svojho stavu
  - Koľko informácií má sledovaný objekt odoslať závislým objektom pri upozornení?
    - **Push architektúra** – pošli všetky potrebné informácie
    - **Pull architektúry** – iba identifikácia, sledovaný objekt si vyžiada (vytiahne) neskôr čo potrebuje

Observer

- Implementácia
  - Môže sa závislý objekt registrovať iba pre špecifické udalosti?
    - Ak áno, ide o model „zverejni – zaregistruj“
  - Môže byť závislý objekt zároveň sledovaným objektom?
    - V princípe áno
  - Čo ak chce byť závislý objekt upozorňovaný iba po zmene stavu viacerých sledovaných objektov?
    - Použitie sprostredkujúceho objektu (vzor Mediator)
    - Sledovaný objekt upozorňuje iba sprostredkujúci objekt, ktorý vykoná potrebné operácie predtým než notifikuje samotné závislé objekty

Observer (príklad 1)

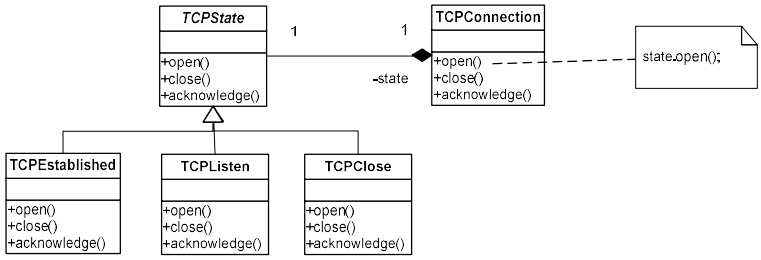
- Typické použitie
  - Softvérová architektúra Model-View-Controller (MVC)
    - Model – spracovávané údaje aplikáciou (sledované objekty)
    - View – prezentačná logika aplikácie, ktorá sleduje zmenu údajov a prezentuje výsledky pre používateľa
    - Controller – biznis logika aplikácie, ktorá spôsobuje zmenu stavu objektov
  - MVC je základom pre OO rámce webových aj desktopových aplikácií (Swing, Struts, JavaServer Faces)

Observer (príklad 2)

- Všeobecná implementácia v Jave
  - `java.util.Observer`
  - `java.util.Observable`

Strategy

- Zámer použitia
  - Definuje skupinu podobných algoritmov (funkcií), ktoré sú spoločne zapuzdrené a je možné ich zamieňať
  - Tento vzor umožňuje variovať použitie týchto algoritmov klientmi
- Motivácia

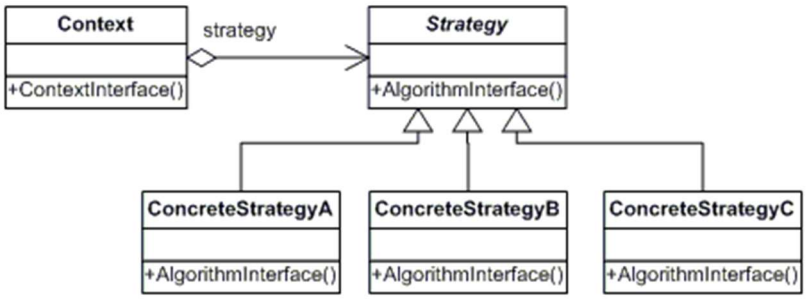


Strategy

- Použitie
  - Príbuzné triedy sa líšia iba svojim správaním
  - Potrebujeme rôzne varianty nejakého algoritmu
  - Algoritmus používa údaje, o ktorých klient nemá vedieť (Strategy je možné použiť, ak chceme skryť zložité špecifické dátové štruktúry algoritmu)
  - Trieda definuje veľa funkcií pričom ich vykonávanie je podmienené
  - Namiesto definície viacerých podmienených operácií je lepšie každú takúto funkciu definovať v osobitnej triede

Strategy

- Štruktúra

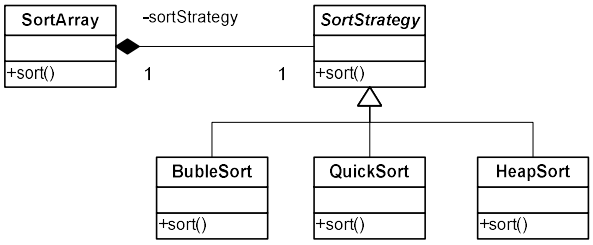


Strategy

- Výhody
  - Poskytuje alternatívu k dedeniu, kde `Context` môže obsahovať rôzne variácie algoritmov alebo správania
  - Eliminuje veľké podmienené funkcie
  - Poskytuje možnosť výberu implementácie pre tú istú funkciu

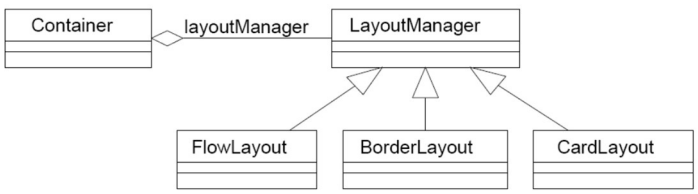
Strategy (príklad 1)

- Trieda chce počas vykonávania rozhodnúť, ktorý algoritmus triedenia poľa bude používať
- Riešenie je zapuzdrenie každého algoritmu do samostatnej triedy pomocou vzoru Strategy



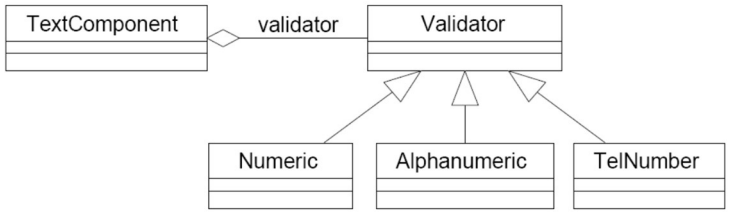
Strategy (príklad 2)

- GUI objekt skladajúci sa z iných GUI komponentov chce rozhodnúť akú stratégiu použiť pre rozloženie na obrazovke
- Riešením je zapuzdrenie rôznych stratégii rozloženia do samostatnej triedy pomocou vzoru Strategy



Strategy (príklad 3)

- GUI komponent chce rozhodnúť ako overiť správnosť zadaných vstupných údajov od používateľa (číselný údaj, text, tel. číslo)
- Riešením je zapuzdrenie rôznych stratégií overenia údajov do jedného objektu pomocou vzoru Strategy

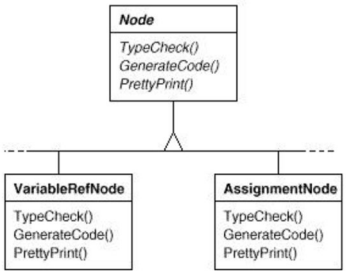


Visitor

- Zámer použitia
  - Reprezentácia operácie, ktorá má byť vykonaná nad každým prvkom nejakej štruktúry, umožňuje rozširovať operácie bez toho, aby sa zasahovalo do štruktúry prvkov
- Motivácia
  - Kompilátor, ktorý číta zdrojový text programu vytvára z neho abstraktný syntaktický strom, tento strom ma rôzne uzly (priradenie, premenná, výraz)
  - Operácie, ktoré nad uzlami chceme vykonať
    - Over či všetky použité premenné sú definované
    - Over inicializáciu premenných pred ich prvým použitím
    - Over typ premennej (výrazu)
    - Generuj (strojový) kód

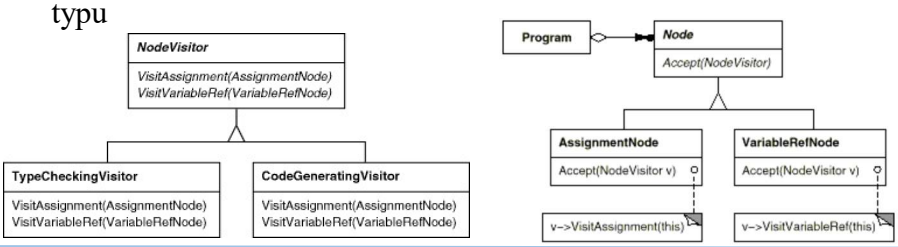
Visitor

- Motivácia
  - Každá z týchto operácií je inak vykonávaná na rôznych uzloch
  - Jeden z týchto spôsobov je vytvárať závislú štruktúru uzlov (dedením)



Visitor

- Motivácia
  - Problémy
    - Pridanie novej operácie vyžaduje zmenu celej štruktúry uzlov
    - Neprehľadné definovať operácie priamo v každom uzle
  - Iný prístup je definovať všetky operácie v osobitnom objekte nazývanom „návštevník“, ktorý bude navštevovať každý uzol štruktúry a uzol pomocou neho vykoná dané operácie podľa svojho typu

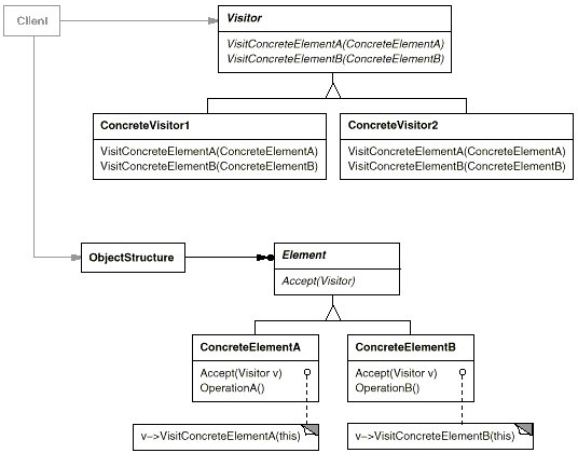


Visitor

- Použitie
  - Ak potrebujeme vykonať veľa rôznych a nezávislých operácií nad každým prvkom nejakej štruktúry objektov
  - Ak sa štruktúra objektov nemení, ale menia sa operácie vykonávané nad prvkami štruktúry objektov (inak je lepšie definovať operácie priamo s objektmi)
  - Ak štruktúra objektov obsahuje množstvo tried s rozdielnym rozhraním a vykonanie operácie je závislé na samotnom type každého prvku štruktúry objektu

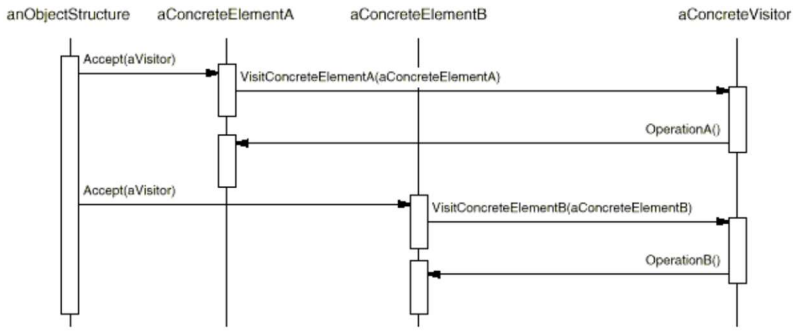
Visitor

- Štruktúra



Visitor

- Spolupráca prvkov



Visitor

- Výhody
  - Pridanie novej operácie je jednoduché
  - Súvisiace operácie nie sú rozložené v rôznych triedach sú lokalizované v jednom objekte, nesúvisiace sú rozložené do odvodených tried
  - Navštevujúci objekt môže udržiavať stav vykonávania operácií pri prechode celou štruktúrou objektov
- Nevýhody
  - Pridanie novej `ConcreteElement` je náročné, vyžaduje novú abstraktnú operáciu vo `Visitor` a implementáciu `ConcreteVisitor`
  - `ConcreteElement` musí mať veľmi silné rozhranie, niekedy je nutné porušiť zapuzdrenie, aby sme získali vnútorný stav objektu

Visitor (príklad)

- Spojenie vzorov Composite a Visitor
- Definícia všeobecného kompozitného objektu

```
public abstract class Component {
  private String name;
  public Component(String name) { this.name = name; }
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public abstract double getPrice();
  public abstract void accept(ComponentVisitor v);
}
```


Visitor (príklad)

- Definícia konkrétneho prvku v kompozite

```
public class Widget extends Component {
  private double price;
  public Widget(String name, double price) {
    super(name);
    this.price = price;
  }
  public void setPrice(double price) { this.price = price; }
  @Override
  public double getPrice() { return this.price; }
  @Override
  public void accept(ComponentVisitor v) { v.visit(this); }
}
```

Visitor (príklad)

- Definícia konkrétneho kompozitu

```
public class WidgetAssembly extends Component {
  private List<Component> components;
  public WidgetAssembly(String name) {
    super(name);
    this.components = new LinkedList<>();
  }
  public void addComponent(Component c) {
    this.components.add(c);
  }
  public void removeComponent(Component c) {
    this.components.remove(c);
  }
  @Override
  public double getPrice() {
    double totalPrice = 0.0;
    for (Component c : this.components)
      totalPrice += c.getPrice();
    return totalPrice;
  }
  @Override
  public void accept(ComponentVisitor v) { v.visit(this); }
}
```


Visitor (príklad)

- Rozhranie navštevujúceho objektu

```
public interface ComponentVisitor {
  void visit(Widget w);
  void visit(WidgetAssembly wa);
}
```

- Ak sa zmení štruktúra kompozitu, musíme pridať aj novú operáciu `visit()`

Visitor (príklad)

- Implementácia jednoduchej operácie navštevujúceho objektu (iba vypíše, ktorý prvok navštívil)

```
public class SimpleVisitor implements ComponentVisitor {
  @Override
  public void visit(Widget w) {
    System.out.println("Navštevujem Widget");
  }
  @Override
  public void visit(WidgetAssembly wa) {
    System.out.println("Navštevujem WidgetAssembly");
  }
}
```


Visitor (príklad)

- Iný navštevujúci objekt porovnávajúci ceny prvkov

```
public class PriceVisitor implements ComponentVisitor {
  private double maxPrice;
  public PriceVisitor(double maxPrice) { this.maxPrice = maxPrice; }
  @Override
  public void visit(Widget w) {
    double price = w.getPrice();
    if (price > this.maxPrice)
      System.out.println("Nekupuj! Widget cena " + price + " presahuje maximálnu cenu (" +
 this.maxPrice + ").");
    else
      System.out.println("Kupuj! Widget cena " + price + " je menšia ako maximálna cena (" +
 this.maxPrice + ").");
  }
  @Override
  public void visit(WidgetAssembly wa) {
    double price = wa.getPrice();
    if (price > this.maxPrice)
      System.out.println("Nekupuj! WidgetAssembly cena " + price + " presahuje maximálnu cenu (" +
 this.maxPrice + ").");
    else
      System.out.println("Kupuj! WidgetAssembly cena " + price + " je menšia ako maximálna cena (" +
 this.maxPrice + ").");
  }
}
```

Visitor (príklad)

- Testovací program

```
public class VisitorTest {
  public static void main(String[] args) {
    Widget w1 = new Widget("Widget1", 10.0);
    Widget w2 = new Widget("Widget2", 20.0);
    WidgetAssembly wa = new WidgetAssembly("Zostava");
    wa.addComponent(w1);
    wa.addComponent(w2);
    SimpleVisitor sv = new SimpleVisitor();
    w1.accept(sv);
    w2.accept(sv);
    wa.accept(sv);
    PriceVisitor pv = new PriceVisitor(25.0);
    w1.accept(pv);
    w2.accept(pv);
    wa.accept(pv);
  }
}
```


Visitor (príklad)

- Výstup programu

```
Navštevujem Widget
Navštevujem Widget
Navštevujem WidgetAssembly
Kupuj! Widget cena 10.0 je menšia ako maximálna cena (25.0).
Kupuj! Widget cena 20.0 je menšia ako maximálna cena (25.0).
Nekupuj! WidgetAssembly cena 30.0 presahuje maximálnu cenu (25.0).
```