Aspekty a ich realizácia v jazyku AspectJ

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

3. október 2016

Obsah

1 Začíname s aspektmi **1**

2 Aspekty v akcii **1**

3 Pretínajúce záležitostí **6**

4 Skryté aspekty **8**

5 Symetrické a asymetrické AOP **8**

6 Ako modelovať aspekty? **10**

7 Niektoré historické aspekty aspektov **10**

8 O jazyku AspectJ **11**

9 Model bodov spájania **12**

10 Videnia **18**

11 Kontext bodu spájania **21**

12 Medzitypové deklarácie **23**

13 Abstraktné aspekty **26**

14 Reflektívna podpora pre body spájania **27**

15 Aspekty a anotácie **28**

16 Inštanciácia aspektov **29**

17 Aspektovo-orientované návrhové vzory a idiómy **33**

18 Idiómy jazyka AspectJ **35**

19 Vzor Worker Object Creation **37**

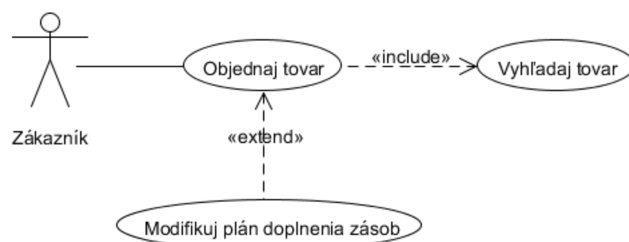
20 Vzor Wormhole **40**

21 Vzor Cuckoo′s Egg **41**

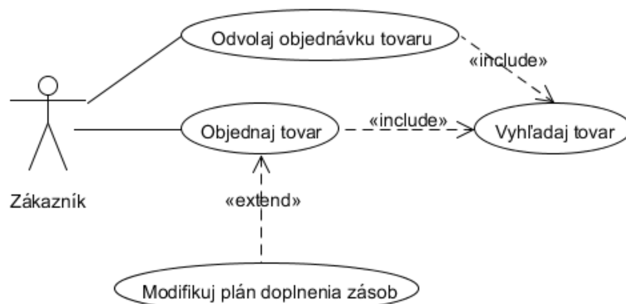
22 AO implementácia vzoru Observer **44**

23 Sumarizácia **48**

1 Začíname s aspektmi



- Vzťah include na úrovni kódu predstavuje volanie procedúry
- Ako by sme mohli zachovať vzťah extend v kóde?



- Ako by sme mohli zachovať prípady použitia na rovnakej úrovni (peer use cases) v kóde?

Prípady použitia a aspekty

- Rozširujúci prípad použitia ovplyvňuje rozširovaný prípad použitia bez jeho vedomia
- Každý prípad použitia na rovnakej úrovni prispieva tým, čo je potrebné z jeho perspektívy, do tried, ktoré spoločne realizujú prípady použitia
- Dalo by sa dosiahnuť v kóde?

2 Aspekty v akcii

Základná trieda

- Predpokladajme, že vyvíjame malý grafický systém
- Používame nasledujúcu reprezentáciu bodu:

```
public class Point {
  private int x;
  private int y;

  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public int getX() { return x; }
  public int getY() { return y; }
}
```

Pridanie monitorovania operácií nad bodmi

- Potrebujeme monitorovať operácie nad bodmi
- Musíme upraviť všetky set a get metódy

```
public class Point {
  private int x;
  private int y;

  public void setX(int x) {
    System.out.println("Moving a point.");
    this.x = x;
  }
  public void setY(int y) {
    System.out.println("Moving a point.");
    this.y = y;
  }
  public int getX() {
    System.out.println("Reading a point.");
    return x;
  }
  public int getY() {
    System.out.println("Reading a point.");
    return y; }
}
```

Dajú sa špecifikovať miesta, v ktorých má byť vykonané monitorovanie?

Čo sme dostali?

- Došlo k prepleteniu kódu: kód na monitorovanie je prepletený s kódom aplikačnej logiky
- Došlo k roztrúseniu kódu: kód na monitorovanie sa opakuje v rôznych metódach
- Čo keby sme chceli vypnúť monitorovanie?
- Mali sme použiť o jednu úroveň abstrakcie viac:

  ```
  public void setX(int x) {
    this.x = x;
    PointMonitor.print("Moving a point.");
  }
  ```

- Monitorovanie vypneme úpravou metódy `print()`
- Ale zostáva problém samotnej prítomnosti monitorovania v aplikačnej logike a potreby manuálneho pridávania tohto kódu

Čo sme vlastne chceli?

- Chceli sme *pripojiť* monitorovacie výpisy ku všetkým set a get metódam
- Vieme presne špecifikovať o aké metódy ide?
- Áno:
  1. metódy triedy `Point`, ktorých názov začína na **set**
  2. metódy triedy `Point`, ktorých názov začína na **get**
- Čo keby programovací jazyk priamo umožňoval vyjadriť, že *pred vykonaním* každej takej metódy treba podať príslušný výpis bez potreby zasahovať do metódy samotnej?
- AspectJ to umožňuje...

Monitorovanie ako aspekt

- Aspekt pre sledovanie operácií nad objektmi triedy `Point`:

```
public aspect AccessMonitoring {
  before(): execution(* Point.set*(..)) {
    System.out.println("Moving a point.");
  }
  before(): execution(* Point.get*(..)) {
    System.out.println("Reading a point.");
  }
}
```

- Vykonaním príkazov

```
Point p = new Point();
p.setX(10);
p.setY(p.getX());
```

- dostáva sa nasledujúci výstup:

```
Moving a point.
Reading a point.
Moving a point.
```

Monitorovanie ukončenia operácií

- Čo keby sme v klasickej Jave chceli monitorovať aj ukončenie operácií?
- Zdanlivo to nie je problém:

```
public void setX(int x) {
  System.out.println("Moving a point.");
  this.x = x;
  System.out.println("Moved a point.");
}
```

- Výpis je však v skutočnosti ešte v rámci vykonávania metódy
- Úplne jasné je to pri metódach, ktoré vracajú hodnotu:

```
public int getX() {
  System.out.println("Reading a point.");
  return x;
  System.out.println("Read a point."); // nevykona sa!
}
```

- Dá sa sčasti obísť blokom **try−finally**
- Pre monitorovanie ukončenia operácií by sme museli pridať výpis *za každé volanie* v klientskom kóde
- Ako by to bolo v jazyku AspectJ?
- Môžeme mať výpis skutočne po vykonaní metódy

```
public aspect AccessMonitoring {
  ...
  after(): execution(* Point.set*(..)) {
    System.out.println("Moved a point.");
  }
  after(): execution(* Point.get*(..)) {
    System.out.println("Read a point.");
  }
}
```

- Môžeme mať výpis aj za každým volaním metódy

```
public aspect AccessMonitoring {
  ...
  after(): call(* Point.set*(..)) {
    System.out.println("Moved a point.");
  }
  after(): call(* Point.get*(..)) {
    System.out.println("Read a point.");
  }
}
```

- Aký je rozdiel medzi **execution** a **call**?
- Pri **call** napr. môžeme obmedziť výpisy na volania len v určitých triedach a metódach

Sledovanie rozsahu hodnôt

```
public aspect RangeMonitoring {
   private boolean xRange(int x) {
      return x >= 0 && x <= 639;
   }
   private boolean yRange(int y) {
      return y >= 0 && y <= 399;
   }
   before(int x): call(void Point.setX(..)) && args(x) {
      if (!xRange(x))
         System.out.println("X mimo rozsahu:" + x);
   }
   before(int y): call(void Point.setY(..)) && args(y) {
      if (!yRange(y))
         System.out.println("Y mimo rozsahu:" + y);
   }
}
```

Riadenie rozsahu hodnôt

```
public aspect RangeControl {
   private boolean xRange(int x) {
      return x >= 0 && x <= 639;
   }
   private boolean yRange(int y) {
      return y >= 0 && y <= 399;
   }
   void around(int x): call(void Point.setX(..)) && args(x) {
      if (!xRange(x))
         System.out.println("X mimo rozsahu:" + x);
      else
         proceed(x);
   }

aspect RangeControl {
   void around(int x): call(void Point.setX(..)) && args(x) {
      if (x < 0)
         proceed(640 + x % 640);
      else if (x > 639)
         proceed(x % 640);
      else
         proceed(x);
   }
   ...
}
```

Body spájania

- Bódy spájania (join points) sú kľúčovým pojmom v AOVS
- Body spájania predstavujú dobre definované miesta vo vykonávaní programu
- Na týchto miestach možno ovplyvniť tok programu
- Príkladom je vykonávanie metódy, volanie metódy, prístup k atribútu (field), nastavenie atribútu apod.
- Od programovacieho jazyka závisí, ktoré body spájania budú *exponované*
- Napr. slučky v AspectJ nepredstavujú exponované body spájania

Bodové prierezy

- Fundamentálnym problémom v AOP je výber bodov spájania: ako vybrať body, ktoré potrebujeme
- Body spájania definujeme pomocou primitívnych *bodových prierezov* definovaných v samotnom programovacom jazyku
- Bodové prierezy predstavujú množiny bodov spájania
- Detailnejšie na nasledujúcich prednáškach

Statické a dynamické body spájania

- Niektoré body spájania možno zachytiť priamo v zdrojovom texte – statické body spájania
- Dynamické body spájania možno zachytiť len pri vykonávaní programu – napr. volania metód
- Aspekty potom môžu pracovať s kontextom bodu spájania – napr. ovplyvňovať hodnoty argumentov

Niektoré známe aplikácie AOP

- Podľa AOSD-Europe:
  - IBM Websphere Application Server – AspectJ
  - JBoss Application Server – JBoss AOP
  - Oracle TopLink – Spring AOP
  - Java ME – AspectJ pri vývoji mobilných aplikácií
  - Siemens SOARIAN – zdravotný IS; AspectJ a FastAOP
  - MySQL – logovanie pomocou AspectJ
- Zaujímavý je aj prehľad Ramnivasa Laddada, podľa ktorého je AOP vo fáze produktívneho využitia[1]

3 Pretínajúce záležitostí

Oddelenie záležitostí

- Oddelenie záležitostí – separation of concerns
- Pojem z článku Edsgera W. Dijkstru[2]
- Dijkstra vlastne pojednával o „inteligentnom uvažovaní“ vo všeobecnosti: vždy sa sústreďujeme len na jeden aspekt predmetu, ktorý študujeme
- Pojem však prenikol do oblasti vývoja softvéru vo význame dekompozície
- Pre úspešné zvládnutie problému musíme byť schopní ho rozložiť
- Jednotlivými záležitosťami sa môžeme zaoberať samostatne
- Tento proces sa niekedy označuje ako modularizácia

[1] R. Laddad. A Real-World Perspective of AOP. Transactions on Aspect-Oriented Software Development VIII, Springer, LNCS 6580/2011, 2011.

[2] E. W. Dijkstra. On the role of scientific thought, 1974 (EWD 447). http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF

Pretínajúce záležitosti

- Pretínajúce záležitosti – crosscutting concerns
- Nie vždy je možné modularizovať všetky záležitosti
- Kód sa stáva zapleteným (tangled) a roztrúseným (scattered)
- Deje sa to súčasne – dva pohľady na ten istý problém
- K týmto javom dochádza vo všetkých prístupoch k programovaniu založených na zovšeobecnených procedúrach (generalized procedures)
- Aspektovo-orientované programovanie umožňuje *modularizáciu pretínajúcich záležitosti*
- Pretínajúce záležitosti sa označujú ako *aspekty*

Zapletenie kódu

- Code tangling
- Rôzne záležitosti zmiešané v rámci jedného modulu
- Napr. v metódach triedy aplikačnej logiky (application/business logic) sú volania metód pre logovanie, bezpečnosť, perzistenciu apod.

Roztrúsenie kódu

- Code scattering
- Prejavuje sa dvoma spôsobmi:
  1. Jedna záležitosť sa opakuje v rôznych moduloch
     - Napr. logovanie je prítomné v metódach tried, ktoré vôbec nesúvisia
  2. Časti tej istej záležitosti sú roztrúsené v rôznych moduloch
     - Napr. autorizácia zahŕňa autentifikáciu používateľa, manažment práv, kontrolu prístupu atď.
     - Napriek tomu, že tvoria logický celok, každá z týchto časti je potrebná – a implementovaná – v inom module

Problém pretínajúcich záležitostí

- Problém pretínajúcich záležitostí bol spozorovaný predovšetkým v kóde
  - Takýto kód sa ťažko udržiava
  - Zmena v takejto pretínajúcej záležitosti často vyžaduje zásah do všetkých miest kde je použitá
  - Podobný problém je s odstránením záležitosti (napr. vypnutie logovania) alebo pridaním ďalšej záležitosti
- Prejavuje sa však aj v návrhu, aj v analýze
- Dokonca je prítomný už na úrovni požiadaviek

Doterajšie riešenia problému pretínajúcich záležitostí

- Metaobject protocol (MOP) – predovšetkým v Smalltalku
- CLOS – možnosť definovať čo sa má vykonať pred a za danou metódou
- Tieto riešenia však nie sú veľmi využívané
- Zvlášť MOP je veľmi silný nástroj, ale potrebujeme špecifikovať pretínanie na vyššej úrovni abstrakcie
- MOP je skôr mechanizmus pre zabezpečenie modularizácie pretínajúcich záležitostí

Aspektovo-orientovaný prístup

- Aspekt = pohľad
- Rôzne pohľady na tú istú vec
- Každým pohľadom sa potrebujeme zaoberať samostatne
- Aspektovo-orientované prístupy sa sústreďujú na problém pretínania
- Aspektovo-orientované programovanie (aspect-oriented programming, AOP) prináša nové jazykové konštrukcie pre to potrebné
- Aspektovo-orientovaný vývoj softvéru (aspect-oriented software development, AOSD) – komplexný prístup k problému pretínajúcich záležitostí na všetkých úrovniach

4 Skryté aspekty

Pripomína vám AOP niečo, s čím ste sa pri doterajšom programovaní stretli?

5 Symetrické a asymetrické AOP

Symetrickosť AOP

- Asymetrické aspektovo-orientované prístupy[3]
  - Rozlišuje sa medzi základnou dekompozíciou a pretínajúcimi záležitosťami (aspektmi)
  - Príkladom je AspectJ
  - Vzťah extend medzi prípadmi použitia
- Symetrické aspektovo-orientované prístupy
  - Program ako celok vzniká spájaním rozličných pohľadov (teda aspektov)
  - Príkladom je Hyper/J
  - Prípady použitia na rovnakej úrovni

[3] W. H. Harrison, H. L. Ossher, and P. L. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical report RC22685, IBM Research, 2002.

Symetrický prístup

- Monitorovanie prístupu v syntaxi podobnej jazyku Hyper/J (staršia verzia)
- Monitorovanie implementujeme ako zvláštnu triedu:

```
class PointAccessMonitoring {
  void movingPoint() {System.out.println("Moving a point."); }
  void movedPoint() {System.out.println("Moved a point."); }
}
```

- Potom *zvlášť* definujeme pravidlá spájania (composition rules):

```
namespace AccessMonitoredGraphics{
  AccessMonitoredGraphics.Point.setX :=
    Merge[Graphics.Point.setX, Monitoring.PointAccessMonitoring.movingPoint];

  AccessMonitoredGraphics.Point.setY :=
    Merge[Graphics.Point.setY, Monitoring.PointAccessMonitoring.movingPoint];
  . . .
}
```

- Hyper/J používa koncept priestoru mien (namespace) – rôzne aspekty tej istej záležitosti potom môžu byť pod rovnakým názvom
- Predpokladáme, že trieda `Point` je definovaná v priestore mien `Graphics`
- Získame novú triedu `Point` s pripojeným kódom pre monitorovanie
- Aj v symetrickom prístupe definujeme aspekty a body spájania, ku ktorým sa majú pripojiť
- Rozdiel je skôr vo vnímaní aspektov: uvažujeme ako sa veci spájajú, nie ako jednou vecou zasiahnuť do iných
- Dôležité pre aspektovo-orientovanú analýzu a návrh
- Zvlášť dôležité pre tzv. aplikačne špecifické aspekty – menej pre všeobecné aspekty ako sú monitorovanie, logovanie, synchronizácia, perzistencia, bezpečnosť apod.
- Deklarovane symetrický aspektovo-orientovaný programovací jazyk v praxi nenájdeme
- Niektoré populárne jazyky vykazujú prvky symetrickej aspektovo-orientovanej dekompozície:[4]
  - Scala – črty (traits)
  - Ruby – otvorené triedy
  - JavaScript – prototypy

[4] 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.

6 Ako modelovať aspekty?

Ako by ste modelovali v aspektovo-orientovanom vývoji softvéru?

Niektoré možnosti modelovania aspektov

- Prípady použitia – na rovnakej úrovni a vo vzťahu extend (AOSD with use cases)
- Theme – založený na kompozícii čiastkových modelov v UML s využitím parametrizácie
- Join Point Designation Diagrams (JPDD) – umžňuje grafickú špecifikáciu bodov spájania vo forme dopytov nad UML modelom

7 Niektoré historické aspekty aspektov

Prvé AO prístupy

- PARC AOP (AspectJ)
  - Aspekty ako moduly pre pretínajúce záležitosti
- Subjektovo-orientované programovanie (Hyper/J)[5]
  - Skladanie subjektov ako rôznych pohľadov na problém
- Kompozičné filtre (Sina/st)[6]
  - Skladanie filtrov (ktoré filtrujú správy pre daný objekt)
- Adaptívne programovanie (DemeterJ)[7]
  - Prispôsobivé stratégie prechádzania grafom tried

PARC AOP

- Štyri vymenované AO prístupy sú stále základom pre novovznikajúce prístupy a jazyky
- PARC AOP je najvplyvnejší z nich
- Keď sa hovorí o AOP, väčšinou sa myslí na PARC AOP
- PARC vyvinul jazyk AspectJ
- V súčasnosti desiatky AO jazykov založených predovšetkým na PARC AOP
  - napr. AspectC++, CeasarJ, AspectS (založený na Smalltalku)...
- Preto sa na ďalších prednáškach pozrieme bližšie na jazyk AspectJ

[5] `http://www.research.ibm.com/hyperspace/`
[6] `http://trese.cs.utwente.nl/composition_filters`
[7] `http://www.ccs.neu.edu/research/demeter/`

8 O jazyku AspectJ

Základné vlastnosti jazyka AspectJ

- Aspektovo-orientované rozšírenie Javy
- Rozšírenie je homogénne
  - Každý program v Jave je regulárny program v AspectJ
  - V aspektoch sa používajú obvyklé konštrukcie Javy
  - Triedy môžu obsahovať niektoré prvky aspektov
- Asymetrický prístup
  - Základná dekompozícia je vyjadrená triedami
  - Pretínajúce záležitostí sú vyjadrené zvláštnym mechanizmom – aspektmi
- Programovací jazyk všeobecného použitia

Vývoj jazyka AspectJ

- Otvorený vývoj
- Od roku 2002 na eclipse.org[8]
- Vývoj od roku 1997[9]
- Verzia 1.0 vyšla na konci roku 2001
- Súčasná verzia (1.8) je založená na Jave 8
- AspectJ Development Tools (AJDT) – podpora v Eclipse

AO konštrukcie v AspectJ

- Základnou konštrukciou je *aspekt* (aspect)
- Aspekty modifikujú vykonávanie programu v bodoch spájania
  - Tieto miesta sú určené *bodovými prierezmi* (pointcuts)
  - Modifikácie sú vyjadrené pomocou *videní* (advices)
  - Videnie sa vykonáva pred, po alebo namiesto bodu spájania
- Aspekty môžu aj dopĺňať triedy novými prvkami
  - Pomocou *medzitypových deklarácií* (inter-type declarations)
  - Do triedy je možné pridať nové prvky, ale aj závislosti dedenia
- Inštancie aspektov sa vytvárajú automaticky

[8] `http://eclipse.org/aspectj/`
[9] `http://www.parc.com/research/csl/projects/aspectj/`

Preklad v jazyku AspectJ

- *Vtkanie* (weaving) aspektov sa realizuje priamo pri preklade (prekladač `ajc`) nad bajtkódom,[10] ale možné je aj vtkanie v čase načítavania (load-time weaving)
  - V starších verziách `ajc` vtkanie sa realizovalo nad zdrojovým kódom – výsledkom bol kód v Jave (musel sa ešte preložiť `javac`om)
- Výsledok je Java bajtkód – preložený program sa vykonáva na Java VM
- Každý program v Jave je platný AspectJ program
- Rozdiel pri preklade voči `javac`u: treba uviesť všetky súbory
  - Možnosť rozhodnúť o pripojení určitého aspektu pri preklade
  - Nie je nutné mať zdrojové súbory – `ajc` môže vtkať aspekty aj do bajtkódu
  - Pri použití @AspectJ notácie, aj aspekty môžu byť pred vtkaním preložené `javac`om

9 Model bodov spájania

Predstavte si, že navrhujete aspektovo-orientovaný jazyk. Aké body spájania by sa podľa vás v ňom mali dať zachytávať?

Body spájania

- Bod spájania (join point) – dobre definovaný bod vo vykonávaní programu
  - Tieň bodu spájania (join point shadow) – konštrukcia v zdrojovom kóde, ktorá zodpovedá danému bodu spájania
- V bodoch spájania je možné vplývať na vykonávanie programu
- Exponované body spájania – body spájania, ku ktorým je v danom programovacom jazyku možné pristúpiť
  - Napr. volanie metódy je v AspectJ bod spájania
  - Bodom spájania však nie je slučka **for**
- V jazyku AspectJ možno pracovať s *kontextom bodu spájania*
  - Napr. kontext volania metódy predstavuje objekt, ktorý ju zavolal, cieľový objekt a argumenty

[10] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Proceedings of 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22–24, 2004. ACM. `http://doi.acm.org/10.1145/976270.976276`

Exponované body spájania v jazyku AspectJ

- Metódy a konštruktory
  - Volanie metódy alebo konštruktora
  - Vykonanie metódy alebo konštruktora
- Prístup k atribútom
  - Čítanie atribútu
  - Zápis atribútu
- Spracovanie výnimiek
  - Vykonanie bloku spracovania výnimky
- Inicializácia tried a objektov
  - Vykonanie bloku statickej inicializácie
  - Vykonanie inicializácie objektu
  - Vykonanie predinicializácie objektu
- Vykonanie videnia
  - Vykonanie všetkých videní v programe

Bodové prierezy

- K bodom spájania sa nepristupuje jednotlivo, ale cez bodové prierezy, ktoré ich obsahujú (quantification)
- Bodový prierez predstavuje množinu bodov spájania
- Bodové prierezy sa definujú pomocou primitívnych (elementárnych) bodových prierezov
  - Primitívne bodové prierezy sú tiež množiny bodov spájania
- Bodový prierez, ktorý zachytáva volania metód triedy `Point`, ktorých názov začína na `set` a nevracajú žiadnu hodnotu:

  ```
  call(void Point.set*(..))
  ```

- Pomocou logickej spojky *alebo* môžeme pridať aj metódy na čítanie bodov:

  ```
  call(void Point.set*(..)) || call(int Point.get*(..))
  ```

Signatúry

- Metódy, konštruktory, typy a atribúty sa v bodových prierezoch zadávajú pomocou svojich signatúr
  - Pod typom sa myslí trieda, rozhranie alebo aspekt
- Da sa zadať aj presne jeden prvok:
  - `call`(**void** `Point.setX`())
  - **within**(`Point`)
  - **set**(**private int** `Point.y`)
- Ale najčastejšie sa používajú signatúry s náhradnými znakmi

Signatúry s náhradnými znakmi

- * zamieňa názov typu, metódy alebo atribútu, alebo časť tohto názvu
  - `i* P*.get*()` – všetky metódy s názvom, ktorý začína na **get**, vo všetkých typoch, ktorých názvy začínajú na `P`, a ktoré vracajú hodnotu typu, ktorého názov začína na `i`
- .. v prípade metódy nahrádza zoznam argumentov alebo jej časť:
  - `void Point.set*(..)` – všetky metódy `void Point.set*(..)`) s hocijakým počtom argumentov
- .. v prípade balíka nahrádza podbalíky (aj priame, aj nepriame)
  - `java..*` – všetky typy v rámci balíka `java`
- + nahrádza podtypy daného typu
  - `Point+` – trieda `Point` a všetky jej podtriedy
- Ak v signatúre metódy nie je uvedený modifikátor, signatúra zahŕňa všetky modifikácie metódy
  - Modifikátory zahŕňajú modifikátory prístupu, **abstract**, **static** a **final**
- Dá sa zadať negácia modifikátora a typu vracanej hodnoty
  - **!final !public !protected !static int** `MyClass.m*()`
  - **!private !public !protected** * `MyClass.*(..)`
- Negácia sa dá použiť aj na typ návratovej hodnoty
  - `!int MyClass.m*()`

Primitívne bodové prierezy

- Primitívne bodové prierezy zodpovedajú identifikovaným typom bodov spájania
- AspectJ zahŕňa viac než dvadsať primitívnych bodových prierezov
- Základné skupiny primitívnych bodových prierezov
  - volania a vykonávania metód a konštruktorov
  - bodové prierezy založené na toku riadenia
  - bodové prierezy vykonávajúcich objektov
  - bodové prierezy argumentov
  - bodové prierezy založené na lexikálnej štruktúre
  - prístup k atribútom
  - inicializácia
  - spracovanie výnimiek
  - vykonanie videnia
  - podmienený bodový prierez
  - anotačné bodové prierezy

Volania a vykonávania metód a konštruktorov

```
call(method_signature)
call(constructor_signature)
execution(method_signature)
execution(constructor_signature)
```

- `* Point+.*(..)` – všetky metódy triedy `Point` a jej podtried
- `Point.new(..)` – všetky konštruktory triedy `Point`
- `* *.set*(..) throws InvalidCoordinatesException` – všetky metódy, ktorých nazov začína na **set**, a ktoré vyhadzujú výnimku `InvalidCoordinatesException`

Bodové prierezy založené na toku riadenia

```
cflow(pointcut)
cflowbelow(pointcut)
```

- Zahŕňajú všetky body spájania vo vykonávaní zadaného bodového prierezu
- V prípade **cflowbelow**() vynechávajú sa body spájania samotného zadaného bodového prierezu
- Aplikujeme nasledujúce videnie nad reprezentáciou bodu

```
before(): !within(PointMonitoring) && cflow(call(* Point.setX(..))) {
   System.out.println(thisJoinPoint);
}
```

- Výstup bude nasledujúci:

```
call(void Point.setX(int))
execution(void Point.setX(int))
set(int Point.x)
```

- Keby sme aplikovali **cflowbelow**(), vo výstupe by nebol prvý riadok

Bodové prierezy vykonávajúcich objektov

this(`type`)
target(`type`)

- **this**(`Point`) – body spájania, v ktorých kontrolu má objektu typu `Point` alebo jeho podtypu, t.j. pre ktoré platí **this instanceof** `Point` == **true**
- **target**(`Point`) – body spájania, z ktorých sa aktivuje bod spájania v objekte typu `Point` alebo jeho podtypu
- Typický sa používajú s bodovým prierezom **call**()
- Typ môže byť reprezentovaný aj identifikátorom objektu (pre potreby práce s kontextom)

Bodové prierezy argumentov

args(`argument_list`)

- `argument_list` je zoznam argumentov
- Argument je reprezentovaný:
  - svojim typom definovaným signatúrou typu
  - identifikátorom objektu (pre potrebe práce s kontextom)
- **args**(**int**) – všetky body spájania vo všetkých metódach, ktorých jeden argument je typu **int**

Bodové prierezy založené na lexikálnej štruktúre

within(`type_signature`)
withincode(`method_signature`)
withincode(`constructor_signature`)

- **within**(`Point`+) – body spájania v rámci triedy `Point` jej podtypov
- !**within**(`PointMonitoring`) && **call**(* *.*(..)) – volania všetkých metód mimo aspektu `PointMonitoring`
  - Idióm pre vyhnutie sa rekurzívnej aplikácii videnia
- **withincode**(**public static void** `Test`.`main`(`String`[])) – body spájania v rámci metódy **public static void** `Test`.`main`(`String`[])

Prístup k atribútom

get(`field_signature`)
set(`field_signature`)

- **get**(∗ `Point.x`) – čítanie atribútu `x` objektov triedy `Point`
- **set**(**private** ∗ ∗.∗) – nastavenie všetkých atribútov s prístupom typu `private`

Inicializácia

staticinitialization(`type_signature`)
initialization(`constructor_signature`)
preinitialization(`constructor_signature`)

- **staticinitialization**(∗) – statická inicializácia všetkých tried (napr. pre sledovanie poradia načítavania tried)
- **initialization**(`Point.new`()) – inicializácia objektu triedy `Point` konštruktormi bez argumentov
- **preinitialization**(`Point.new`()) – inicializácia objektu triedy `Point` (pred volaním **super**()) konštruktormi bez argumentov

Spracovanie výnimiek

handler(`type_signature`)

- **handler**(`InvalidCoordinatesException`) – vykonanie bloku spracovania výnimky `InvalidCoordinatesException` (t.j. ak k tejto výnimke príde)

Vykonanie videnia

advice()

- `advice`() – vykonanie všetkých videní
- **within**(`PointMonitoring`) && `advice`() – vykonanie všetkých videní v rámci aspektu `PointMonitoring`

Podmienený bodový prierez

if(`boolean_expression`)

- **if**(`p.x` < 100) – všetky body spájania, v ktorých platí podmienka, že atribút `x` objektu `p` má hodnotu menšiu než 100
  - Objekt `p` je súčasťou kontextu a deklaruje sa vo videní
  - Mechanizmus prenášania bude vysvetlený neskôr

Definovanie zložených bodových prierezov

- Pomocou primitívnych bodových prierezov a logických spojok a (&&), alebo (||) a negácie (!)
- **!within**(`PointMonitoring`) && **call**(∗ ∗.∗(..)) – volania všetkých metód okrem volaní v rámci aspektu `PointMonitoring`
- **call**(∗ `Point.set`∗(..)) || **call**(∗ `Point.get`∗()) – volania všetkých metód ∗ `Point.set`∗(..)) alebo ∗ `Point.get`∗())

Pomenované bodové prierezy

- Bodové prierezy je možné pomenovať

  ```
  aspect PointMonitoring {
     . . .
     pointcut getAndSet():
        call(* Point.set*(..)) || call(* Point.get*());
     . . .
  }
  ```

- Pomenované bodové prierezy sa ďalej dajú použiť
  - ako bodové prierezy pre vykonávanie videní
  - pre definovanie iných zložených bodových prierezov

```
aspect PointMonitoring {
   . . .
   pointcut getMethods(): call(* Point.get*());
   pointcut getAndSet(): call(* Point.set*(..)) || getMethods();

   before(): getAndSet() {. . .}
   . . .
}
```

- Pomenované bodové prierezy sa dedia

10 Videnia

Čo možno robiť so zachyteným bodom spájania?

Rámcová syntax videní

```
advice_type(argument_list): pointcut { advice_body }
```

- Videnia definujú aktivity (`advice_body`), ktoré sa majú vykonať v spojitosti so zachytenými bodmi spájania
  - Zoznam argumentov videnia (`argument_list`) sa používa na prenos kontextu

  - Bodový prierez videnia (**pointcut**) môže byť pomenovaný alebo nepomenoavný

- Tri typy videní (`advice_type`):
  - **before**
  - **after**
  - **around**

Videnie `before`

before(`argument_list`): **pointcut** {. . .}

- Vykonáva sa pred bodom spájania

```
before(): call(void Point.get*(..)) {
  System.out.println("Reading a point.");
}
```

Videnie `after`

`after`(`argument_list`): **pointcut** {. . .}
`after`(`argument_list`): **returning**(`return_value`) **pointcut** {. . .}
`after`(`argument_list`): **throwing**(`exception`) **pointcut** {. . .}

- Vykonáva sa po bode spájania

```
after(): call(int Point.get*()) {
  System.out.println("--Point read.");
}
```

- Ak nepotrebujeme návratovú hodnotu, resp. výnimku, za klauzulami `return_value` a `exception` nemusíme ju uviesť (ani zátvorky)

Videnie `after returning`

- Videnie `after` je možné obmedziť na úspešne vykonané body spájania (pri ktorých nenastála výnimka)
- Predpokladajme napr., že metódy `Point.set*(..)` vyhadzujú `InvalidCoordinatesException` v prípade, že sa zadajú nepovolené súradnice
- Nás však zaujíma len prípad skutočného premiestnenia bodu

```
after() returning: call(void Point.set*(..)) {
  System.out.println("--Point moved.");
}
```

- Videnie `after` **returning** môže pracovať aj s návratovou hodnotou
- Návratovú hodnotu treba deklarovať za kľúčovým slovom **returning**
- V našom príklade by sme mohli napr. vypísať prečítanú súradnicu bodu:

```
after() returning(int c): call(int Point.get*()) {
  System.out.println("--Point read " + c);
}
```

Videnie `after throwing`

- Videnie **after** je možné obmedziť práve na body spájania, pri ktorých došlo k výnimke
- V našom príklade by sme mohli vypísať dodatočné upozornenie:

```
after() throwing: call(void Point.set*(..)) {
  System.out.println("--Moving point failed");
}
```

- Videnie **after throwing** môže pracovať s výnimkou
- Výnimku treba deklarovať za kľúčovým slovom **throwing**

```
after() throwing(InvalidCoordinatesException e):
  call(void Point.set*(..)) {
  System.out.println("--Moving a point failed: " + e);
}
```

Videnie `around`

```
return_value around(argument_list): pointcut {...}
```

- Vykonáva sa namiesto bodu spájania
- Vykonanie samotného bodu spájania sa dá vyvolať pomocou kľúčového slova `proceed()`
- Musí byť definovaný typ návratovej hodnoty (`return_value`)
- Môže byť **void**
- Ak sa uvedie `Object`, AspectJ zabezpečí pretypovanie (aj pre primitívne typy)
- `Object` sa musí uviesť ak bodový prierez zachytáva body spájania s rôznymi návratovými hodnotami
- `Object` pokrýva aj typ **void**
- Videnie **around** nemá veľký zmysel bez využitia kontextu bodu spájania

```
void around(int i): call(void Point.setX(int)) && args(i) {
  if (i > 0 && i < 320)
    proceed(i);
}
```

Priorita aspektov

- Priorita (precedence) aspektov sa prejavuje na úrovni poradia vykonávania videní
- Hovorí sa, že aspekt s vyššou prioritou dominuje nad aspektom s nižšou prioritou
- Dá sa nastaviť medzi aspektmi klauzulou `declare precedence`
- Pri dedení vyššiu prioritu má odvodený aspekt

Poradie vykonávania videní

- Základné pravidlo – bodu spájania sú vždy najbližšie videnia s najnižšou prioritou (tok programu zhora nadol):

before_1 (max)	around_1 (max)	
before_2	around_2	
...	...	
before_n (min)	around_n (min)	
bod spájania	bod spájania	bod spájania
	around_n (min)	after_n (min)
	...	...
	around_2	after_2
	around_1 (max)	after_1 (max)

- V rámci jedného aspektu rozhoduje lexikálne poradie: videnie uvedené skôr má vyššiu prioritu

Circular advice precedence

```
public class C {
   public void m() { System.out.println("mmm"); }
   public static void main(String[] args) { (new C()).m(); }
}

public aspect A {
   before(): call(void m()) { System.out.println("b1"); }
   before(): call(void m()) { System.out.println("b2"); }
   after(): call(void m()) { System.out.println("a1"); }
   void around(): call(void m()) {
      System.out.println("(");
      proceed();
      System.out.println(")");
   }
}
```

chyba pri kompilácii: can't determine precedence between two or more pieces of advice that apply to the same join point

→ presunúť **after**() za **around**()

11 Kontext bodu spájania

Zachytili sme bod spájania. Čo všetko zaujímavé je okolo neho?

Zachytenie kontextu bodu spájania

- Kontext bodu spájania predstavujú aktuálne hodnoty prvkov, ktoré sú s nim spojené
- Napr. kontext volania metódy predstavuje objekt, ktorý ju zavolal, cieľový objekt a argumenty
  - Kontext závisí aj od videnia

  - Pri videniach typu after prichádzajú do úvahy aj návratová hodnota a zachytená výnimka

- Kontext sa dá zachytiť prostredníctvom
  - reflektívneho API jazyka AspectJ
  - bodových prierezov **this**(), **target**() a **args**() a videní **after returning** a **after throwing**

- Tak ako v Jave uprednostňujeme polymorfizmus pred reflektívnym API, treba uprednostňovať zachytenie kontextu bodovými prierezmi kde sa to dá

Prenos kontextu bodu spájania

- Kontext bodu spájania sa prenáša podobne ako argumenty do metódy

```
void around(int i): call(void Point.setX(int)) && args(i) {
  if (i > 0 && i < 320)
    proceed(i);
}
```

- Videnie (akoby metóda) definuje typ premenných kontextu (akoby argumentov tejto metódy)
- Pri metóde hodnoty argumentov zadáme pri volaní
- Videnie však býva vyvolané bodovým prierezom
- Hodnoty premenných kontextu videnie dostáva z bodového prierezu
- Následne sa dajú využiť v tele videnia (ako argumenty v tele metódy)

Prenos kontextu s pomenovaným bodovým prierezom

```
pointcut xSetter(int i): call(void Point.setX(int)) && args(i);

void around(int i): xSetter(i) {
  if (i > 0 && i < 320)
    proceed(i);
}
```

Hodnota atribútu

- Hodnotu, na ktorú sa chystáme nastaviť atribút, zachytíme tiež pomocou **args**()

```
void around(int i): set(int Point.x) && args(i) {
  if (i > 0 && i < 320)
    proceed(i);
}
```

Prenos návratovej hodnoty vo videní after

```
pointcut xGetter(): call(int Point.getX());

after() returning(int i): xGetter() {
  System.out.println("Vratena hodnota: " + i);
}
```

Prenos výnimky vo videní after

```
pointcut xSetter(int i): call(void Point.set*(i));

after() throwing(InvalidCoordinatesException e):
  call(void Point.set*(int)) {
  System.out.println("Suradnica mimo rozsahu: " + e);
}
```

12 Medzitypové deklarácie

Symetrický prístup

- Monitorovanie prístupu v syntaxi podobnej jazyku Hyper/J (staršia verzia)
- Monitorovanie implementujeme ako zvláštnu triedu:

```
class PointAccessMonitoring {
  void movingPoint() {System.out.println("Moving a point."); }
  void movedPoint() {System.out.println("Moved a point."); }
}
```

- Potom *zvlášť* definujeme pravidlá spájania (composition rules):

```
namespace AccessMonitoredGraphics{
  AccessMonitoredGraphics.Point.setX :=
    Merge[Graphics.Point.setX, Monitoring.PointAccessMonitoring.movingPoint];

  AccessMonitoredGraphics.Point.setY :=
    Merge[Graphics.Point.setY, Monitoring.PointAccessMonitoring.movingPoint];
  ...
}
```

- Hyper/J používa koncept priestoru mien (namespace) – rôzne aspekty tej istej záležitosti potom môžu byť pod rovnakým názvom
- Predpokladáme, že trieda `Point` je definovaná v priestore mien `Graphics`
- Získame novú triedu `Point` s pripojeným kódom pre monitorovanie
- Aj v symetrickom prístupe definujeme aspekty a body spájania, ku ktorým sa majú pripojiť
- Rozdiel je skôr vo vnímaní aspektov: uvažujeme ako sa veci spájajú, nie ako jednou vecou zasiahnuť do iných
- Dôležité pre aspektovo-orientovanú analýzu a návrh

- Zvlášť dôležité pre tzv. aplikačne špecifické aspekty – menej pre všeobecné aspekty ako sú monitorovanie, logovanie, synchronizácia, perzistencia, bezpečnosť apod.
- Deklarovane symetrický aspektovo-orientovaný programovací jazyk v praxi nenájdeme
- Niektoré populárne jazyky vykazujú prvky symetrickej aspektovo-orientovanej dekompozície:[11]
  - Scala – črty (traits)
  - Ruby – otvorené triedy
  - JavaScript – prototypy

Symetrická aspektovo-orientovaná dekompozícia umožňuje mať oddelené pohľady na tu istú triedu. Čo by bolo analógiou tohto v asymetrickom aspektovo-orientovanom programovaní?

Medzitypové deklarácie

- Aspekty môžu zavádzať nové prvky do typov a vzťahy medzi typmi
- Triedy, rozhrania a aspekty označujeme ako typy
- Medzitypové deklarácie (inter-type declarations)
  - Predtým známe ako introductions
- Druhy medzitypových deklarácií:
  - zavedenie členského prvku (atribútu alebo metódy)
  - zavedenie vzťahu dedenia
  - zavedenie chyby pri kompilácii
  - zavedenie anotácie
  - zmäkčenie výnimky (exception softening)

Príklad: účet – trieda Account

- Príklad z AspectJ in Action[12]

```
public abstract class Account {
  private float _balance;
  private int _accountNumber;

  public Account(int accountNumber) {
    _accountNumber = accountNumber;
```

[11] 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.
[12] http://www.manning.com/laddad/

```
  }

  public void credit(float amount) {
    setBalance(getBalance() + amount);
  }

  public void debit(float amount) throws InsufficientBalanceException {
    float balance = getBalance();
    if (balance < amount) {
      throw new InsufficientBalanceException(
        "Total balance not sufficient");
    }
    else {
      setBalance(balance - amount);
    }
  }

  public float getBalance() {
    return _balance;
  }

  public void setBalance(float balance) {
    _balance = balance;
  }
}

public class SavingsAccount extends Account {
    public SavingsAccount(int accountNumber) {
        super(accountNumber);
    }
}


public aspect MinimumBalanceRuleAspect {
  private float Account._minimumBalance;

  public float Account.getAvailableBalance() {
    return getBalance() - _minimumBalance;
  }

  after(Account account):
    execution(SavingsAccount.new(..)) && this(account) {
    account._minimumBalance = 25;
  }

  before(Account account, float amount) throws InsufficientBalanceException:
    execution(* Account.debit()) && this(account) && args(amount) {
    if (account.getAvailableBalance() < amount) {
      throw new InsufficientBalanceException(
        "Insufficient available balance");
    }
  }
}
```

Príklad: účet – zavedenie členského prvku

- Zaviedli sme atribút `_minimumBalance` a metódu `getAvailableBalance()` na sledovanie minimálneho zostatku
- Modifikátory prístupu týchto prvkov sú vzhľadom k aspektu, ktorý ich

zaviedol

- Prečo sme tieto prvky neuviedli rovno v triede?
  - Správu minimálneho zostatku pokladáme za zvláštnu záležitosť
  - Nechceme ju miešať so základnou funkcionalitou účtu

Zavedenie dedenia

- Dá sa zaviesť aj vzťah implements, aj extends
- Musia sa dodržiavať pravidla dedenia platné v Jave
- Zavedenie značkovacieho rozhrania za účelom sledovania prvkov:

```
aspect AccountTrackingAspect {
  declare parents: banking..entities.* implements Identifiable
}
```

- Závadzanie vzťahov extends je zaujímavé predovšetkým z hľadiska konfigurovateľnosti
  - Viac tried rovnakého rozhrania, ale inej implementácie
  - Aspektom určíme ktorá z týchto tried sa využije
  - Vrátime sa k tomu v súvislosti s Theme/UML

13 Abstraktné aspekty

Abstraktné aspekty

- Aspekty môžu byť abstraktné – kľúčové slovo **abstract**
- Podobne ako abstraktné triedy, abstraktné aspekty nemôžu mať inštancie
- Od abstraktných aspektov môžu dediť ďalšie abstraktné aspekty a konkrétne aspekty
- Od konkrétnych aspektov sa nedá dediť
- Aspekty môžu dediť od tried (bez ohľadu na abstraktnosť – všetky kombinácie)

Abstraktné bodové prierezy

- Abstraktné aspekty môžu obsahovať abstraktné bodové prierezy – bodové prierezy bez tela
- Abstraktný aspekt nad abstraktnými bodovými prierezmi definuje plnohodnotné videnia
- Abstraktné bodové prierezy sa dajú prekonať (konkretizovať) v odvodených aspektoch
- Abstraktné aspekty sú výhodné ak vieme špecifikovať čo sa má urobiť pre množinu bodov spájania, ktorú nevieme špecifikovať vo všeobecnosti

Príklad: abstraktný monitor

```
public abstract aspect MyMonitor {
  public abstract pointcut monitoredPoints();

  before(): monitoredPoints() {
    System.out.println("Monitor:" + thisJoinPoint);
  }
}

public aspect MyPointMonitor extends MyMonitor {
  public pointcut monitoredPoints():
    call(void Point.set*(..)) && call(* Point.get*());
}
```

14 Reflektívna podpora pre body spájania

Reflektívna podpora pre body spájania

- Analogicky k Java Reflective API, AspectJ poskytuje reflektívnu podporu pre body spájania
- Za týmto účelom AspectJ poskytuje tri špeciálne objekty (podobne ako **this** v Jave):
  - **thisJoinPoint**
  - `thisJoinPointStaticPart`
  - `thisEnclosingJoinPointStaticPart`
- Dajú sa použiť pre získanie jednoduchej textovej informácie (majú prekonanú metódu toString()):

  ```
  after() returning(Object x): call(* C.m()) {
    System.out.println(thisJoinPoint);
  }
  ```

- Ale aj pre zistenie kontextu bodu spájania

`thisJoinPoint`

- Informácie o aktuálnom bode spájania
- getArgs()
- getTarget()
- getThis()
- getStaticPart()

`thisJoinPointStaticPart`

- Statická časť informácií o aktuálnom bode spájania
- getKind()
- getSignature()
- getSourceLocation()

`thisEnclosingJoinPointStaticPart`

- Statické informácie o zahŕňajúcom bode spájania
- Bod spájania, ktorý obsahuje daný bod spájania
- Napr. pre volanie metódy je to vykonávanie metódy v rámci ktorej bola zavolaná
- Malé cvičenie: na aký bod spájania bude ukazovať `thisEnclosingJoinPointStaticPart` pri bode spájania vykonávania metódy?

```
before(): execution(* C.m()) {
  System.out.println(thisEnclosingJoinPointStaticPart);
}
```

15 Aspekty a anotácie

Anotácie v signatúrach[13]

- Signatúra typu anotácie:
  - `@<qualified−name>`, napr. `@Foo` alebo `@org.xyz.Foo`
  - `@(<type−pattern>)`, napr. `@(org.xyz..∗)` alebo `@(Foo` — Boo)—
- Signatúry typov anotácií možno použiť v signatúrach typov, atribútov, metód a konštruktorov:
  - `(@Immutable ∗)`
  - `(!@Immutable ∗)`
  - `@SensitiveData ∗ ∗`
  - `@Transaction ∗ (@Persistent org.xyz..∗).∗(..)`

Anotačné bodové prierezy

- Analogické k bežným verziám, len zachytávajú body spájania, ktoré majú zadanú anotáciu:
  - `@this(annot)`
  - `@target(annot)`
  - `@args(annot,...)`
  - `@within(annot)`
  - `@withincode(annot)`
  - `@annotation(annot)`
- `annot` je signatúra typu anotácie alebo identifikátor jej inštancie (pre potreby práce s kontextom)

[2] http://eclipse.org/aspectj/doc/released/adk15notebook/

Anotačné medzitypové deklarácie

- Medzitypové deklarácie na zavedenie anotácií:
  - `declare @type: C : @SomeAnnotation`
  - `declare @method: * C.foo*(..) : @SomeAnnotation`
  - `declare @constructor: C.new(..) : @SomeAnnotation`
  - `declare @field: * C.* : @SomeAnnotation`

Príklad: náhradná trieda

- Potrebujeme nahradiť inštancie danej triedy inštanciami novej triedy
- Trieda, ktorú chceme nahradiť:

```
class OldClass {
   public OldClass() {
      ...
   }
   ...
}
```

- Trieda, ktorou nahrádzame:

```
class NewClass extends OldClass {
   public NewClass() {
      ...
   }
   ...
}
```

- Aspekt, ktorý zabezpečí nahradenie:

```
aspect Swap {
   public pointcut oldClassConstructor(): call(OldClass.new());

   Object around(): oldClassConstructor() {
      return new NewClass();
   }
}
```

- Stačilo by použiť štandardný mechanizmus Javy: `@Deprecated`
- Kompilátor upozorní na použitie starej triedy
- Aspekt, ktorý zabezpečí zavedenie anotácie `@Deprecated`:

```
aspect Swap_a {
   declare @constructor: OldClass.new(): @Deprecated;
}
```

16 Inštanciácia aspektov

Ako sa aktivujú aspekty?

Inicializácia aspektov

- Inštancie aspektov sa nedajú priamo vytvárať
- Predsa vznikajú a aspekty môžu mať stav daný ich atribútmi
- Aspekty teda môžu obsahovať atribúty a metódy – ako triedy
- Aspekty sa aj inicializujú podobne ako triedy:
  - konštruktormi
  - blokom príkazov
  - statickým blokom príkazov

Konštruktory aspektov

- Budeme experimentovať s triedou `C`:

```
public class C {
   int i;

   int m() { return i * i; }

   public static void main(String[] args) {
      int a = new C().m();
      int b = new C().m();
      int c = new C().m();
  }
}
```

Konštruktory konkrétnych aspektov

- Konkrétne aspekty môžu mať len konštruktor bez argumentov

```
public aspect B {
   public B() {
      System.out.println("Aspekt B");
   }

   Object around(C t): execution(* C.m()) && this(t) {
      System.out.println("B: " + this);
      return proceed(t);
   }
}
```

- Videnie v aspekte `B` sa vyvolá trikrát
- Zakaždým sa vypíše rovnaká referencia aspektu **this**

Konštruktory abstraktných aspektov

- Abstraktné aspekty môžu mať aj konštruktory s argumentmi

```
public abstract aspect A {
   protected int n;

   public A(int i) { n = i; }
   public A(int i, int j) { n = i * j; }
}
```

- Konkrétne aspekty potom môžu vyvolať konštruktor, ktorý potrebujú

```
public aspect B extends A {
   public B() {
      super.A(25);
   }
}
```

Inštanciácia aspektov

- Inštancie aspektov sa tvoria automaticky na základe bodového prierezu
- Ako presne – pre každý bod spájania alebo len jedna inštancia?
- Terminologická poznámka:
  - Inštancie tried sa volajú objekty
  - Inštancie aspektov sa volajú – aspekty!
  - Ak z kontextu nie je jasné o čo ide, lepšie je inštancie aspektov explicitne takto označiť

Druhy inštanciácie aspektov

- Inštancie aspektov sa nedajú priamo vytvárať, ale dá sa regulovať ako sa vytvárajú, t.j. spôsob inštanciácie
- Presnejšie ide o to v spojitosti s čím sa inštancie aspektov vytvárajú – preto sa hovorí aj o druhoch asociácie aspektov
- Spôsoby inštanciácie aspektov:
  - pre celý program, t.j. virtuálny stroj (implicitne) – **issingleton**()
  - pre objekt – **perthis**() a **pertarget**()
  - pre tok riadenia – **percflow**() a **percflowbelow**()
  - pre typ – `pertypewithin(type_pattern)`
- Syntax (hranaté zátvorky označujú voliteľnosť):

```
aspect Aspect1 [inst_specifier([pointcut])] {
   . . .
}
```

- Aspekt dedí spôsob inštanciácie od nadaspektu a nemôže ho zmeniť

Jednočlenný aspekt

- Implicitný spôsob inštanciácie; možno ho uviesť explicitne:

```
aspect Aspect1 issingleton() {
   . . .
}
```

- Vytvorí sa presne jeden aspekt daného typu
- Inštanciácia sa odohrá keď sa dosiahne prvý bod spájania príslušného aspektu

Aspekt pre objekt

- Keď potrebujeme v inštancii aspektu udržiavať stav pre každý dotknutý objekt
- Dva druhy:
  - **perthis**() – pre vykonávaný objekt
  - **pertarget**() – pre cieľový objekt
- V zátvorkách sa uvedie bodový prierez
- Inštancia aspektu sa vytvorí pre každý zachytený vykonávaný alebo cieľový objekt pri príslušnom bode spájania

```
public aspect X perthis(this(C)) {
  Object around(C t): execution(* C.m()) && this(t) {
    System.out.println(this);
    return proceed(t);
  }
}
```

- Pri pripojení tohto aspektu k triede `C` zo slajdu 30 vypíšu sa tri rôzne referencie na aspekt
- Namiesto **this** sme mohli použiť aj volanie **aspectOf**(`t`)
  - Statická metóda aspektu
  - Pre jednočlenný aspekt je bez argumentov
- Bodový prierez pre inštanciáciu aspektov nemusíme určiť hneď

  ```
  public abstract aspect Abc perthis(mypoints()) {
    abstract pointcut mypoints();
    . . .
  }
  ```

- Konkrétné aspekty tak predsa môžu ovplyvniť inštanciáciu:

  ```
  public aspect AbcSpec extends Abc {
    pointcut mypoints(): call(* C.m(..));
    . . .
  }
  ```

Aspekt pre tok riadenia

- Môžeme mať aj inštanciu aspektu viazanú na tok riadenia
- Dva druhy – analogicky k bodovým prierezom `cflow`() a **cflowbelow**():
  - **percflow**()
  - **percflowbelow**()
- V zátvorkách sa tiež uvedie bodový prierez ako pri aspektoch pre objekty

- Inštancia aspektu sa vytvorí pre každý zachytený tok riadenia pri (**cflow**()) alebo pod (**cflowbelow**()) zachyteným bodom spájania
- O toku riadenia možno uvažovať ako o konceptuálnom objekte
- Inštancie aspektov pre toky riadenia možno potom využiť pri manažmente transakcií
- Príklad:[14]

```
public abstract aspect TransactionManagementAspect percflow(transacted()) {
  abstract pointcut transacted();
  . . .
}

public aspect BankingTransactionManagementAspect
  extends TransactionManagementAspect {
  pointcut transacted(): execution(* banking..Account+.*(..)) ||
    execution(* banking..Customer+.*(..)));
  . . .
}
```

Aspekt pre typ

- Niekedy nie je potrebné mať aspekt pre každú inštanciu nejakého typu
- Ale niekedy stačí jeden aspekt pre typ ako taký
- Napríklad môžeme potrebovať aspekt pre každý typ v určitom balíku:

```
aspect AspectT pertypewithin(myPackage..*) {
  . . .
}
```

- V zátvorkách sa na rozdiel od iných typov inštanciácie neuvádza bodový prierez, ale signatúra typu

17 Aspektovo-orientované návrhové vzory a idiómy

Idiómy a vzory vo vývoji softvéru

- Pôvod vzorov
  - Christopher Alexander – vzory v architektúre
  - Iný pohľad na architektúru: jazyk vzorov
- Hillside Group: K. Auer, G. Booch, R. Johnson, H. Hilerbrand, K. Beck, W. Cunnigham a J. Coplien – 1993[15]

[14] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

[15] http://www.hillside.net/

  - Kultúra vzorov[16]

Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.[17]

Druhy vzorov

- Úroveň granularity vzorov:
  - analytické
  - architektonické
  - návrhové (*design patterns*)
  - idiómy
- Rôzne katalógy vzorov:
  - GoF vzory
  - PLoP konferencie[18]
  - J2EE vzory[19]
- Jazyky vzorov

Aspektovo-orientované návrhové vzory a idiómy

- Aspektovo-orientované idiómy – viažu sa na jazyk
- Autochtónne aspektovo-orientované návrhové vzory
- Aspektovo-orientované reimplementácie objektovo-orientovaných návrhových vzorov (všetky GoF)
- Zatiaľ väčšinou v jazyku AspectJ
- Niektoré autochtónne aspektovo-orientované vzory boli implementované aj v jazyku CaesarJ[20] [21], ale asymetrickým spôsobom
- Niektoré boli implementované aj symetrickým spôsobom v jazykoch Hyper/J a CaesarJ [22] [23]
- Reimplementácie niektorých objektovo-orientovaných návrhových vzorov boli realizované v jazyku CaesarJ[24]

[16] `http://www.comsis.fon.bg.ac.yu/ComSIS/Volume02/InvitedPapers/JamesCoplin.htm`
[17] `http://www.hillside.net/patterns/definition.html`
[18] `http://st-www.cs.uiuc.edu/~plop/`
[19] `http://java.sun.com/blueprints/corej2eepatterns/index.html`
[20] `http://caesarj.org/`
[21] R. Šelmeci. Tvorba aplikácii kombinovaním aspektovo-orientovaných návrhových vzorov. Bakalársky projekt, FIIT STU, 2008.
[22] J. Bálik and V. Vranić. Sustaining Composability of Aspect-Oriented Design Patterns in Their Symmetric Implementation. In 2nd International Workshop on Empirical Evaluation of Software Composition Techniques, ESCOT 2011, at ECOOP 2011, July 2011, Lancaster, UK.
[23] J. Bálik. Diversity of Aspect-Oriented Approaches and Aspect-Oriented Design Patterns. Diplomová práca, FIIT STU, 2010.
[24] `http://caesarj.org/`

- Ukazuje sa, že niektoré objektovo-orientované vzory môžu byť nahradené autochtónnymi aspektovo-orientovanými vzormi[25] [26]

Kategorizácia aspektovo-orientovaných vzorov

- Jedna z možných kategorizácií aspektovo-orientovaných vzorov podľa prevládajúceho prvku AOP (ovplyvnená asymetrickým chápaním AOP):[27] [28]
  - Vzory bodových prierezov (Wormhole a Avoiding the Advising of Aspect Elements)
  - Vzory videní (Cuckoo's Egg a Worker Object Creation)
  - Vzory medzitypových deklarácií (Providing a Default Interface Implementation)
- V zátvorkách sú uvedené vzory, na ktoré sa pozrieme bližšie
- Táto kategorizácia má význam z hľadiska kompozície aspektovo-orientovaných vzorov
- Známe aspektovo-orientované vzory (vrátane idiómov) boli zaradené do týchto kategórií[29]

18 Idiómy jazyka AspectJ

Idiómy

- Idióm – spôsob používania jazykových konštrukcií ktorý nie je zrejmý priamo z poznania jednotlivých prvkov, z ktorých pozostáva
- Idiómy v prirodzených jazykoch sa označujú ešte ako frázy[30]
- V programovacích jazykoch sa idiómy považujú za „malé" vzory
  - Malé rozsahom – krátke úryvky kódu
  - Malé dosahom – platia len pre daný jazyk (a veľmi podobné jazyky)

[25] P. Bača and V. Vranić. Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns. In Proc. of 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2011, September 2011, Bratislava, Slovakia, IEEE Computer Society. To appear.

[26] P. Bača. Nahradenie objektovo-orientovaných vzorov autochtónnymi aspektovo-orientovanými vzormi. Diplomová práca, FIIT STU, 2010.

[27] R. Menkyna, V. Vranić, and I. Polášek. Composition and Categorization of Aspect-Oriented Design Patterns. In Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010, January 2010, Herľany, Slovakia, IEEE.

[28] R. Menkyna. Aspect-Oriented Design Patterns. Bakalársky projekt, FIIT STU, 2007.

[29] P. Bača. Nahradenie objektovo-orientovaných vzorov autochtónnymi aspektovo-orientovanými vzormi. Diplomová práca, FIIT STU, 2010.

[30] `http://en.wikipedia.org/wiki/Idiom`

Idiómy jazyka AspectJ

- Pojem idióm sa používa dosť voľne
- V prvom vydaní *AspectJ in Action* boli uvedené štyri idiómy, z ktorých niektoré vyzerali ako bežné syntaktické záležitostí (napr. poskytnutie prázdných definícií bodových prierezov)
- Idiómy v druhom vydaní *AspectJ in Action* sa zdajú byť relevantné:
  - Providing a Default Interface Implementation
  - Top-Level Join Point
  - Avoiding the Advising of Aspect Elements
  - Selecting Join Points Only in Subclasses
  - Providing a Default Interface Implementation
  - Return-Value Restriction
- Ďalším zaujímavým zdrojom je práca Stefana Hanenberga a kol.[31]

Vyhnutie sa ovplyvneniu prvkov aspektu

- Avoiding the Advising of Aspect Elements[32]
- Ak bodový prierez pri určitom videní zachytáva body spájania v samotnom videní, videnie sa rekurzívne aktivuje
- Riešenie je vylúčenie bodov z príslušného aspektu

```
public aspect Tracing {
    before(): call(* *.*(..)) && !within(Tracing) {
        System.out.println("Calling: " + thisJoinPointStaticPart);
    }
}
```

- Problém nie je taký jednoduchý[33]

Poskytnutie implicitnej implementácie v rozhraní

- Default Interface Implementation
- Niekedy väčšina implementácií rozhrania implementuje jeho metódy rovnako (napr. pri adaptéroch vo Swingu)
- Bolo by výhodné, keby implicitná implementácia mohla byť v samotnom rozhraní. . .[34]

---

[31] S. Hanenberg, A. Schmidmeier, and R. Unland. AspectJ Idioms for Aspect-Oriented Software Construction. In Proc. of 8th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, June 25-–29, 2003. http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/KuleszaEtAl.pdf

[32] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

[33] E. Bodden et al. Avoiding Infinite Recursion with Stratified Aspects. In Robert Hirschfeld et al., editors, Proc. of NODe 2006, LNI P-88, Erfurt, Germany, September 2006. http://www.sable.mcgill.ca/~ebodde/meta/

[34] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

```
public interface Identifiable {
   public void setId(String id);
   public String getId();

   static aspect Impl {
      private String Identifiable._id;
      public void Identifiable.setId(String id) { _id = id; }
      public String Identifiable.getId() { return _id; }
   }
}
```

- Ďalšie rozhranie

```
public interface Nameable {
   public void setName(String name);
   public String getName();

   static aspect Impl {
      private String Nameable._name;
      public void Nameable.setName(String name) { _name = name; }
      public String Nameable.getName() { return _name; }
   }
}
```

- Použitie:

```
public class Entity implements Nameable, Identifiable { }
```

- Pripomína to viacnásobné dedenie a tzv. mixin triedy
- Tento idióm je prekonaný v samotnej Jave 8, ktorá umožňuje uvádzať implementáciu metód v rozhraní

19 Vzor Worker Object Creation

Problémová situácia: Swing dispatching thread

```
public class Test {
   public static void main(String[] args) {
      JFrame appFrame = new JFrame();
      appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      DefaultTableModel tableModel = new DefaultTableModel(4,2);
      JTable table = new JTable(tableModel);
      appFrame.getContentPane().add(table);
      appFrame.pack();
      appFrame.setVisible(true);
      String value = "[0,0]";
      tableModel.setValueAt(value, 0, 0);
      JOptionPane.showMessageDialog(appFrame, "Press OK to continue");
      int rowCount = tableModel.getRowCount();
      System.out.println("Row count = " + rowCount);
      Color gridColor = table.getGridColor();
      System.out.println("Grid color = " + gridColor);
   }
}
```

- Volania, ktoré aktualizujú už realizované používateľské rozhranie založené na rámci Swing musia byť vykonané prostredníctvom odosielacej nite Swingu (Swing dispatching thread)

- V obyčajnej Jave každé také volanie musíme identifikovať a obaliť do konštrukcie podobnej tejto:

```
Event-thread.invokeLater(new Runnable() {
   public void run() {
      . . . // kod ktory sa ma vykonat
   }
});
```

Pracujúci objekt

- Ako v Jave preniesť odkaz na metódu, prostredníctvom ktorého ju možno spústiť?
- Java takéto niečo priamo nepodporuje, ale dá sa to pomocou tzv. pracujúcich objektov (worker objects)
- Taký objekt implementuje potrebnú metódu pod štandardným názvom (používa sa rozhranie `Runnable` a metóda `run()`)
- Napr. metóde `m()` pošleme pracujúci objekt

```
m(new Runnable() {
   public void run() {
      . . . // kod ktory sa ma vykonat
   }
});
```

- Metóda `m()` ho vykoná

```
m(Runnable o) {
   o.run();
}
```

Vzor Worker Object Creation

- Potrebujeme zabezpečiť, aby každé volanie určitých metód bolo vykonané prostredníctvom pracujúceho objektu
- Takto obalené volania potom môžeme preniesť do iného kontextu
- V AOP môžeme zachytiť volania všetkých takých metód a obaliť ich do pracujúceho objektu:

```
void around() : <pointcut> {
   Runnable worker = new Runnable () {
      public void run() {
         proceed();
      }
   };
   invoke.Queue.add(worker); // vykonanie - aj odlozene
}
```

- To je podstata vzoru Worker Object Creation[35] (alias Proceed Object[36])

Aplikácia pri práci s rámcom Swing

```
public class Test {
   public static void main(String[] args) {
      JFrame appFrame = new JFrame();
      appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      DefaultTableModel tableModel = new DefaultTableModel(4,2);
      JTable table = new JTable(tableModel);
      appFrame.getContentPane().add(table);
      appFrame.pack();
      appFrame.setVisible(true);
      String value = "[0,0]";
      tableModel.setValueAt(value, 0, 0);
      JOptionPane.showMessageDialog(appFrame, "Press OK to continue");
      int rowCount = tableModel.getRowCount();
      System.out.println("Row count = " + rowCount);
      Color gridColor = table.getGridColor();
      System.out.println("Grid color = " + gridColor);
   }
}
```

- Abstraktný aspekt implementuje všeobecné riešenie – okrem bodových prierezov

```
public abstract aspect SwingThreadSafetyAspect {
   abstract pointcut uiMethodCalls();
   abstract pointcut uiSyncMethodCalls();

   pointcut threadSafeCalls(): call(void JComponent.revalidate())
      || call(void JComponent.repaint(..))
      || call(void add*Listener(EventListener))
      || call(void remove*Listener(EventListener));

   pointcut excludedJoinpoints(): threadSafeCalls()
      || within(SwingThreadSafetyAspect)
      || if(EventQueue.isDispatchThread());

   pointcut routedMethods(): uiMethodCalls() && !excludedJoinpoints();
   pointcut voidReturnValueCalls(): call(void *.*(..));

   void around(): routedMethods() && voidReturnValueCalls()
      && !uiSyncMethodCalls() {
      Runnable worker = new Runnable() {
         public void run() {
            proceed();
         }
      };
      EventQueue.invokeLater(worker);
      }

   Object around(): routedMethods()
```

[35] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

[36] S. Hanenberg, A. Schmidmeier, and R. Unland. AspectJ Idioms for Aspect-Oriented Software Construction. In Proc. of 8th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, June 25-–29, 2003. http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/KuleszaEtAl.pdf

```
        && (!voidReturnValueCalls() || uiSyncMethodCalls()) {
        RunnableWithReturn worker = new RunnableWithReturn() {
            public void run() {
                _returnValue = proceed();
            }
        };
        try {
            EventQueue.invokeAndWait(worker);
        } catch (Exception ex) {
        // ... log exception
            return null;
        }
        return worker.getReturnValue();
    }
}
```

- Konkrétny aspekt definuje bodové prierezy pre daný kontext

```
public aspect DefaultSwingThreadSafetyAspect extends SwingThreadSafetyAspect {
    pointcut viewMethodCalls(): call(* javax..JComponent+.*(..));

    pointcut modelMethodCalls(): call(* javax..*Model+.*(..))
        || call(* javax.swing.text.Document+.*(..));

    pointcut uiMethodCalls(): viewMethodCalls() || modelMethodCalls();

    pointcut uiSyncMethodCalls(): call(* javax..JOptionPane+.*(..))
        /* || ... */;
}
```

20 Vzor Wormhole

Problémová situácia

- Ako preniesť kontext volajúceho (vykonávajúci objekt, cieľový objekt a argumenty) volanému?
- Dá sa to urobiť prostredníctvom argumentov alebo zriadením úložiska pre danú niť
- Obidva spôsoby spôsobia prepletenie kódu
- Hľadáme body spájania definované vzhľadom na kontext volaného v toku riadenia bodov spájania definovaných v kontexte volajúceho:[37]

```
public aspect WormholeAspect {
  pointcut callerSpace(<caller context>): <caller pointcut>;
  pointcut calleeSpace(<callee context>): <callee pointcut>;
  pointcut wormhole(<caller context>, <callee context>):
    cflow(callerSpace(<caller context>)) && calleeSpace(<callee context>);

  // advices to wormhole
  void around(<caller context>, <callee context>):
    wormhole(<caller context>, <callee context>) {
    ... advice body
  }
}
```

[37] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

- Videnie **around**() je uvedené ako príklad – môže tam byť hociktorý typ videnia a viac videní

Príklad

- Predpokladajme, že v grafickom systéme chceme zabrániť posúvaniu grafických objektov, ktoré tvoria pozadie
- Grafický objekt, ktorý je súčasťou pozadia, má nastavený atribút `Background`
- Potrebujeme zachytiť operacie, ktoré nastavujú súradnice
- Pri týchto operáciach potrebujeme aj kontext – grafický objekt, ktorý operáciu vyvolal

```
public aspect BlockBackground {
   pointcut gObjects(GObject o): execution(* GObject+.*(..)) && this(o);

   pointcut pointSetters(Point p, int c): execution(void Point.set*(int)) && this(p) && args(c);

   pointcut pointSettersInGObjects(GObject o, Point p, int c): cflow(gObjects(o)) && pointSetters(p, c);

   void around(GObject o, Point p, int c): pointSettersInGObjects(o, p, c) {
      if (!o.background)
         proceed(o, p, c);
      }
}
```

21 Vzor Cuckoo′s Egg

Problémová situácia

- Problém: treba nahradiť objekt jedného typu objektom iného typu
- Nový typ v každom prípade musí byť odvodený od pôvodneho
- Následne by všetky výskyty pôvodneho typu bolo potrebné nahradiť novým
- Čo ak je to potrebné len za určitých podmienok?

Vzor Cuckoo's Egg

- Riešením je vzor Cuckoo's Egg[38] – tu je v Coplienovej forme:[39]

  Problem: Instead of an object of the original type, under certain conditions, an object of some other type is needed.

  Context: The original type may be used in various contexts. The need for the object of another type can be determined before the instantiation takes place.

  Forces: An object of some other type is needed, but the type that is going to be instantiated may not be altered.

---

[38] R. Miles. *AspectJ Cookbook.* O'Reilly, 2004.

[39] Slajdy prezentácie: J. Bálik and V. Vranić. Sustaining Composability of Aspect-Oriented Design Patterns in Their Symmetric Implementation. In 2nd International Workshop on Empirical Evaluation of Software Composition Techniques, ESCOT 2011, at ECOOP 2011, July 2011, Lancaster, UK.

Solution: Put the other type instead of the original type before instantiation and provide its instance instead of the original type instance if the conditions for this are fulfilled.

Resulting Context: The original type remains unchanged, while it appears to give instances of the other type under certain conditions. There may be several such types chosen for instantiation according to the conditions.

Rationale: No need to adapt the original type.

Základný tvar vzoru Cuckoo's Egg

- Základný tvar vzoru:

```
public aspect MyClassSwapper {
   public pointcut myConstructors(): call(MyClass.new());

   Object around(): myConstructors() {
      return new AnotherClass();
   }
}
```

- Trieda `AnotherClass` musí dediť od `MyClass`

```
public class AnotherClass extends MyClass {
   . . .
}
```

Ďalšie možnosti

- Bodový prierez môže zahŕňať volania konštruktorov viacerých tried

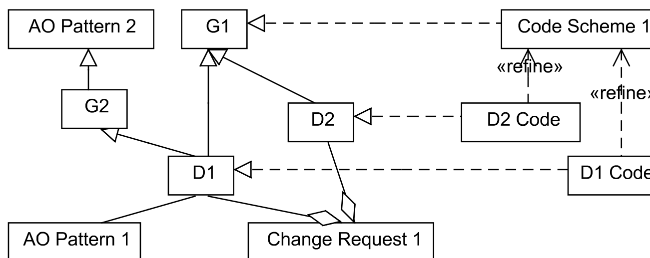
```
public pointcut myConstructors(): call(MyClass1.new()) || call(MyClass2.new());
```

- Náhradu možno obmedziť lexikálne

```
public pointcut myConstructors(): call(MyClass.new()) && within(SomeClass);
```

Praktické použitie

- Jeden zo vzorov vhodných na realizáciu zmien aspektmi
- Riadenie zmien pomocou AOP[40] [41]



[40] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog. Developing Applications with Aspect-Oriented Change Realization. In Proc. 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques CEE-SET 2008, October 2008, Brno, Czech Republic. To appear.

[41] M. Bebjak, V. Vranić, and P. Dolog. Evolution of Web Applications with Aspect-Oriented Design Patterns. In M. Brambilla and E. Mendes, eds., Proc. of ICWE 2007 Workshops, 2nd Int. Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with 7th Int. Conf. on Web Engineering, ICWE 2007, July 19, 2007, Como, Italy.

```
public aspect ExchangeClass {
   public pointcut exhangedClassConstructor(): call(ExchangedClass.new(..));
   ExchangedClass around(): exhangedClassConstructor() {
      return getExchangingObject(proceed());
   }
   ExchangedClass getExchangingObject(ExchangedClass obj) {
      if (. . .) {
         return obj;
      }
      else {
         return new ExchangingClass();
      }
   }
}
```

- Introducing a Resource Backup – zavedenie náhradného zdroja
- Predpokladajme, že prispôsobujeme aplikáciu na členský marketing (affiliate marketing)
- Jedna z jej vlastností je posielanie notfikácií o nových predajoch, registrovaných členoch atď.
- Chceli by sme mať záložný SMTP server pre posielanie notifikácií
- Táto zmena môže byť implementovaná ako Class Exchange

```
SMTPServer around(): call(SMTPServer.new(..)) && !within(SMTPServerBackup) {
   return getServerObject(proceed());
}
SMTPServer getExchangingObject(SMTPServer server) {
   if (server.isConnected()) {
      return server;
   }
   else {
      return new SMTPServer(/* alternative SMTP server params */);
   }
}
```

Náhradná trieda medzitypovou deklaráciou

- Ak chceme len upozorniť na použitie určitého typu, stačí použiť štandardný mechanizmus Javy: `@Deprecated`
- Kompilátor upozorní na použitie starej triedy
- Aspekt, ktorý zabezpečí zavedenie anotácie `@Deprecated`:

```
aspect Swap_a {
   declare @constructor: OldClass.new(): @Deprecated;
}
```

22 AO implementácia vzoru Observer

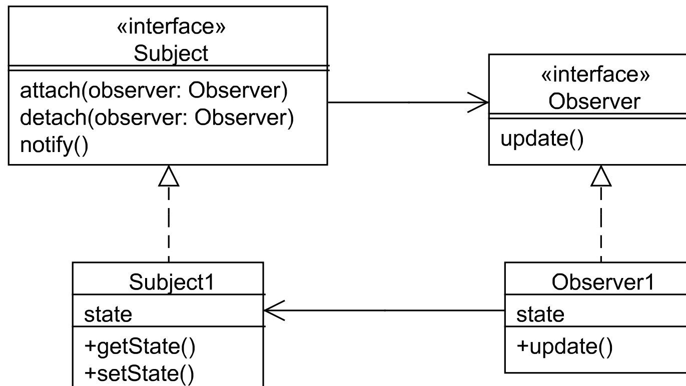
Aspektovo-orientovaná implementácia OO vzorov

- Pozreli sme sa na niektoré AO návrhové vzory a idiómy
- Klasické OO vzory sa často dajú vyjadriť kompaktnejšie pomocou AOP
- Je otázkou, či v AOP ešte stále zostávajú vzormi
- Hannemann a Kiczales implementovali GoF vzory AO spôsobom[42]
- Príklad: vzor Observer – trochu inak ako u Hannemanna a Kiczalesa

Vzor Observer

- Definovanie závislosti stavu viacerých objektov od ďalšieho objektu
- Vhodný ak treba oddeliť od seba dva vzájomne závislé aspekty
- Vhodný ak zmena v jednom objekte vyžaduje zmeny v iných objektoch, ktorých počet a presný typ nie je známy
- Vzťah Model-View vo vzore Model-View-Controler je príkladom použitia vzoru Observer – zodpovedá vzťahu Subject-Observer

Štruktúra vzoru Observer



Príklad použitia vzoru Observer

- Pozrieme sa na OO implementáciu vzoru Observer na príklade
- Problémová oblasť: teplotné senzory
- Rôzne spôsoby zobrazenia teploty: digitálny, analógový, rozsahový...

[42] http://www.cs.ubc.ca/labs/spl/projects/aodps.html

Senzor a zobrazenie

```
public interface TempSensor { // Subject interface
  void addDisplay(TempDisplay d); // attach observer
  void removeDisplay(TempDisplay d); // detach observer
  void notifyDisplays(); // notify observers
  double readTemp();
  void measureTemp();
}

public interface TempDisplay { // Observer interface
  void refresh(); // update observer
  void display();
  void measureTemp();
}
```

Senzor teploty ľudského tela

```
public class HumanTempSensor implements TempSensor { // a subject
  private ArrayList<TempDisplay> displays = new ArrayList<>(); // a list of observers
  private double temp;
  public double refreshRate;

  public void measureTemp() {
    // get the temperature from the physical device
    notifyDisplays();
  }
  public void setTempDebug(double t) { temp = t; }
  public void addDisplay(TempDisplay d) {
    displays.add(d);
  }
  public void removeDisplay(TempDisplay d) { /*...*/ }
  public void notifyDisplays() {
    for (int i = 0; i < displays.size(); i++) {
      ((TempDisplay)displays.get(i)).refresh();
    }
  }
}
```

Digitálne zobrazenie

```
public class DigitalTemp implements TempDisplay { // an observer
  private HumanTempSensor sensor;
  private float temp;
  public DigitalTemp(HumanTempSensor s) { sensor = s; }
  public void refresh() {
    temp = (float)sensor.readTemp();
  }
  public void display() { // only two decimal places
    System.out.println(Math.round(temp * 100.0) / 100.0);
  }
  public void measureTemp() {
    sensor.measureTemp();
  }
}
```

Rozsahové zobrazenie

```
enum TempRange {
  LOW, NORMAL, HIGH
}
```

```
public class RelTemp implements TempDisplay { // an observer
  private HumanTempSensor sensor;
  TempRange range;
  double high = 37.0;
  double low = 35.0;
  public RelTemp(HumanTempSensor s) { sensor = s; }

  public void refresh() {
    double temp = sensor.readTemp();

    if (temp <= low)
      range = TempRange.LOW;
    else if (temp >= high)
      range = TempRange.HIGH;
    else
      range = TempRange.NORMAL;
  }

  public void display() {
    switch (range) {
      case LOW: System.out.println("LOW");
        break;
      case HIGH: System.out.println("HIGH");
        break;
      default: System.out.println("NORMAL");
    }
  }

  public void measureTemp() {
    sensor.measureTemp();
  }
}
```

Použitie

```
public class M {
  public static void main(String args[]) {
    HumanTempSensor s = new HumanTempSensor();

    DigitalTemp d1 = new DigitalTemp(s);
    s.addDisplay(d1);
    RelTemp d2 = new RelTemp(s);
    s.addDisplay(d2);

    s.setTempDebug(37.33333333);

    d1.display(); // 37.33
    d2.display(); // HIGH
  }
}
```

AO implementácia vzoru Observer

- Veci týkajúce sa technického zabezpečenia vzoru Observer sa nemajú miešať s pracovnou logikou:
  - časti rozhraní `TempDisplay` a `TempSensor` a ich implementácia
  - logika spojenia displeja s príslušným senzorom
  - občerstvenie displejov pri zmene hodnoty v senzore

Senzor má byť len senzorom

```
public interface TempSensor {
  double readTemp();
  void measureTemp();
}

public class HumanTempSensor implements TempSensor {
  private double temp;
  double refreshRate;
  public double readTemp() { return temp; }
  public void measureTemp() { /*...*/ }
  void setTempDebug(double t) { temp = t; }
}
```

Displej má byť len displejom

```
public interface TempDisplay {
  void display();
  void measureTemp();
  void refresh();
}

public class DigitalTemp implements TempDisplay {
  private HumanTempSensor sensor;
  private float temp;
  public DigitalTemp(HumanTempSensor s) { sensor = s; }
  public void refresh() {
    temp = (float)sensor.readTemp();
  }
  public void display() { // only two decimal places
    System.out.println(Math.round(temp * 100.0) / 100.0);
  }
  public void measureTemp() {
    sensor.measureTemp();
  }
}
```

Ďalší displej

```
interface TempRange {
  int LOW = -1, NORMAL = 0, HIGH = 1;
}

public class RelTemp implements TempDisplay { // an observer
  private HumanTempSensor sensor;
  int range;
  double high = 37.0;
  double low = 35.0;
  public RelTemp(HumanTempSensor s) { sensor = s; }

  public void refresh() {
    double temp = sensor.readTemp();

    if (temp <= low)
      range = TempRange.LOW;
    else if (temp >= high)
      range = TempRange.HIGH;
    else
      range = TempRange.NORMAL;
  }
```

```
  public void display() {
    switch (range) {
      case TempRange.LOW: System.out.println("LOW");
        break;
      case TempRange.HIGH: System.out.println("HIGH");
        break;
      default: System.out.println("NORMAL");
    }
  }

  public void measureTemp() {
    sensor.measureTemp();
  }
} // TempRange
```

Logiku vzoru Observer zabezpečí aspekt

```
public aspect TempObserver {
  private ArrayList TempSensor.displays = new ArrayList();
  public void TempSensor.addDisplay(TempDisplay d) {
    displays.add(d);
  }
  public void TempSensor.removeDisplay(TempDisplay d) { /*...*/ }
  public void TempSensor.notifyDisplays() {
    for (int i = 0; i < displays.size(); i++) {
      ((TempDisplay)displays.get(i)).refreshTemp();
    }
  }
  public void TempDisplay.refreshTemp() {
    refresh();
  }

  after(TempSensor sensor): this(sensor) &&
    (execution(* TempSensor+.measureTemp(..)) ||
     execution(* TempSensor+.setTempDebug(..))) {
    sensor.notifyDisplays();
  }
}
```

23 Sumarizácia

- Aspektovo-orientované programovanie umožňuje zasiahnuť do kódu bez jeho úprav
- Aspektovo-orientované vnímanie nás sprevádza od skorých úvah o vývoji softvérového systému, pričom bežné spôsoby programovania
- neumožňujú jeho priame vyjadrenie
- Aspektovo-orientované črty sú prítomné aj v etablovaných programovacích jazykoch, ktoré nie sú označované ako aspektovo-orientované
- Pri modelovní softvéru je potrebné vyjadriť dva spôsoby aspektovo-orientovanej dekompozície: asymetrický a symetrický
- AspectJ je referenčnou implementáciou asymetrického AOP
- Zložitosť asymetrického AOP tkvie v jazyku bodových prierezov, ktoré definujú homogénne a heterogénne množiny bodov spájania

- Zachytený bod spájania a jeho kontext je plne pod kontrolou videnia
- Tvorba dobrých bodových prirezov je náročná
- Zavádzanie prvkov do tried pomocou medzitypových deklarácií umožňuje napodobniť symetrické AOP
- Inštanciáciu aspektov v AspectJ možno regulovať
- Pomocou aspektov možno
  - manipulovať volaniami ako objektmi: Worker Object Creation
  - preniesť kontext volajúceho k volanému: Wormhole
  - nahradiť objekt iným objektom a to aj iného typu: Cuckoo's Egg
  - možno dosiahnuť vyčlenenie logiky objektovo-orientovaných návrhových vzorov