Komponentový a kompozičný prístup k aspektovo-orientovanému programovaniu

Poznámky k prednáškam z predmetu Aspektovo-orientovaný vývoj softvéru

Valentino Vranić
http://fiit.sk/~vranic/, vranic@stuba.sk
Ústav informatiky, informačných systémov a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

23. október 2017

Obsah

1 Úvod

- Veľký počet AO jazykov
- Okrem iného, preberieme AO vlastnosti jazykov JAsCo a CasearJ pre ich zaujímavé črty, ktoré AspectJ nemá
- Obidva sú programovacie jazyky všeobecného použitia a predstavujú rozšírenia Javy
- Hocijaký program v Jave (v príslušnej verzii) je platným programom v týchto jazykoch
- Aj JAsCo, aj CaesarJ sa zameriavajú na podporu modularizácie komponentov
- Pozrieme sa aj na prístup kompozičné filtre (Compose*)
- Akademické projekty bez priemyselného použitia

2 JAsCo

2.1 JAsCo

- Vyvinutý na Vrije Universiteit Brussel `http://ssel.vub.ac.be/jasco/`
- Jazyk sa zameriava na aspektovo-orientované komponenty
- Posledná aktualizácia webového sídla v roku 2006...
- Najvýznamnejšie črty:
  - oddelenie spôsobu pripojenia aspektov od špecifických bodov spájania (connectors)
  - bodové prierezy so stavom (stateful pointcuts)

2.2 Háky

- Aspektovo-orientované konštrukcie sa uvádzajú v triedach ako tzv. háky (hook)
- Hák je podobný vhniezdenej triede
- V zmysle AspectJ, hák je vlastne videnie
- Bodový prierez háku sa definuje v jeho konštruktore
- Podmieňujúca metóda `isApplicable`() umožňuje definovať kedy aspekt má byť aktívny (uvádza sa bez deklarácie typu návratovej hodnoty)

2.3 Príklad háku

- Trieda s hákom[1]

```
class MyClass {
   boolean turnedOn = false;

   hook MyHook {
      MyHook(amethod (..args)) { execute(amethod); }
      isApplicable() { return global.turnedOn; }
      around() { System.out.println("around"); }
   }
}
```

- Konštruktor zachytáva vykonávania metód – zatiaľ nevieme akých
- Metóda `isApplicable()` dovolí vykonanie háku len ak sme ho „zapli“
- `global` je referencia na inštanciu aspektu

2.4 Konektory

- Kým nie sú pripojené, aspekty neovplyvňujú program
- Pripojenie sa realizuje konektormi
- Príklad konektora (pre uvedený hák):

```
static connector MyConnector {
   MyClass.MyHook hook0 = new p.MyClass.MyHook(* *.*(*));
}
```

- V háku definujeme signatúru metód, ktoré zachytávame
- Pripomína to viazanie v Theme/UML

2.5 Bodové prierezy so stavom

- Bežné bodové prierezy porovnávajú body spájania
- Bodové prierezy so stavom (stateful pointcuts) umožňujú porovnávať postupnosti bodov spájania
- Bodové prierezy so stavom si pamätajú históriu zachytených bodov spájania
- Aspekty so stavom (stateful aspects) sú aspekty, ktoré obsahujú bodové prierezy so stavom

[1] Príklady v jazyku JAsCo sú podľa: S. Hanenberg. Aspect-Oriented Software Development. Slajdy, Universität Duisburg-Essen, 2006.

2.6 Príklad bodových prierezov so stavom

- Chceme logovať vykonávania metód zachytených bodovým prierezom `p2()`, ale len medzi vykonaním metód zachytených pomocou `p1()` a `p3()`
- Napr. logovanie aktivity používateľa medzi prihlásením a odhlásením

```
public class MyLogger {
  hook LogHook {
    LogHook(startmethod(..a1), logmethod(..a2), stopm(..a3)) {
       start>p1;
       p1: execute(startmethod) > p3||p2;
       p2: execute(logmethod) > p3||p2;
       p3: execute(stopm) > p1;
    }

    before() { System.out.println(thisJoinPoint.getName()); }
    \\ ak chceme len p2(): before p2() { . . . }
  }
}
```

- Hák uplatníme na nasledujúcu triedu:

```
public class Main {
  public static void main(String[] args) {
    Main m = new Main();
    m.x();
    m.x();
  }
  public void x() { y(); }
  public void y() { z(); }
  public void z() { zzz(); }
  public void zzz() { System.out.println("out"); }
}
```

- Príslušný konektor:

```
static connector MyLoggerConnector {
  MyLogger.LogHook hook0 = new MyLogger.LogHook(
    void mypackage.Main.x(),
    void mypackage.Main.y(),
    void mypackage.Main.z()
  );
}
```

- Výsledok:

```
x
y
z
out
x
y
z
out
```

- Ak zmeníme metódu `main()` takto:

```
public static void main(String[] args) {
  Main m = new Main();
  m.z();
  m.z();
}
```

- ... výsledok bude:

```
out
out
```

- Dôvod: nikdy sme neopustili začiatočný stav

3 CaesarJ

3.1 CaesarJ

- CaesarJ – programovací jazyk založený na Jave
- Vyvinutý na Technickej univerzite v Darmstadte `http://www.caesarj.org/`
- Prvá stabilná verzia z konca roku 2004
- Open source
- Pôvodny zámer: Independent Components with On-Demand Remodularization

3.2 Caesarove triedy

- AO rozšírenie je postavené na rozšírení pre komponenty
- Komponenty sú stelesnené v tvare špeciálnych tried označených kľúčovým slovom `cclass` – tzv. Caesarove triedy
- Oddelené hierarchie dedenia: Caesarove a obyčajné triedy nemôžu byť vo vzťahu dedenia

3.3 Virtuálne triedy

- Vhniezdené Caesarove triedy sa pri dedení prekonávajú – základný rozdiel Caesarovych tried voči obyčajným triedam v Jave
- Presnejšie, prekonávajú sa metódy týchto tried
- Tieto triedy sa v jazyku CaesarJ označujú ako virtuálne
- Vo vzťahoch s inými triedami sa použijú rozšírené verzie tried: vždy sa použije najšpeciálnejšia trieda

3.4 Príklad virtuálnych tried

- Graf pozostáva z vrcholov a hrán[2]

[2]Príklady v tejto časti podľa CaesarJ Programming Guide, `http://www.caesarj.org/index.php?pagename=ProgrammingGuide.Contents`

```
public cclass Graph {
    public cclass Edge {
        Node start, end;
    }
    public cclass UEdge extends Edge {...}
    public cclass Node {...}
}
```

- Potrebujeme graf s ohodnotenými vrcholmi a hranami

```
public cclass WeightedGraph extends Graph {
    public cclass Edge {
        float cost;
    }
    public cclass Node {
        float cost;
    }
}
```

3.5 Mechanizmy kompozície

- Tri mechanizmy
  - viazanie
  - kompozícia virtuálnych tried
  - bodové prierezy a videnia
- Virtuálne triedy a viazania sú pre CaesarJ to čo medzitypové deklarácie pre AspectJ

3.6 Viazanie

- CaesarJ umožňuje vytvorenie tzv. viazania (binding) medzi dvomi triedami
- Jedna trieda je pritom obaľováč (wrapper), druhá obaľovaná (wrappee)

3.7 Príklad viazania

- Májme nasledujúci model elektrických schém:

```
public class Wire {
   public Pin getStartPin() { ... }
   public Pin getEndPin() { ... }
   ...
}
public class Pin {
   public Component getOwner() { ... }
   public Iterator getAttachedWires() { ... }
   ...
}
public class Component {
   public int getPinCount() { ... }
   public Pin getPinByNr(int nr) { ... }
   ...
}
```

- Nad týmto modelom chceme využiť algoritmy pre prácu s grafmi

```
public cclass Graph {
   . . .
   public cclass Edge {
      public Node getStartNode { ... }
      public Node getEndNode { ... }
   }
   public cclass Node {
      public Iterator getEdges() { ... }
   }
}
```

- Za týmto účelom vytvoríme viazania zodpovedajúcich tried

```
cclass ElectricGraph extends Graph {
   cclass Edge wraps Wire {
      ...
   }
   cclass Node wraps Component {
      ...
   }
}
```

- Vhniezdené triedy v triede `ElectricGraph` musia implementovať metódy tried `Edge` a `Node` pomocou zodpovedajúcich metód tried `Wire` a `Component`

```
cclass ElectricGraph extends Graph {
   cclass Edge wraps Wire {
      public Node getStartNode {
         return Node(wrappee.getStartPin().getOwner());
      }
      ...
   }
   ...
}
```

- Kľúčove slovo `wrappee` umožňuje pristúpiť k inštancii obalenej triedy

3.8 Inštancie obaľovacích tried

- Inštancie obaľovacích tried sa nevytvárajú priamo
- Vytvárajú sa výrazom tvaru `exp.Cls(obj)`:
  - `exp` je výraz, ktorý vráti objekt triedy, v ktorej je obaľovacia trieda definovaná
  - `Cls` je obaľovacia trieda
  - `obj` je referencia, ktorej typ je podtypom Cls
- Prvé volanie vytvorí obaľovací objekt; každé ďalšie vráti ten istý objekt
- V predchádzajúcom príklade volanie

  `Node(wrappee.getStartPin().getOwner())`

  t.j.

  `this.Node(wrappee.getStartPin().getOwner())`

  vráti obaľováč prvku, ku ktorému je pripojený drôt

3.9 Kompozícia virtuálnych tried

- Virtuálne triedy možno skladať pomocou operátora &
- Kompozícia sa vyjadruje ako dedenie

```
cclass Composition extends Layer3 & Layer2 & Layer1 { }
```

- Kompozícia sa realizuje sprava doľava
- V príklade najprv `Layer2` prekonáva `Layer1`, a následne `Layer3` prekonáva kompozíciu `Layer2` & `Layer1`

3.10 Cuckoo's Egg

- Vzor Cuckoo's Egg možno v jazyku CaesarJ implementovať pomocou videní – asymetricky
- Dá sa však implementovať aj symetricky – kompozíciou virtuálnych tried[3]

```
\\ prazdna implementacia vseobecneho vajca, aby sme mali spolocny zaklad pre polymorfizmus
public cclass GeneralEgg {
   public cclass Egg { };
}

\\ normalne hniezdo s pravym vajcom
public cclass UncompromisedNest extends GeneralEgg {
   \\ prekonanie celej triedy MyEgg.Egg
   public cclass Egg {
      public void exec() {
         System.out.println("a real egg.");
      }
   }
   public cclass Nest {
      public Nest() { }
      public void exec() {
         System.out.print("The nest contains ");
         new Egg().exec();
      }
   }
   public void app() {
      new Nest().exec();
   }
}

\\ kukucie vajce je tiez druh vajca
public cclass CuckoosEgg extends GeneralEgg {
   public cclass Egg {
      public void exec() {
         System.out.println("a cuckoo's egg.");
      }
   };
}

\\ podhodenie kukucieho vajca:
\\ kompozicia kukucieho vajca a nekompromitovaneho hniezda
public cclass CuckoosNest extends CuckoosEgg & UncompromisedNest { }
```

[3] J. Bálik. Diversity of Aspect-Oriented Approaches and Aspect-Oriented Design Patterns. Diplomová práca, FIIT STU, 2010.

```
public class Main {
   public static void main(String[] s) {
      new UncompromisedNest().app();
      new CuckoosNest().app();
   }
}
```

```
The nest contains a real egg.
The nest contains a cuckoo's egg.
```

3.11 Observer

- AO reimplementácia vzoru Observer v jazyku CaesarJ[4]
- Protokol pozorovania:

```
abstract public cclass ObserverProtocol {
   abstract public cclass Subject {
      abstract public void addObserver(Observer o);
      abstract public void removeObserver(Observer o);
      abstract public void changed();
      abstract public String getState();
   }
   abstract public cclass Observer {
      abstract public void notify(Subject s);
   }
}
```

- Všeobecná implementácia protokolu pozorovania:

```
public cclass ObserverProtocolImpl extends ObserverProtocol {
   abstract public cclass Subject {
      private List observers = new LinkedList();
      public void addObserver(Observer o) {
         observers.add(o);
      }
      public void removeObserver(Observer o) {
         observers.remove(o);
      }
      public void changed() {
         Iterator it = observers.iterator();
         while (it.hasNext()) {
            ((Observer) it.next()).notify(this);
         }
      }
   }
}
```

- Pozorovanie zmien farieb – definované abstraktne v zmysle protokolu vzoru Observer:

[4] J. Brichau et al. Survey of Aspect Languages and Models. AOSD-Europe-VUB-01, 2005. http://moodle.fiit.stuba.sk/moodle/file.php/72/Literatura/AOSD-Europe_Reports/aspLang.pdf (dostupné po prihlásení do FIIT Moodle)

```
abstract public cclass ColorObserver extends ObserverProtocol {
   public cclass PointSubject extends Subject wraps Point {
      public String getState() {
         return "Point colored " + wrappee.getColor();
      }
   }
   public cclass LineSubject extends Subject wraps Line {
      public String getState() {
         return "Line colored " + wrappee.getColor();
      }
   }
   public cclass ScreenObserver extends Observer wraps Screen {
      public void notify(Subject s) {
         wrappee.display("Color changed: " + s.getState());
      }
   }
   after(Point p): (call(void Point.setColor(Color)) && target(p)) {
      PointSubject(p).changed();
   }
   after(Line l): (call(void Line.setColor(Color)) && target(l)) {
      LineSubject(l).changed();
   }
}
```

- Pozorovanie zmien farieb zabezpečíme kompozíciou virtuálnych tried:

```
public cclass ColorObserverImpl extends ObserverProtocolImpl & ColorObserver { }
```

- Príklad použitia:

```
public cclass Test {
   private deployed static final CO co = new ColorObserverImpl();
   public void test() {
      Line l = new Line();
      Point p = new Point();
      Screen s1 = new Screen();
      Screen s2 = new Screen();
      co.LineSubject(l).addObserver(co.ScreenObserver(s1));
      co.PointSubject(p).addObserver(co.ScreenObserver(s1));
      co.LineSubject(l).addObserver(co.ScreenObserver(s2));
      l.setColor(new Color("red"));
      p.setColor(new Color("blue"));
   }
}
```

3.12 Bodové prierezy a videnia

- Bodové prierezy a videnia sú podobné ako v AspectJ
- Uvádzajú sa v Caesorovych triedach

```
public cclass AnAspect {
    public pointcut APointcut() : .... ;
    public around() : APointcut() {
        // code to insert
    }
}
```

3.13 Rozmiestnenie aspektov

- V jazyku CaesarJ aspekty sa môžu rozmiestňovať programovo

```
...
AnAspect a = new AnAspect();
// a not activated here
deploy( a ) {
    // a is activated now
    ...
}
// a is deactivated again
...
```

- Dá sa použiť aj rozmiestnenie ako v AspectJ

```
public deployed cclass AnAtCompileTimeActivatedAspect {
   // ...pointcuts...
   // ...advices...
}
```

3.14 Bodové prierezy

- Obmedzené vzhľadom na to čo poskytuje AspectJ
- Nejestvujú abstraktné bodové prierezy
- Nie je možné prekonávanie bodových prierezov
- Videnia sú tiež nepomenované

4 Kompozičné filtre

4.1 Aspekty ako filtre

- Composition Filters – TRESE, University of Twente[5]
- Skladanie filtrov, ktoré filtrujú správy pre daný objekt
- Filtre napojíme bez zásahu do triedy objektu, nad ktorým pracujú
- Výskum od roku 1990 s implementáciami pre jazyk Sina a ComposeJ
- Posledná implementácia je jazykovo agnostická: Compose*[6]

4.2 Príklad v jazyku Sina

- Porovnanie s AspectJ[7]

[5] `http://trese.cs.utwente.nl/oldhtml/composition_filters/`

[6] L. M.J. Bergmans. Compose*: Language-Independent Aspects for .NET. University of Twente, 2004. `http://trese.cs.utwente.nl/publications/files/0325MSARD04Istanbul.pdf`

[7] V. Vranić. Towards multi-paradigm software development. Journal of Computing and Information Technology (CIT), 10(2): 133-147, 2002. `http://hrcak.srce.hr/file/69419`

```
class Point {
  int x,y;
  Point(int x, int y){...}
  void set(int x, int y){...}
  void setX(int x){...}
  void setY(int y){...}
  int getX(){...}
  int getY(){...}
}
```

```
class Line {
  int x1,y1,x2,y2;
  Line(int x1, int y1, int x2, int y2){...}
  void set(int x1, int y1, int x2, int y2){...}
  int getX1(){...}
  int getY1(){...}
  int getX2(){...}
  int getY2(){...}
  }
```

```
Point
  acc: ShowAccess;
  inputfilters
     WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
     ReadAccess: Dispatch = {getX, getY, acc.ReadAccess, inner.*};
     CreateAccess: Dispatch = {Point, acc.CreateAccess, inner.*};
     Execute: Dispatch = {true => inner.*};

Line
  acc: ShowAccess;
  inputfilters
     WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
     ReadAccess: Dispatch = {getX, getY, getX1, getY1, acc.ReadAccess, inner.*};
     CreateAccess: Dispatch = {Line, acc.CreateAccess, inner.*};
     Execute: Dispatch = {true => inner.*};
```

5 Aspektovo-orientované črty inde

- Aspektovo-orientované črty etablovaných programovacích jazykov, ktoré nie sú označené ako aspektovo-orientované:[8]
  - Traits – Scala
  - Open classes – Ruby
  - Prototypes – JavaScript
- Uvedené mechanizmy dokonca umožňujú symetrické AOP

6 Sumarizácia

- Aspektovo-orientovaná modularizácia je aplikovateľná na komponenty
- Spôsob pripojenia aspektov možno oddeliť od špecifických bodov spájania (háky a konektory v JAsCo)
- Bodové prierezy možno založiť na konfigurácii stavov (aspekty so stavom v JAsCo)
- Komponenty možno spájať na úrovni parciálnych tried, z ktorých pozostávajú (viazanie a virtuálne triedy v CeasarJ)
- Ďalšie prístupy: kompozičné filtre, ale aj aspektovo-orientované črty programovacích jazykov, ktoré nie sú označené ako aspektovo-orientované

[8] J. Bálik and V. Vranić. Symmetric Aspect-Orientation: Some Practical Consequences. In Proc. of NEMARA 2012: International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, at AOSD 2012, March 2012, Potsdam, Germany, ACM. `http://fiit.stuba.sk/~vranic/pub/`