Caesar

Ben Rister
work by Mira Mezini and Klaus Ostermann

Outline

- Ideal Properties of Aspect Systems
- Cast of Characters
- Motivating Example
- Analysis

Ideal Properties

- Powerful extensibility
- Easy to write and understand
- Dynamic configurability
- High reusability

Cast of Characters: FOA

- "Feature-oriented approaches"
- Subclasses with composition

```
class OriginalClass { ... }

refines class OriginalClass {
  List newListData;
  public void newMethod(Object o) { ... }
}
```

Cast of Characters: AspectJ

- See Monday's paper/talk

```
public aspect NewAspect {
  protected pointcut theCut() :
    call(void TheClass.theMethod());

  around(): theCut() { ... }
}
```

Cast of Characters: Caesar

- AspectJ +
  greater aspect structuring +
  dynamic bindings

```
interface AspectInterface {
  interface ObjectInterface {
    provided void someFunction();
    expected void anotherFunction();
  }
}

class AspectImplementation {
  class ObjectInterface {
    void someFunction() { ... }
  }
}
```

Motivating Example

- Subject-Observer pattern: Observer is notified whenever Subject (typically) changes value
  - Subject.addObserver(Observer)
  - Subject.removeObserver(Observer)
  - Observer.notify(Subject)

Power/Ease (FOA)

```
class Figure { ... }
class Screen { ... }

refines class Figure {
  List observers;
  public void addObserver(Figure o) { ... }
  public void removeObserver(Figure o) { ... }
}

refines class Screen {
  public void notify(Screen s) { ... }
}
```

How do we trigger the notify?

Override every modifying function

Power/Ease (FOA)

```
refines class Figure {
  List observers;
  public void addObserver(Figure o) { ... }
  public void removeObserver(Figure o) { ... }

  public void setPoint(Point p) {
    super.setPoint(p);
    notifyObservers();
  }

  public void setLine(Line l) {
    super.setLine(l);
    notifyObservers();
  }

  ...

}
```

Power/Ease (AspectJ)

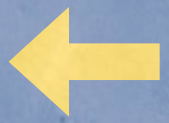
```
public abstract aspect ObserverProtocol {
  protected interface Subject { }
  protected interface Observer { }
  private WeakHashMap perSubjectObservers;
  protected List getObservers(Subject s) {
    if (perSubjectObservers == null)
      perSubjectObservers = new WeakHashMap();
    List observers = (List) perSubjectObservers.get(s);
    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(s, observers);
    }
    return observers;
  }
  public void addObserver(Subject s,Observer o) {
    getObservers(s).add(o);
  }
  public void removeObserver(Subject s,Observer o) {
    getObservers(s).remove(o);
  }
  abstract protected void notifyObserver(Subject s, Observer o);
  abstract protected pointcut subjectChange(Subject s);
  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() )
      notifyObserver(s, ((Observer)iter.next()));
  }
}
```

Power/Ease (AspectJ)

```
public aspect FigureObserver extends ObserverProtocol {
  declare parents: Figure implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    (call(void Figure.setPoint(Color)) ||
     call(void Figure.setLine(Color))
    ) && target(s);

  protected void notifyObserver(Subject s, Observer o) {
    ...
  }
}
```



Much nicer!

Pointcuts can do **much** more than subclassing

Power/Ease (Caesar)

```
interface ObserverProtocol {
  interface Subject {
    provided void addObserver(Observer o);
    provided void removeObserver(Observer o);
    provided void changed();
  }
  interface Observer { expected void notify(Subject s); }
}

class ObserverProtocolImpl implements ObserverProtocol {
  class Subject {
    List observers = new LinkedList();
    void addObserver(Observer o) { observers.add(o); }
    void removeObserver(Observer o) { observers.remove(o); }
    void changed() {
      ... ((Observer)iter.next()).notify(this); ...
    }
  }
}
```

Power/Ease (Caesar)

```
class FigureObserver binds ObserverProtocol {
  class FigureSubject binds Subject wraps Figure { }

  class ScreenObserver binds Observer wraps Screen {
    void notify(Subject s) {
      ...
    }
  }

  after(Figure f): (call(void f.setPoint(Point)) ||
                    call(void f.setLine(Line))) {
    FigureSubject(f).changed();
  }
}
```

Scorecard

	FOA	AspectJ	Caesar
Power			
Ease			

Dynamic Config (FOA)

- No. Once refined, a class is always refined.

```
class Figure { ... }
class Screen { ... }

refines class Figure {
  List observers;
  public void addObserver(Figure o) { ... }
  public void removeObserver(Figure o) { ... }
}

refines class Screen {
  public void notify(Screen s) { ... }
}
```

Dynamic Config (AspectJ)

- No. Once the aspect is included, it always affects the pointcuts.

```
public aspect FigureObserver extends ObserverProtocol {
  declare parents: Figure implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    (call(void Figure.setPoint(Color)) ||
     call(void Figure.setLine(Color))
    ) && target(s);

  protected void notifyObserver(Subject s, Observer o) {
    ...
  }
}
```

Dynamic Config (Caesar)

- Yes! Aspects must be deployed before they take effect

Static deployment

```
deployed class FigureObserver { ... };
...
Figure fig = new Figure();
...
FigureObserver fo = FigureObserver.THIS;
fo.FigureSubject fs = fo.FigureSubject(fig);
fs.addObserver(...);
```

Dynamic deployment

```
Figure fig = ...;
ObserverProtocolImpl opi;
if(...) opi = new ShapeObserver();
if(...) opi = new ColorObserver();

deploy(opi) {
  ...
}
```

Scorecard

	FOA	AspectJ	Caesar
Power			
Ease			
Dynamic Config			

Reusability (FOA)

- Need to explicitly shadow parent functions--can require effort linear with size of program
- Bound by original prototypes of overridden functions

Reusability (AspectJ)

- Aspects (generally) have flat structure
  - All state ends up in one place--hard to customize and maintain
  - Can use inter-type declarations, but this pretty much reduces to FOA
- Can use abstract aspects to produce generally applicable aspects (once bound), but the awkwardness in structure remains

Reusability (Caesar)

- Interface/implementation/binding model makes aspects more portable
- Flexibility in deployment makes it more likely a given aspect will be useful for a particular program without explicit design
  - Cheerfully applies to subclasses
- Increase in structure can cause improvements in maintainability (see OOP)

Scorecard

	FOA	AspectJ	Caesar
Power			
Ease			
Dynamic Config			
Reusability			

Extra Discussion

- Deployment improvements?
- How often is this additional power needed? Is it worth the extra effort in the rest of the cases?
- Still vulnerable to changes in the base due to wildcards and such
- Future directions?