Refactoring OO Source Code to Aspects

Miguel Pessoa Monteiro
Escola Superior de Tecnologia
Instituto Politécnico de Castelo Branco
PORTUGAL

AO Refactoring

Whenever a better way of doing things comes up, style rules change:

- Structured Programming
  - 'Gotos' Considered Harmful
- OOP
  - 'Switch' Statements Considered a Code Smell
- AOP
  - ?

Refactoring

Refactorings:
"Behaviour-preserving transformations of the source code"

- Adding new functionality is not refactoring
- Optimisation is not refactoring
- Changing code that does not compile is not refactoring
  (what would be the behaviour?)

Refactoring

Refactorings remove *Bad Smells in the Code*
i.e., potential problems or flaws

- Some will be strong, some will be subtler
- Some smells are obvious, some aren't
- Some smells mask other problems
- Some smells go away unexpectedly when we fix something else

Refactoring

Examples of *code smells*:

- Duplicated Code, Large Class, Lazy Class, Long Method, Long Parameter List, Primitive Obsession, Speculative Generality, Temporary Field, Inappropriate Intimacy, Data Class, Refused Bequest, Comments, ...
- Frequent cause:
  the *paradigm shift problem*

Refactoring

Example:
When smelling *Long Method*, programmers consider using one or several of the following:

- *Extract Method*,
  Replace Temp with Query,
  Replace Method with Method Object,
  Decompose Conditional.

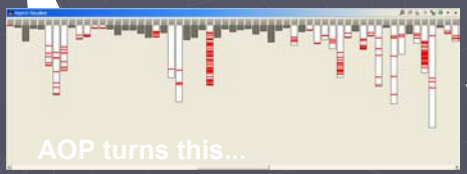
Refactoring

```
public void add(Object element) {
   if(!readOnly) {
      int newSize = size + 1;
      if(newSize > elements.length) {
         Object[] newElements = new Object[elemnts.length + 10];
         for(int i=0; i<size; i++)
            newElements[i] = elements[i];
         elements = newElements;
      }
      elements[size++] = element;
   }
}
```

```
public void add(Object element) {
   if(readOnly)
      return;
   if(atCapacity())
      grow();
   addElement(element);
}
```


Aspect-Oriented Programming

- Modularisation of crosscutting concerns
- e.g. *middleware* concerns
  - Persistence, security, performance tuning, logging, distribution, synchronisation, transaction support,...



Aspect-Oriented Programming

Defining characteristics of AOP [Filman & Friedman 2000]:

- Quantification
  - *In programs P, whenever condition C arises, perform action A*
- Obliviousness
  - AOP can be applied to programs oblivious to the quantification by the aspects

Aspect-Oriented Programming

Fundamental concept of AOP: joinpoint

- Any identifiable execution point in a software system.
- Examples:
  - Access to object fields (for reading and for writing)
  - Execution of a method (or constructor, or exception handler)

Aspect-Oriented Programming

Important constructs of AspectJ (for the purposes of this talk):

- Pointcuts – defines sets of joinpoints (a *single* joinpoint is not crosscutting)
- Advice – executes upon reaching a joinpoint
  - Can execute before, after or instead of the original joinpoint

Aspect-Oriented Programming

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
     this.p1 = p1;
     Display.update(this);
  }
  void setP2(Point p2) {
     this.p2 = p2;
     Display.update(this);
  }
  void moveBy(int dx, int dy) { ... }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
     this.x = x;
     Display.update(this);
  }
  void setY(int y) {
     this.y = y;
     Display.update(this);
  }
  void moveBy(int dx, int dy) { ... }
}
```

Aspect-Oriented Programming

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
     this.p1 = p1;

  }
  void setP2(Point p2) {
     this.p2 = p2;

  }
  void moveBy(int dx, int dy) { ... }
}
```

```
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
     this.x = x;

  }
  void setY(int y) {
     this.y = y;

  }
  void moveBy(int dx, int dy) { ... }
}
```


Aspect-Oriented Programming

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
     this.p1 = p1;

  }
  void setP2(Point p2) {
     this.p2 = p2;

  }
  void moveBy(int dx, int dy) { ... }
}
```

```
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
     this.x = x;

  }
  void setY(int y) {
     this.y = y;

  }
  void moveBy(int dx, int dy) { ... }
}
```

```
aspect DisplayUpdating {

  pointcut move(FigureElement figElt):
     target(figElt) &&
     (call(void FigureElement.moveBy(int, int) ||
      call(void Line.setP1(Point))          ||
      call(void Line.setP2(Point))          ||
      call(void Point.setX(int))            ||
      call(void Point.setY(int)));


  after(FigureElement fe) returning: move(fe) {
     Display.update(fe);
  }
}
```

Aspect-Oriented Programming

Important constructs of AspectJ

- Inter-type declarations – defines extra methods and fields in target classes (private means private to the aspect)
- Declared within the aspect, but owned by target classes
- Aspects can also hold their own state and behaviour.

Aspect-Oriented Programming

Inter-type declarations:

- like member declarations, but with a *TargetType*

```
long            field = 37;
void            method() {
   //some logic...
}
```

Aspect-Oriented Programming

Inter-type declarations:

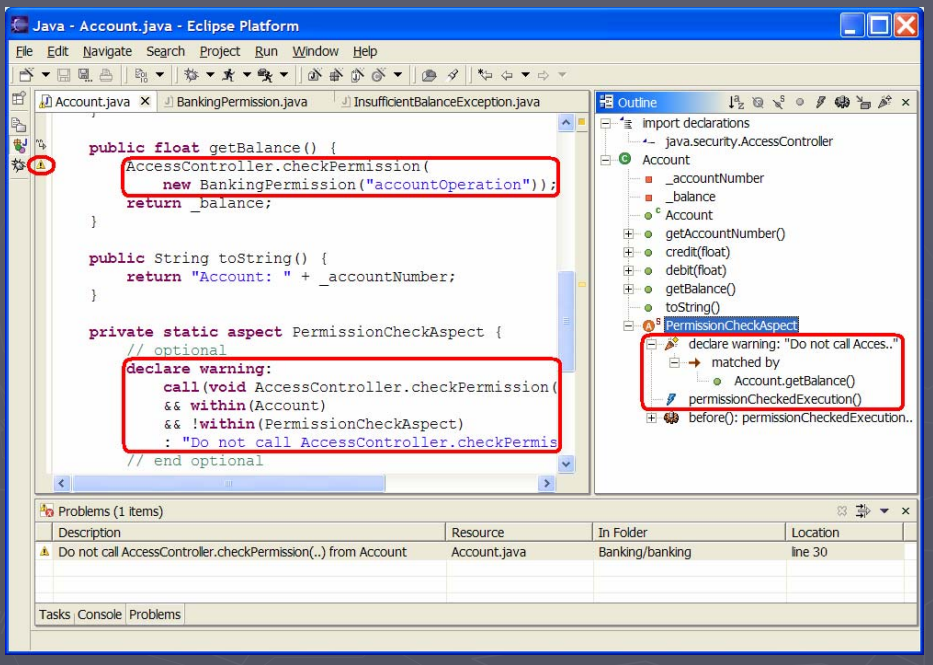
- like member declarations, but with a *TargetType*

```
long TargetType.field = 37;
void TargetType.method() {
   //some logic...
}
```

Aspect-Oriented Programming

Important constructs of AspectJ

- declare error/warning clauses
  - `declare error: Pointcut: String;`
  - `declare warning: Pointcut: String;`
- Useful for
  - Enforce policies,
  - Detect trouble spots, ...



Aspect-Oriented Programming

Can aspects quantify over *any* code base?

- No, because
  not all joinpoints are supported
  (nor would that be desirable).
- Examples of *fragile* joinpoints:
  - Line numbers
  - Access to local variables

AO Refactoring

With AOP do we still need refactoring?

Yes, because

- AspectJ's quantification capabilities are not limitless
- Also same reasons as with OO:
  - Requirements change over time
  - Environments change over time
  - Programs are often coded in bad style

AO Refactoring

Prerequisites for applying AO refactoring

- Good unit test coverage (of course!)
- OO base code in good style (e.g. no OO code smells):
  - Small methods with meaningful names
  - Parameter lists not too long

AO Refactoring

When should we refactor to aspects?

- When the code stinks (in the light of AOP)
- Currently under active research:
  - New aspect-oriented code smells
  - New aspect-oriented refactorings that remove such smells

AO Code Smells

Traditional OO smells in the light of AOP

- Divergent Change [Fowler]
  - "One change that alters many classes"
  - ➔ Code scattering
- Shotgun Surgery [Fowler]
  Solution Sprawl [Kerievsky]
  - "One class that suffers many kinds of change"
  - ➔ Code tangling
- Remove smells using *Extract Feature into Aspect*

AO Code Smells

Double Personality

- Class with superimposed roles
- Class implementing one or more interfaces

▶ Smell affects *one* class or *several*?

- One class affected: remove smell using *Split Class into Aspect and Interface*
- Multiple classes: use *Extract Feature into Aspect*

AO Code Smells

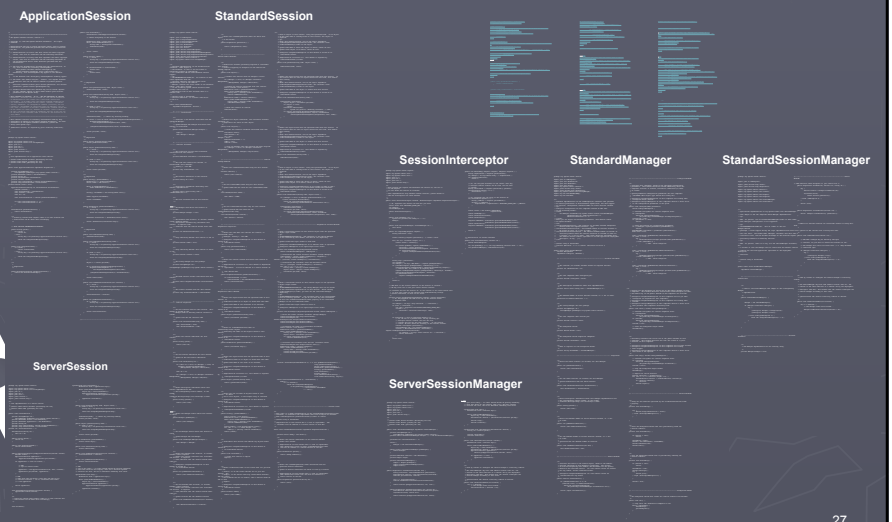
Aspect Laziness

- Aspects do not hold own state and behaviour. Instead pushes burden to classes, through inter-type declarations

▶ You see the smell when:

- State/behaviour is needed by only a subset of all instances
- State/behaviour needed only in some phases
- Multiple instances of state/behaviour needed

Aspect-Oriented Programming



AOP Refactorings

- Extract Feature into Aspect
- Move Method from Class to Inter-type
- Move Field from Class to Inter-type
- Extract Fragment into Advice
- Change Abstract Class to Interface
- Split Abstract Class into Aspect and Interface
- Extract Inner Class to Standalone
- Inline Class within Aspect
- Inline Interface within Aspect
- Replace Implements with Declare Parents

- Tidy Up Internal Aspect Structure
- Generalize Target Type with Marker Interface
- Replace Inter-type Field with Aspect Map
- Replace Inter-type Method with Aspect Method
- Extend Marker Interface with Signature
- Introduce Aspect Protection

- Extract Superaspect
- Pull Up Advice
- Pull Up Declare Parents
- Pull Up Marker Interface
- Pull Up Pointcut
- Pull Up Inter-type Declaration
- Push Down Advice
- Push Down Declare Parents
- Push Down Marker Interface
- Push Down Pointcut
- Push Down Inter-type Declaration

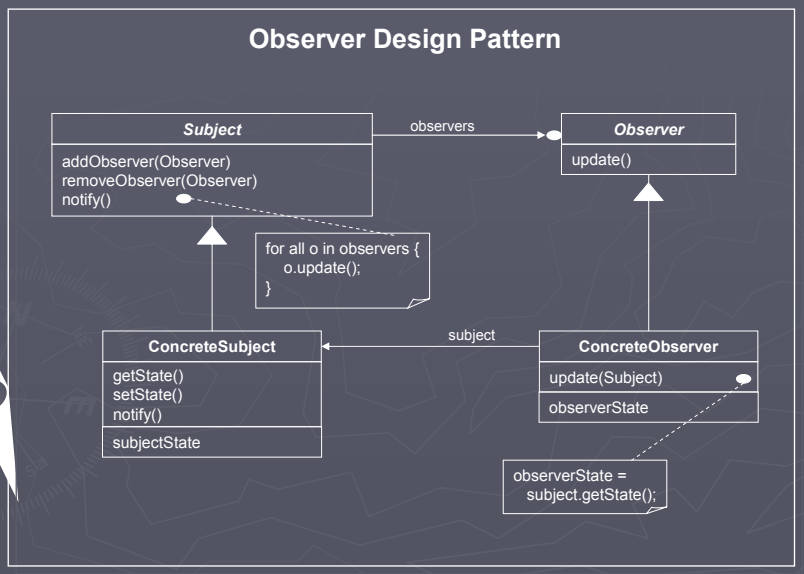
Refactoring Strategy

1. Move all scattered elements to the aspect
2. Tidy up the internal structure of the extracted aspect
3. In case there are several similar Aspects,
   factor out common code to a superaspect

Observer Design Pattern

- Intent:
  "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
  [Gamma95]

Observer Design Pattern



Refactoring Example

- Starting point
  "Flower" example from:
  Bruce Eckel,
  Thinking in Patterns, revision 0.9, May 20, 2003.
  64.78.49.204/TIPatterns-0.9.zip
- Ending point:
  AspectJ implementation of Observer
  Jan Hannemann & Gregor Kiczales,
  www.cs.ubc.ca/~jan/AODPs/

Subject Class (clean)

```
public class Flower {
    private boolean _isOpen;

    public Flower() {
        _isOpen = false;
    }
    public boolean isOpen() {
        return _isOpen;
    }
    private void setIsOpen(boolean newValue) {
        _isOpen = newValue;
    }
    public void open() { // Opens its petals
        System.out.println("Flower open.");
        setIsOpen(true);
    }
    public void close() { // Closes its petals
        System.out.println("Flower close.");
        setIsOpen(false);
    }
}
```


The *real* Subject Class

Smell: *Double Personality*

```
public class Flower {
    private boolean isOpen;
    private OpenNotifier oNotify = new OpenNotifier();
    private CloseNotifier cNotify = new CloseNotifier();

    public Flower() {
        isOpen = false;
    }
    public void open() { // Opens its petals
        System.out.println("Flower open.");
        isOpen = true;
        oNotify.notifyObservers();
        cNotify.open();
    }
    public void close() { // Closes its petals
        System.out.println("Flower close.");
        isOpen = false;
        cNotify.notifyObservers();
        oNotify.close();
    }
    public Observable opening() {
        return oNotify;
    }
    public Observable closing() {
        return cNotify;
    }
    private class OpenNotifier extends Observable {
        //...
    }
    private class CloseNotifier extends Observable {
        //...
    }
}
```

Subject Class (clean)

```
public class Hummingbird {
    private String name;

    public Hummingbird(String nm) {
        name = nm;
    }
    public void breakfastTime() {
        System.out.println("Hummingbird " + name + "'s breakfast time!");
    }
    public void bedtimeSleep() {
        System.out.println("Hummingbird " + name + "'s bed time!");
    }
}
```


Subject Class (clean)

```
public class Bee {
    private String name;

    public Bee(String nm) {
        name = nm;
    }
    public void breakfastTime() {
        System.out.println("Bee " + name + "'s breakfast time!");
    }
    public void bedtimeSleep() {
        System.out.println( "Bee " + name + "'s bed time!");
    }
}
```

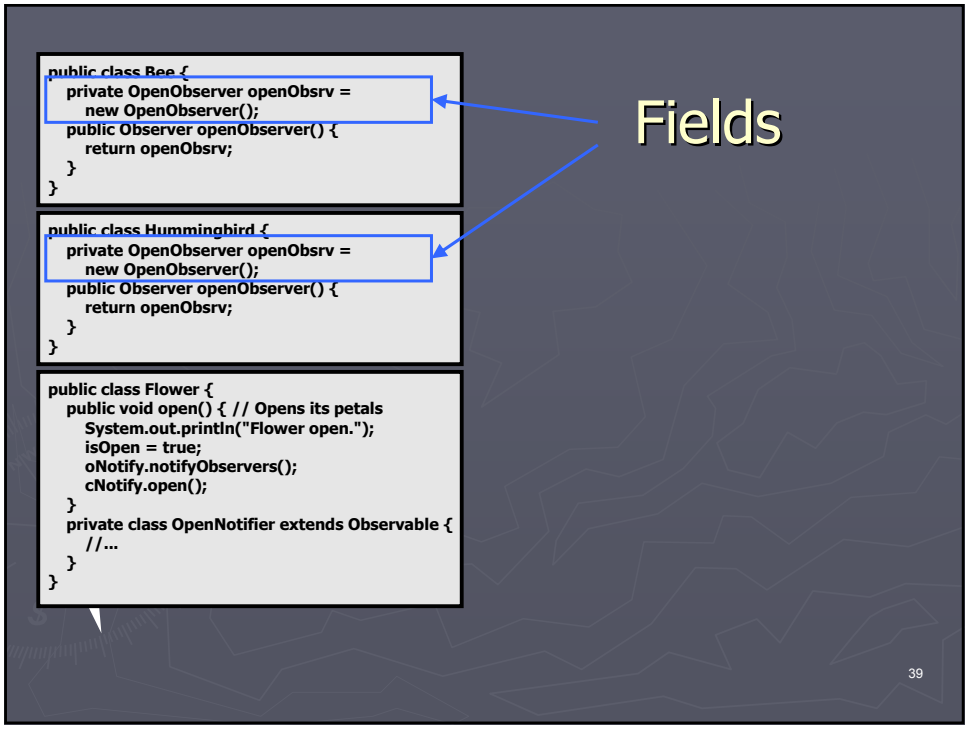
```
public class Bee {
    private String name;
    private OpenObserver openObsrv = new OpenObserver();
    private CloseObserver closeObsrv = new CloseObserver();

    public Bee(String nm) {
        name = nm;
    }
    // An inner class for observing openings:
    private class OpenObserver implements Observer {
        public void update(Observable ob, Object a) {
            System.out.println("Bee " + name + "'s breakfast time!");
        }
    }
    // Another inner class for closings:
    private class CloseObserver implements Observer {
        public void update(Observable ob, Object a) {
            System.out.println("Bee " + name + "'s bed time!");
        }
    }
    public Observer openObserver() {
        return openObsrv;
    }
    public Observer closeObserver() {
        return closeObsrv;
    }
}
```

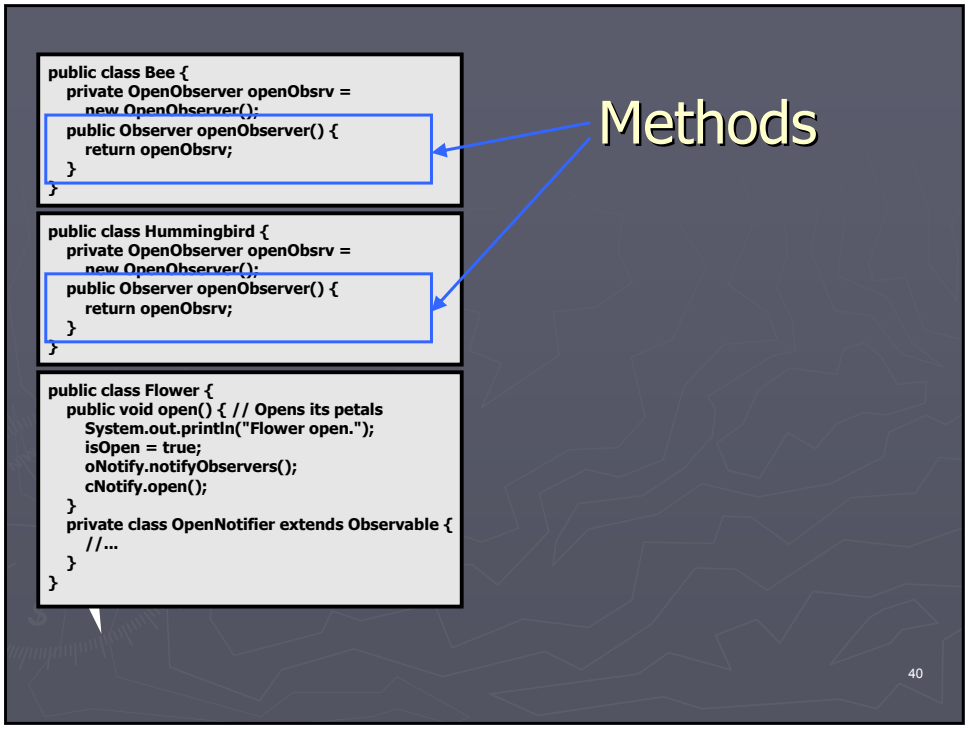

Refactoring Example

```
public class Bee {
  private OpenObserver openObsrv =
    new OpenObserver();
  public Observer openObserver() {
    return openObsrv;
  }
}
```

```
public class Hummingbird {
  private OpenObserver openObsrv =
    new OpenObserver();
  public Observer openObserver() {
    return openObsrv;
  }
}
```

```
public class Flower {
  public void open() { // Opens its petals
    System.out.println("Flower open.");
    isOpen = true;
    oNotify.notifyObservers();
    cNotify.open();
  }
  private class OpenNotifier extends Observable {
    //...
  }
}
```


public class Bee {
private OpenObserver openObsrv =
new OpenObserver();
public Observer openObserver() {
return openObsrv;
}
}
public class Hummingbird {
private OpenObserver openObsrv =
new OpenObserver();
public Observer openObserver() {
return openObsrv;
}
}
public class Flower {
public void open() { // Opens its petals
System.out.println("Flower open.");
isOpen = true;
oNotify.notifyObservers();
cNotify.open();
}
private class OpenNotifier extends Observable {
//...
}
}
Fields


public class Bee {
private OpenObserver openObsrv =
new OpenObserver();
public Observer openObserver() {
return openObsrv;
}
}
public class Hummingbird {
private OpenObserver openObsrv =
new OpenObserver();
public Observer openObserver() {
return openObsrv;
}
}
public class Flower {
public void open() { // Opens its petals
System.out.println("Flower open.");
isOpen = true;
oNotify.notifyObservers();
cNotify.open();
}
private class OpenNotifier extends Observable {
//...
}
}
Methods

Code Fragment

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```


Inner Class

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```



```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

- Move Field from Class to Inter-type

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
}
```

- Move Method from Class to Inter-type


```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

- Move Field from Class to Inter-type
- Move Method from Class to Inter-type

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
```

- Extract Fragment to Advice


```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public class OpenNotifier extends Observable {
   //...
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
```

- Extract Inner Class to Standalone

```
public class Bee {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Hummingbird {
   private OpenObserver openObsrv =
      new OpenObserver();
   public Observer openObserver() {
      return openObsrv;
   }
}
```

```
public class Flower {
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   private class OpenNotifier extends Observable {
      //...
   }
}
```

```
public class OpenNotifier extends Observable {
   //...
}
```

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
```

- Inline Class within Aspect

- First phase (extraction) completed
- Internal structure still not a good one
- New-found modularity makes it easier to restructure

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
```

Smell:

- Duplicated Code
  - Multiple introductions of the same members
- Remove smell using
  - *Generalize Target Type with Marker Interface*

```
public aspect ObservingOpen {
   private OpenObserver Bee.openObsrv =
      new OpenObserver();
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
```


Remove smell using

- *Generalize Target Type with Marker Interface*
  1. Introduce marker interfaces
  2. Introduce declare parents to bind the marker interfaces to case-specific types
  3. Replace all references to case-specific types with references to the marker interfaces

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   public OpenObserver Hummingbird.openObsrv =
      new OpenObserver();
   public Observer Hummingbird.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;
```

Remove smell using

- *Generalize Target Type with Marker Interface*
  1. Introduce marker interfaces
  2. Introduce declare parents to bind the marker interfaces to case-specific types
  3. Replace all references to case-specific types with references to the marker interfaces

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   public OpenObserver Observer.openObsrv =
      new OpenObserver();
   public Observer Observer.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;
```


Smell:

- Aspect Laziness
- Extra state/behaviour composed statically
- Dynamic composition not possible

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   public OpenObserver Observer.openObsrv =
      new OpenObserver();
   public Observer Observer.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;
```

Remove smell using

- *Replace Inter-type Field with Aspect Map*
  1. Introduce a mapping structure
  2. Introduce the managing logic

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}
   private WeakHashMap
      subject2ObserversMap
         = new  WeakHashMap();
   public OpenObserver Observer.openObsrv =
      new OpenObserver();
   public Observer Observer.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;

   private List getObservers(Subject subject)
   public void addObserver(Subject subject,
                           Observer observer)
```


Remove smell using

- *Replace Inter-type Method with Aspect Method*
  1. Introduce new version of method getting the target object as parameter
  2. Replace calls to inter-type method with calls to aspect method

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}
   private WeakHashMap
      subject2ObserversMap
         = new  WeakHashMap();
   public OpenObserver Observer.openObsrv =
      new OpenObserver();
   public Observer Observer.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject flower) returning :
         flowerOpen(flower) {
      notifyObservers(flower);
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;

   private List getObservers(Subject subject)
   public void addObserver(Subject subject,
                           Observer observer)
   public void notifyObservers(Flower flower)
```

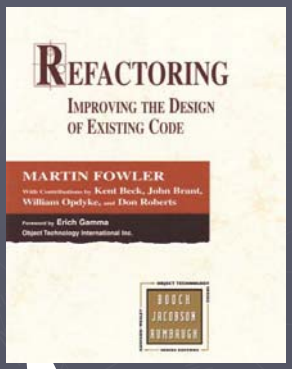
For more details:

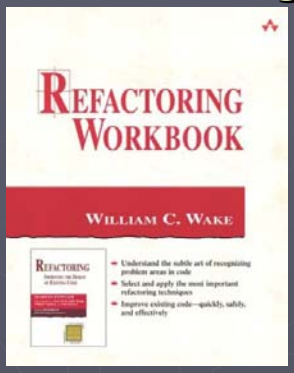
gec.di.uminho.pt/mpm/

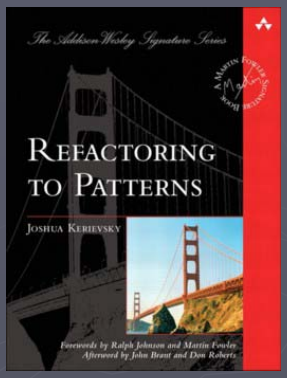
- Description of the Refactorings
- Illustrative Example

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}
   private WeakHashMap
      subject2ObserversMap
         = new  WeakHashMap();
   public OpenObserver Observer.openObsrv =
      new OpenObserver();
   public Observer Observer.openObserver() {
      return openObsrv;
   }
   pointcut flowerOpen(Subject flower):
      execution(void open()) && this(flower);
   after(Subject flower) returning :
         flowerOpen(flower) {
      notifyObservers(flower);
   }
   static class OpenNotifier extends Observable {
      //...
   }
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird)
      implements Observer;

   private List getObservers(Subject subject)
   public void addObserver(Subject subject,
                           Observer observer)
   public void notifyObservers(Flower flower)
```


Refactoring







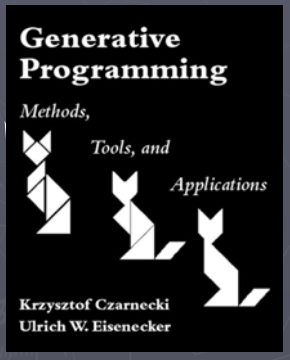
- Refactoring home page - www.refactoring.com/
- Refactoring mailing list at Yahoo groups.yahoo.com/group/refactoring/

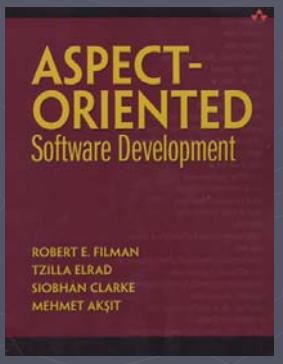
Aspect-Oriented Software Development

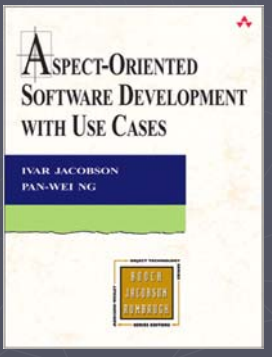
AOSD (a.k.a. Advanced Separation of Concerns):
http://aosd.net/
http://aosd.net/mailman/listinfo/discuss_aosd.net
http://aosd.net/mailman/listinfo/announce_aosd.net

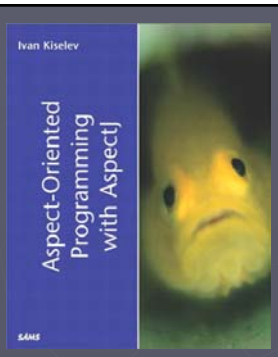
AOP:
https://dev.eclipse.org/mailman/listinfo/aspectj-users
http://dev.eclipse.org/mhonarc/lists/aspectj-dev/maillist.html

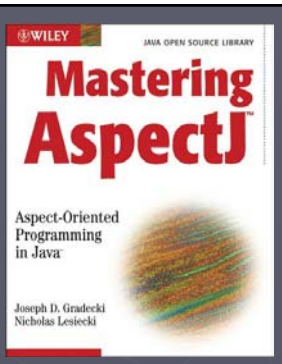
Aspect-Oriented Software Development (AOSD) a.k.a. Advanced Separation of Concerns (ASoC)

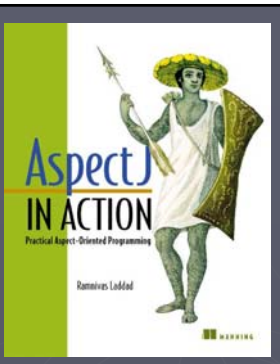


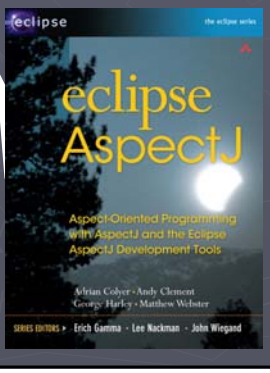


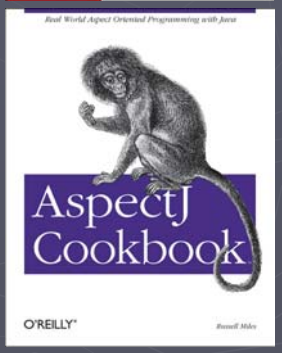












Towards a Catalog of Aspect-Oriented Refactorings

Questions?

Miguel Pessoa Monteiro
Escola Superior de Tecnologia
Instituto Politécnico de Castelo Branco
PORTUGAL