# Аспектно-оријентисано програмирање: основе, могућности и узори

Валентино Вранић

Пословни факултет у Ваљеву, Универзитет Сингидунум, Београд

Slovak University of Technology, Bratislava, Slovakia

vranic@fiit.stuba.sk
http://fiit.stuba.sk/~vranic/

Универзитет Сингидунум
Пословни факултет у Ваљеву – 22. децембар 2008.
Факултет за информатику и менаџмент – 24. децембар 2008.

## Преглед


1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
4. Узори дизајна аспектно-оријентисаног програмирања
5. Сумаризација

# Пример: једноставни мониторинг

- Претпоставимо да развијамо мали графички систем
- Тачка је у њему представљена на следећи начин:

```
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; }
}
```

# Мониторинг операција над тачкама (1)

- Желимо да вршимо мониторинг операција над тачкама
- Морамо да модификујемо све set и get методе

# Мониторинг операција над тачкама (2)

```
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; }
}
```

# Шта смо добили?

- Преплетен код: код за мониторинг је преплетен с кодом апликативне логике
- Разбацан код: код за мониторинг се понавља у различитим методама
- Шта кад бисмо хтели да искључимо мониторинг?
- Донекле би помогао један ниво апстракције више:

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

- Мониторинг можемо да искључимо модификацијом методе print()
- Али нисмо елиминисали проблем самог присуства кода за мониторинг у апликативној логици, као ни потребу његовог ручног увођења

# Шта смо у ствари хтели?

- Хтели смо да *додамо* поруке мониторинга свим set и get методама
- Знамо тачно о којим се методама ради:
  1. методе класе Point чије име почиње на set
  2. методе класе Point чије име почиње на get
- Шта кад би програмски језик омогућавао да се искаже да *пре извршења* сваке такве методе треба исписати одговарајуће податке без потребе да се метода модификује?
- Програмски језик AspectJ управо то омогућава...

# Мониторинг као аспект (1)

- Аспект за мониторинг set и get метода класе Point:

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

- Аспект садржи два *виђења* типа (**before**())
- Оба виђења допуњавају *тачке спајања* дефинисане *тачкастим пресецима* типа **call**()

# Мониторинг као аспект (2)

- Код је довољно превести заједно с неизмењеном класом Point
- Извршењем наредби:

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

- добија се следећи излаз:

*Moving a point.*
*Reading a point.*
*Moving a point.*

# Мониторинг завршетка извршења метода (1)

- Како бисмо у језику Java вршили мониторинг завршетка извршења метода?
- Наизглед није проблем додати наредбу за излаз:

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

- Излаз се у ствари реализује у оквиру извршавања методе
- Код метода које враћају вредност овај проблем се сасвим јасно испољава:

```
public int getX() {
    System.out.println("Reading a point.");
    return x;
    System.out.println("Read a point."); // ne izvrsva se
}
```

# Мониторинг завршетка извршења метода (2)

- За мониторинг завршетка извршења метода потребно је додати наредбу за излаз *за сваки позив* у клијентском коду
- У језику AspectJ је могуће дефинисати излаз после стварног завршетка извршења методе:

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

# Мониторинг завршетка извршења метода (3)

- Могуће је дефинисати излаз и после сваког позива методе:

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

- Коришћењем тачкастог пресека **call**() између осталог можемо да ограничимо излаз на позиве реализоване у одређеним класама и методама

# Мониторинг завршетка извршења метода (4)

- Опет је довољно аспект превести с неизмењеном класом Point
- Извршењем наредби:

```
Point p = new Point();

p.setX(10);
p.setY(p.getX());
```

- добија се следећи излаз:

*Moving a point.*
*--Point moved.*
*Reading a point.*
*--Point read.*
*Moving a point.*
*--Point moved.*

# Пример: праћење распона вредности

- Можемо и да пратимо да ли су координате у оквиру одређеног распона вредности

```
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 izvan raspona:" + x);
    }
    before(int y): call(void Point.setY(..)) && args(y) {
        if (!yRange(y))
            System.out.println("Y izvan raspona:" + y);
    }
}
```

## Пример: блокирање извршења кода на основу вредности

- Уколико је координата изван распона дозвољених вредности, можемо да блокирамо даљи рад с њом

```
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 izvan raspona:" + x);
        else
            proceed(x);
    }
    . . . // slicno za y
}
```

## Пример: управљање на основу вредности

- Можемо чак и да коригујемо вредност координате

```
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);
    }
    . . . // slicno za y
}
```

# Преглед

1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
   - Модел тачака спајања
   - Сложени тачкасти пресеци
   - Вићења
   - Контекст тачке спајања
   - Међутипске декларације
   - Инстанцијација аспеката
   - Апстрактни аспекти
4. Узори дизајна аспектно-оријентисаног програмирања
   - Аспектно-оријентисана имплементација узора Observer
   - Узор Worker Object Creation
5. Сумаризација

## Раздвајање интереса

- Separation of concerns[1]

  *focussing one's attention on some aspect*

- Интерес (concern) је ствар којој посвећујемо пажњу
- Раздвајање интереса значи да се у једном тренутку посвећује сва пажња једном интересу
- У програмирању се тежи што бољој *локализацији* интереса
- Један од начина је *модуларна декомпозиција*

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

# Модуларна декомпозиција

- Модуларне јединице: модули, компоненте, објекти, функције/процедуре итд.
- Хијерархијски организиване
- Циљ је висока кохезија унутар модуларне јединице, а слаба спрега с осталим модуларним јединицама

# Пресецајући интереси

- Софтвер моделира стварни свет
- А у стварном свету су ретке потпуне хијерархије
- Долази до спреге међу модуларним јединицама које припадају различитим хијерархијама
- То су интереси које није могуће модуларизовати: *пресецајући интереси* (crosscutting concerns) — означавају се као *аспекти*
- Последица је *преплетен* (tangled) и *разбацан* (scattered) код
- Ово се одиграва истовремено: два погледа на тај исти проблем

# Преплетеност кода

- Code tangling
- Различити интереси помешани у оквиру једног модула
- Нпр. методе апликативне логике садрже позиве метода за логирање, безбедност, перзистенцију и сл.

# Разбацаност кода

- Code scattering
- Испољава се на два начина:
  1. Један интерес се понавља у различитим модулима
     - Нпр. логирање је присутно у методама класа које уопште нису повезане
  2. Делови једног интереса су разбацани по разним модулима
     - Нпр. ауторизација обухвата идентификациују корисника, менаџмент права, контролу приступа итд.
     - Иако представљају логичку целину, сваки од ових делова је потребан — и имплементиран — у другом модулу

# Проблем пресецајућих интереса

- Проблем пресецајућих интереса је био примећен пре свега на нивоу кода
  - Такав код се тешко одржава
  - Промена у пресецајућем интересу често изискује промене на свим местима на којима је примењен
  - Сличан проблем настаје и приликом одстрањивања интереса (нпр. искључење логирања) или додавања новог интереса
- Али овај проблем се испољава и на нивоу дизајна, анализе, па чак и у спецификацији захтева
- Ивар Јакобсон је показао да случајеви коришћења представљају аспектно-оријентисану декомпозицију и да се као такви могу модуларизовати путем аспектно-оријентисаног програмирања[2]

[2] I. Jacobson and P. W. Ng. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2004.

# Аспектно-оријентисани приступ пресецајућим интересима

- Аспектно-оријентисано програмирање (aspect-oriented programming, AOP) доноси нове језичке конструкције којима омогућава модуларизацију пресецајућих интереса
- Аспектно-оријентисано програмирање надовезује на објектно-оријентисано програмирање, али примењиво је на све приступе засноване на тзв. уопштеним процедурама (generalized procedures)

# Основни смерови аспектно-оријентисаног програмирања

- PARC AOP (AspectJ)
  - Аспекти као модули пресецајућих интереса
- Субјектно-оријентисано програмирање (Hyper/J)[3]
  - Композиција субјеката као различитих погледа на елементе система
- Композициони филтери (Sina/st)[4]
  - Композиције филтера који филтрирају поруке послате објектима
- Адаптивно програмирање (DemeterJ)[5]
  - Адаптивне стратегије прелазака графом класа

---

[3] http://www.research.ibm.com/hyperspace/

[4] http://trese.cs.utwente.nl/composition_filters

[5] http://www.ccs.neu.edu/research/demeter/

# PARC AOP

- Ова четири приступа аспектно-оријентисана приступа представљају основу за новонастајуће приступе и језике
- PARC AOP је најутицајнији међу њима
- PARC је развио језик AspectJ
- У садшње време постоје десетине аспектно-оријентисаних језика заснованих пре свега на приступу PARC AOP
  - нпр. AspectC++, CeasarJ, AspectS (заснован на језику Smalltalk)...
- Постоје и имплементације аспектно-оријентисаних проширења типа PARC AOP и у објектно-оријентисаним оквирима (frameworks) као што је Spring
- Погледаћемо ближе језик AspectJ

# Преглед

1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
   - Модел тачака спајања
   - Сложени тачкасти пресеци
   - Вијења
   - Контекст тачке спајања
   - Међутипске декларације
   - Инстанцијација аспеката
   - Апстрактни аспекти
4. Узори дизајна аспектно-оријентисаног програмирања
   - Аспектно-оријентисана имплементација узора Observer
   - Узор Worker Object Creation
5. Сумаризација

# Развој језика AspectJ

- Аспектно-оријентисано проширење језика Java
- Програмски језик опште намене
- Отворен развој — од 1997.[6]
- Од 2002. на eclipse.org[7]
- Верзија 1.0 се појавила крајем 2001.
- AspectJ прати развој језика Java — садашња верзија (AspectJ 6, верзија 1.6.2) је заснована на верзији Java 6
- Развојна окружења — подршка за Eclipse, JBuilder и NetBeans
- Користи се у пракси

[6] http://www.parc.com/research/csl/projects/aspectj/

[7] http://eclipse.org/aspectj/

# Аспектно-оријентисане конструкције у језику AspectJ

- Основна конструкција је *аспект* (aspect)
- Аспекти модификују извршавање програма у тачкама спајања
  - Ова места су одређена *тачкастим пресецима* (pointcuts)
  - Модификације се исказују помоћу *вићења* (advices)
  - Вићење се извршава пре, после или уместо тачке спајања
- Аспекти могу и да уводе нове елементе у постојеће структуре и везе међу њима помоћу *међутипских декларација* (inter-type declarations)
  - У класу, интерфејс или аспект је могуће увести нови елемент, али нпр. и дефинисати наслеђивање међу њима
- Инстанце аспеката се стварају аутоматски

# Превод у језику AspectJ

- *Ткање* (weaving) аспеката се реализује директно приликом превода (преводилац ajc)
- Резултат је Java бајткод — преведени програм се извршава на виртуелној машини језика Java (JVM)
- Сваки програм у језику Java је важећи AspectJ програм
- Разлика између преводилаца ajc и javac: преводиоцу ajc треба увек задати све фајлове

# Тачке спајања

- Тачка спајања — добро дефинисана тачка у извршавању програма
- У тачкама спајања је могуће утицати на извршавање програма
- Експониране тачке спајања — тачке спајања које су доступне у датом програмском језику
  - Позив методе је тачка спајања у језику AspectJ
  - Петља **for** није тачка спајања у језику AspectJ
- Тачке спајања имају *контекст*
  - Контекст позива методе су објект који позвао методу, циљни објектат, аргументи и повратна вредност

# Експониране тачке спајања у језику AspectJ (1)

- Методе и конструктори
  - Позив м методе или конструктора
  - Извршење методе или конструктора
- Приступ атрибутима
  - Читање атрибута
  - Запис атрибута
- Обрада изузетака
  - Извршење блока обраде изузетка

# Експониране тачке спајања у језику AspectJ (2)

- Иницијализација класа и објеката
  - Извршење блока статичке иницијализације
  - Извршење иницијализације објекта
  - Извршење прединицијализације објекта
- Извршење виђења
  - Извршење свих виђења у програму

# Тачкасти пресеци

- Тачкама спајања се не приступа појединачно него путем тачкастих пресека који их садрже
- Тачкасти пресек представља скуп тачака спајања
- Тачкасти пресеци се дефинишу помоћу примитивних (елементарних) тачкастих пресека
- Тачкасти пресек који обухвата позиве метода класе Point чије име почиње на set без повратне вредности:

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

- Помоћу логичког оператора *или* можемо да укључимо и методе типа get:

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

- Сложеност аспектно-оријентисаног програмирања је пре свега у дефинисању одговарајућег тачкастог пресека

# Сигнатуре

- Типови (класе, интерфесји и аспекти) и њихови атрибути, конструктори и атрибути се у тачкастим пресецима задају помоћу својих сигнатура
- Могуће је задати и само један елемент:
  - **call**(**void** Point.setX())
  - **within**(Point)
  - **set**(**private int** Point.y)
- Али најчешће се користе сигнатуре са знаковима замене

# Сигнатуре са знаковима замене (1)

- ∗ замењује име или део имена типа, методе или атрибута
  - i∗ P∗.get∗() — све методе с именом који почиње на g у свим типовима чије име почиње на P, а који враћају вредност типа чије име почиње на i
- .. у случају методе замењује листу аргумената или њен део:
  - void Point.**set**∗(..) — све методе void Point.**set**∗()) с било којим бројем аргумената
- .. у случају пакета замењује потпакете (и директне, и индиректне)
  - java..∗ — сви типови у оквиру пакета java
- + замењује подтипове датог типа
  - Point+ — класа Point и све њене поткласе

# Сигнатуре са знаковима замене (2)

- Ако у сигнатури методе није наведен модификатор (модификатор приступа, **abstract**, **static** и **final**), сигнатура обухвата све модификације методе
- Могуће је задати и негацију модификатора и тип повратне вредности
  - !**final** !**public** !**protected** !**static** **int** MyClass.m*()
  - !**private** !**public** !**protected** * MyClass.*(..)
- Негацију је могуће применити и на тип повратне вредности
  - !**int** MyClass.m*()

# Примитивни тачкасти пресеци

- Примитивни тачкасти пресеци одговарају експонираним типовима тачака спајања
- AspectJ обухвата више од двадесет примитивних тачкастих пресека (не рачунајући анотационе верзије)
- Основне групе тачкастих пресека
    - позиви и извршавање метода и конструктора
    - приступ атрибутима
    - иницијализација
    - обрада изузетака
    - тачкасти пресеци засновани на току управљања
    - тачкасти пресеци засновани на лексикалној структури
    - тачкасти пресеци извршних објеката
    - тачкасти пресеци аргумената
    - извршење виђења
    - условни тачкасти пресек
    - анотациони тачкасти пресеци
- Погледаћемо ближе најчешће коришћене тачкасте пресеке

# Позиви и извршавање метода и конструктора

**call**(method_signature)
**call**(constructor_signature)
**execution**(method_signature)
**execution**(constructor_signature)

- * Point+.*(..) — све методе класе Point и њених поткласа
- Point.**new**(..) — сви конструктори класе Point
- * *.**set***(..) **throws** InvalidCoordinatesException — све методе чије име почиње на set, а које избацују изузетак InvalidCoordinatesException

# Тачкасти пресеци засновани на току управљања (1)

**cflow**(**pointcut**)
**cflowbelow**(**pointcut**)

- Обухватају све тачке спајања у току извршавања датог тачкастог пресека
- У случају **cflowbelow**() изостављају се тачке спајања тачкастог пресека који се прати

# Тачкасти пресеци засновани на току управљања (2)

- Применимо следеће виђење на већ наведену репрезентацију тачке:

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

- Излаз ће бити следећи:

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

- Кад бисмо применили **cflowbelow**(), излаз не би садржавао први ред

# Тачкасти пресеци засновани на лексикалној структури

**within**(type_signature)
**withincode**(method_signature)
**withincode**(constructor_signature)

- **within**(Point+) — тачке спајања у оквиру класе Point и њених подтипова
- !**within**(PointMonitoring) && **call**(* *.*(..)) — позиви свих метода у оквиру аспекта PointMonitoring
  - Идиом за избегавање бесконачне рекурзије примене виђења
- **withincode**(**public static void** Test.main(String[])) — тачке спајања у оквиру методе **public static void** Test.main(String[])

# Тачкасти пресеци извршних објеката

**this**(type)
**target**(type)

- **this**(Point) — тачке спајања у оквиру објекта типа Point или његовог подтипа (тј. типа за који важи **this instanceof** Point == **true**)
- **target**(Point) — тачке спајања у којима је објект над којим је метода позвана типа Point или његовог подтипа
- Тип може да буде представљен и идентификатором објекта (за потребе рада с контекстом)

# Тачкасти пресеци аргумената

**args**(argument_list)

- argument_list је листа аргумената
- Аргумент је представљен:
  - својим типом дефинисаним сигнатуром типа
  - идентификатором објекта (за потребе рада с контекстом)
- **args**(**int**) — све тачке спајања у свим методама чији је један аргумент типа **int**

# Дефинисање сложених тачкастих пресека

- Помоћу примитивних тачкастих пресека и логичких оператора *и* (&&), *или* (||) и *не* (!)
- !**within**(PointMonitoring) && **call**(∗ ∗.∗(..)) — позиви свих метода осим позива у оквиру аспекта PointMonitoring
- **call**(∗ Point.**set**∗(..)) || **call**(∗ Point.get∗()) — позиви свих метода ∗ Point.**set**∗(..)) или ∗ Point.get∗())

# Именовани тачкасти пресеци (1)

- Тачкасте пресеке је могуће именовати и користити даље
  - као тачкасте пресеке над којима се извршава виђење
  - у дефиницијама других сложених тачкастих пресека

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

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

- Именовани тачкасти пресеци се наслеђују

# Оквирна синтакса виђења

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

- Виђења дефинишу активности (advice_body) које треба извршити у вези са обухваћеним тачкама спајања
  - Листа аргумената виђења (argument_list) се користи за пренос контекста
  - Тачкасти пресек виђења (**pointcut**) може да буде именован или неименован
- Постоје три основна типа виђења (advice_type):
  - **before**() — извршава се пре тачке спајања
  - **after**() — извршава се после тачке спајања
  - **around**() — извршава се уместо тачке спајања
- Погледајмо детаљније виђење **around**

# Вђење around (1)

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

- Извршава се уместо обухваћене тачке спајања
- Извршавање саме тачке спајања је могуће изазвати помоћу клаузуле proceed()

# Вићење around (2)

- Тип повратне вредности вићења **around**() мора да буде дефинисан (return_value)
- Може да буде **void**
- Ако се наведе Object, AspectJ ће обезбедити промену типа (и код примитивних типова)
- Тип Object мора да се употреби уколико тачкасти пресек вићења обухвата тачке спајања с различитим типовима повратне вредности
- Object укључује и тип **void**
- Вићење **around** нема велики смисао без употребе контекста тачке спајања

# Обухватање контекста тачке спајања

- Контекст тачке спајања представља вредности елемената повезаних с њом у тренутку њеног обухватања
- Нпр. контекст позива методе се састоји од објекта који је методу позвао, циљног објекта, аргумената методе, повратне вредности и изузетка
- Контекст се обухвата превасходно путем тачкастих пресека **this**(), **target**() и **args**()
- Део контекста се обухвата помоћу виђења **after**() **returning** и **after**() **throwing**
- До комплетног контекста је могуће доћи и путем рефлективног API језика AspectJ
  - **thisJoinPoint**
  - **thisJoinPointStaticPart**
  - **thisEnclosingJoinPointStaticPart**
- Треба давати предност обухватању контекста тачкастим пресецима

# Пренос контекста тачке спајања у вићење

- Контекст тачке спајања се преноси у вићење слично као аргументи у методу

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

- Вићење (аналогно методи) дефинише тип променљивих контекста (аналогно аргументима методе)
- Код методе се вредности аргумената задају приликом позива
- Вредности променљивих контекста вићење добија из тачкастог пресека
- Потом се могу кориситити у телу вићења (као аргументи у телу методе)

# Пренос контекста с именованим тачкастим пресеком

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

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

# Вредност атрибута

- Вредност коју треба да поприми атрибут обухватамо такође помоћу тачкастог пресека **args**()

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

# Пренос повратне вредности у виђењу after() returning

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

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

# Пренос изузетка у вићењу after() throwing

```
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);
}
```

# Међутипске декларације

- Inter-type declarations (првобитно називане introductions)
- Аспекти могу да у типове (класе, интерфејси и аспекти) уводе нове елементе и везе међу њима
- Врсте међутипских декларација:
    - увођење чланског елемента (атрибута или методе)
    - увођење везе наслеђивања
    - увођење грешке или упозорења приликом превођења
    - увођење анотације
    - умекшавање изузетка (exception softening)

# Пример: рачун — класа Account (1)

- Пример из књиге AspectJ in Action[8]

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

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

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

[8] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003. http://www.manning.com/laddad/

# Пример: рачун — класа Account (2)

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

# Пример: рачун — класа Account (3)

```
...
  public float getBalance() {
    return _balance;
  }

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

# Пример: рачун — класа SavingsAccount

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

# Пример: рачун — увођење чланског елемента (1)

```
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");
    }
  }
}
```

# Пример: рачун — увођење чланског елемента (1)

- Увели смо атрибут _minimumBalance и методу getAvailableBalance() ради праћења минималног остатка
- Модификатори приступа ових елемената важе у односу на аспект који их је увео
- Зашто нисмо ове елементе увели директно у класи?
  - Обраду минималног остатка схватамо као посебан интерес
  - Не желимо да га мешамо с основном функционалношћу рачуна

# Увођење наслеђивања

- Могуће је увести и однос implements, и extends
- При томе се мора држати правила наслеђивања која важе у језику Java
- Увођење интерфејса за означавање ради праћења елемената:

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

- Увођење односа extends интересантно је пре свега из перспективе способности конфигурисања
  - Више класа с истим интерфејсом, али различитим имплементацијама
  - Аспектом одредимо која од ових класа ће се искористити

# Инстанцијација аспеката

- Инстанце аспеката се стварају аутоматски на основу тачкастог пресека
- Не могу се стварати директно, али могуће је регулисати начин инстанцијације:
  - инстанца за цео програм, тј. виртуелну машину (имплицитно) — **issingleton**()
  - инстанца за објект — **perthis**() а **pertarget**()
  - инстанца за ток управљања — **percflow**() а **percflowbelow**()
  - инстанца за тип — **pertypewithin**(type_pattern)
- Синтакса (угласта заграда означава опционалност):

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

- Аспект наслеђује начин инстанцијације и не може да га промени

# Апстрактни аспекти

- Аспекти могу да буду и апстрактни — кључна реч **abstract**
- Слично апстрактним класама, апстрактни аспекти не могу да имају инстанце
- Од апстрактних аспеката могу да наслеђују асптрактни и конкретни аспекти, али од конкретних аспеката више нема наслеђивања
- Аспекти могу да наслеђују и од класа (без обзира на то да ли су апстрактне или не)

# Апстрактни тачкасти пресеци

- Апстрактни аспекти могу да садрже асптрактне тачкасте пресеке — тачкасти пресеци без тела
- Апстрактни аспект може да над апстрактним тачкастим пресецима дефинише вијђења
- Апстрактне тачкасте пресеке је могуће превазићи (override) — конкретизовати — у изведеним аспектима
- Апстрактни тачкасти пресеци су корисни ако знамо да спецификујемо шта треба извршити над скупом тачака спајања, али овај скуп у општем случају не знамо да одредимо

# Пример: апстрактни монитор

```
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*());
}
```

# Преглед

1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
   - Модел тачака спајања
   - Сложени тачкасти пресеци
   - Виђења
   - Контекст тачке спајања
   - Међутипске декларације
   - Инстанцијација аспеката
   - Апстрактни аспекти
4. Узори дизајна аспектно-оријентисаног програмирања
   - Аспектно-оријентисана имплементација узора Observer
   - Узор Worker Object Creation
5. Сумаризација

# Узори

- Patterns
- Порекло узора
  - Christopher Alexander — узори у архитектури
  - Другачији поглед на архитектуру: језик узора
- Hillside Group: K. Auer, G. Booch, R. Johnson, H. Hilerbrand, K. Beck, W. Cunnigham a J. Coplien — 1993[9]
  - Култура узора[10]
- Узор: контекст — силе — конфигурација која их разлаже

  *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.*[11]

- Узори дизајна (design patterns)

[9] http://www.hillside.net/
[10] http://www.comsis.fon.bg.ac.yu/ComSIS/Volume02/InvitedPapers/JamesCoplin.htm
[11] http://www.hillside.net/patterns/definition.html

# Даље следи

1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
   - Модел тачака спајања
   - Сложени тачкасти пресеци
   - Виђења
   - Контекст тачке спајања
   - Међутипске декларације
   - Инстанцијација аспеката
   - Апстрактни аспекти
4. Узори дизајна аспектно-оријентисаног програмирања
   - Аспектно-оријентисана имплементација узора Observer
   - Узор Worker Object Creation
5. Сумаризација

# Аспектно-оријентисане имплементације ОО узора

- Класични објектно-оријентисани узори се могу исказати компактније помоћу аспектно-оријентисаног програмирања
- Hannemann и Kiczales су имплементирали GoF узоре[12] на аспектно-оријентисани начин[13]
- Следи узор Observer на мало другачији начин
- Претходно ћемо погледати његову ОО имплементацију

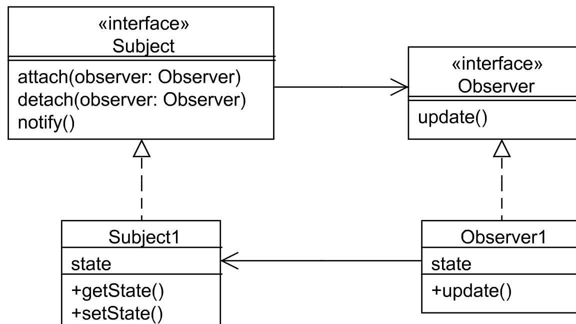
[12] E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Design. Addison Wesley, 1995.

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

# Узор Observer

- Служи за дефинисање зависности стања више објеката (посматрачи) од датог објекта (предмет)
- Погодан за употребу уколико промена у једном објекту изискује измене у другим објектима којих број и тачан тип није познат

# Структура узора Observer



# Пример употребе узора Observer

- ОО имплементација узора Observer на примеру температурних сензора
- Температурни сензори мере температуру
- Измерену температуру приказују на различитим дисплејима: дигиталним, аналогним, с опсегом...
- На један температурни сензор може истовремено да буде прикључених више дисплеја

# Сензор и дисплеј

```
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();
}
```

## Сензор температуре људског тела

```
public class HumanTempSensor implements TempSensor { // a subject
  private ArrayList 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();
    }
  }
}
```

# Дигитални дисплеј

```
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();
  }
}
```

# Дисплеј с опсегом (1)

```
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;
    }
    . . .
```

# Дисплеј с опсегом (2)

```
    . . .
    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();
    }
}
```

# Примена

```
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
    }
}
```

# Аспектно-оријентисана имплементација узора Observer

- Ствари које се тичу техничке реализације узора Observer не би требало мешати с апликационом логиком:
  - делови интерфејса TempDisplay и TempSensor и њихова имплементација
  - логика спајања дисплеја са сензором
  - обнављање дисплеја приликом промене вредности у сензору

# Сензор

```
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; }
}
```

# Дисплеј

```
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();
  }
}
```

# Дисплеј с опсегом (1)

```
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;
    }
    ...
```

# Дисплеј с опсегом (2)

```
...
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
```

# Логику узора Observer обезбеђује аспект

```
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();
    }
}
```

# Радни објект

- Понекад је потребно да пренесемо референцију методе
- У језику Java се за то користе тзв. радни објекти (worker objects)
- Такав објект имплементира методу коју треба пренети под стандардним називом (користи се интерфејс Runnable и метода run())
- Пошаљимо нпр. методи m() радни објект

```
m(new Runnable() {
    public void run() {
        . . .
    }
});
```

- Метода m() ће га извршити

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

# Узор Worker Object Creation

- Треба да обезбедимо да сваки позив унутар одређених метода буде извршен путем радног објекта
- Пример: позиви који актуализују већ реализован кориснички интерфејс заснован на оквиру Swing морају да се изврше путем диспечинг нити оквира Swing (Swing dispatching thread)
- У језику Java сваки такав позив морамо да идентификујемо и обавијемо у конструкцију сличну следећој:

```
Event-thread.invokeLater(new Runnable() {
    public void run() {
        . . .
    }
});
```

# Схема узора

- У аспектно-оријентисаном програмирању можемо да обухватимо позиве свих таквих метода и да их обавијемо у радни објект:

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

# Пример примене (1)

- Пример — Swing dispatching thread
- Предметни код би могао да изгледа нпр. овако (комплетан пример је из књиге AspectJ in Action):

```
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);
    }
}
```

# Пример примене (2)

- Апстрактни аспект имплементира опште решење — осим тачкастих пресека

```
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 *.*(..));
    . . .
```

# Пример примене (3)

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

# Пример примене (4)

```
    ...
    Object around(): routedMethods()
        && (!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();
    }
}
```

# Пример примене (5)

- Конкретан аспект дефинише тачкасте пресеке према контексту примене

```
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+.*(..))
        /* || ... */;
}
```

# Преглед

1. Уместо увода
2. Принципи аспектно-оријентисаног програмирања
3. Основне конструкције програмског језика AspectJ
   - Модел тачака спајања
   - Сложени тачкасти пресеци
   - Виђења
   - Контекст тачке спајања
   - Међутипске декларације
   - Инстанцијација аспеката
   - Апстрактни аспекти
4. Узори дизајна аспектно-оријентисаног програмирања
   - Аспектно-оријентисана имплементација узора Observer
   - Узор Worker Object Creation
5. Сумаризација

# Сумаризација (1)

- Пресецајући интереси (crosscutting concerns) чине код преплетеним и разбацаним
- Аспектно-оријентисано програмирање омогућава модуларизацију пресецајућих интереса
- AspectJ — најзаступљенији аспектно-оријентисани програмски језик
  - PARC AOP
  - Аспекти: тачкасти пресеци (pointcut), виђења (advice) над њима и међутипске декларације
- Узори дизајна у аспектно-оријентисаном програмирању — Observer и Worker Object Creation

# Сумаризација (2)

- Аспектно-оријентисани развој софтвера је предметом интензивног истраживања
- AOSD-Europe[14] — истраживачки конзорцијум академских и производних организација настао као европска мрежа ексцелентности
- Све је заступљенији у пракси
- Уврштен је у листу десет технологија које највише обећавају у MIT Technology Review[15]
- IBM га је означио за виталан за опстанак IBM Software Group

[14] http://www.aosd-europe.net/
[15] 10 emerging technologies that will change the world. *Technology Review*, Jan.–Feb. 2001.

# Шта даље?

- Инсталирајте AspectJ[16] и испробајте неке од примера
- На сајту језика AspectJ ћете наћи и приручник с примерима
- Погледајте неку од књига о аспектно-оријентисаном програмирању
  - Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.
  - Russell Miles. *AspectJ Cookbook*. O'Reilly, 2004.
  - Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison Wesley, 2004.

[16] http://eclipse.org/aspectj/