Reuso de Software Aula 10

Padrões de Projeto em DSOA (Java vs. AspectJ)

Eduardo Figueiredo
http://www.dcc.ufmg.br/~figueiredo
reuso.software@gmail.com

09 Abril 2012

Agenda

- Estudo de Hannemann and Kiczales
  - Implementação dos 23 padrões de projeto em Java e AspectJ
  - Avaliação preliminar (qualitativa)
- Estudo de Garcia *et al* (AOSD 2005)
  - Avaliação quantitativa

Slides in English 

DP Implementation


J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. Proceedings of the Conf. on OO Progr., Systems, Lang., and Appl. (OOPSLA), 2002.

Study Goals

- Develop and compare Java and AspectJ implementations of the 23 GoF patterns
- Aim to keep the purpose and intention of each design pattern
  - Changing the solution structure and implementation

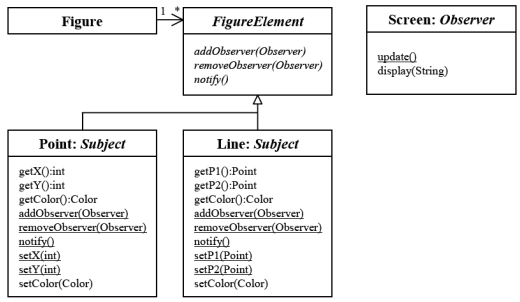
Problems with OO Solution

- Adding or removing a design pattern to/from the system is often invasive
  - Difficult to change
- Design patterns are reusable solutions
  - But their implementations are not
- If multiple patterns are used, it become difficult to document (track) them

Evaluation Criteria

- Locality
- Reusability
- Composability
- Pluggability

Observer (OO Solution)



Java Implementation

```
public interface ChangeSubject {
  public void addObserver(ChangeObserver o);

  public void removeObserver(ChangeObserver o);

  public void notifyObservers();
}
```

```
public interface ChangeObserver {
  public void refresh(ChangeSubject s);
}
```

Screen (Java Impl.)

```
public class Screen implements ChangeSubject, ChangeObserver {
  private HashSet observers = new HashSet();
  private String name;

  public Screen(String s) {
    this.name = s;
  }

  public void display (String s) {
    System.out.println(name + ": " + s);
    notifyObservers();
  }

  public void refresh(ChangeSubject s) {
    String subjectTypeName = s.getClass().getName();
    subjectTypeName = subjectTypeName.substring(
      subjectTypeName.lastIndexOf(".")+1, subjectTypeName.length());
    display("update received from a "+subjectTypeName+" object");
  }
}
```

```
public void addObserver(ChangeObserver o) {
  this.observers.add(o);
}

public void removeObserver(ChangeObserver o) {
  this.observers.remove(o);
}

public void notifyObservers() {
  for (Iterator e = observers.iterator() ; e.hasNext() ;) {
    ((ChangeObserver)e.next()).refresh(this);
  }
}
```

Point (Java Impl.)

```
public class Point implements ChangeSubject {
  private HashSet observers = new HashSet();
  private int x; private int y;
  private Color color;

  public Point(int x, int y, Color color) { ... }

  public int getX() {return x; }
  public int getY() { return y; }
  public Color getColor() { return color; }

  public void setX(int x) {
    this.x = x;
    notifyObservers();
  }

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

  public void setColor(Color color) {
    this.color = color;
    notifyObservers();
  }
}
```

```
public void addObserver(ChangeObserver o) {
  this.observers.add(o);
}

public void removeObserver(ChangeObserver o) {
  this.observers.remove(o);
}

public void notifyObservers() {
  for (Iterator e = observers.iterator() ; e.hasNext() ;) {
    ((ChangeObserver)e.next()).refresh(this);
  }
}
```

Screen (AspectJ Impl.)

```
public class Screen {
  private String name;

  public Screen(String s) {
    this.name = s;
  }

  public void display (String s) {
    System.out.println(name + ": " + s);
  }
}
```

Point (AspectJ Impl.)

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

  public Point(int x, int y, Color color) {
    this.x=x;
    this.y=y;
    this.color=color;
  }

  public int getX() { return x; }

  public int getY() { return y; }

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

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

  public Color getColor() { return color; }

  public void setColor(Color color) { this.color=color; }
}
```

ObserverProtocol (AspectJ)

```
public abstract aspect ObserverProtocol {
  protected interface Subject  { }
  protected interface Observer { }

  private WeakHashMap perSubjectObservers;

  protected List getObservers(Subject subject) {
    if (perSubjectObservers == null) {
      perSubjectObservers = new WeakHashMap();
    }
    List obs= (List)perSubjectObservers.get(subject);
    if ( obs == null ) {
      obs = new LinkedList();
      perSubjectObservers.put(subject, obs);
    }
    return obs;
  }

  public void addObserver(Subject subject, Observer observer) {
    getObservers(subject).add(observer);
  }

  public void removeObserver(Subject subject, Observer observer) {
    getObservers(subject).remove(observer);
  }
}
```

```
protected abstract pointcut subjectChange(Subject s);

after(Subject subject): subjectChange(subject) {
  Iterator iter = getObservers(subject).iterator();
  while ( iter.hasNext() ) {
    updateObserver(subject, ((Observer)iter.next()));
  }
}

protected abstract void updateObserver(
    Subject subject, Observer observer);
```

CoordinateObserver (AspectJ)

```
public aspect CoordinateObserver extends ObserverProtocol{
  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject subject):
    (call(void Point.setX(int)) ||
    call(void Point.setY(int)) ) && target(subject);

  protected void updateObserver(Subject subject, Observer observer) {
    ((Screen)observer).display("Screen updated "+
    "(point subject changed coordinates).");
  }
}
```

ColorObserver (AspectJ)

```
public aspect ColorObserver extends ObserverProtocol{
  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject subject):
    call(void Point.setColor(Color)) && target(subject);

  protected void updateObserver(Subject subject, Observer observer) {
    ((Screen)observer).display("Screen updated "+
    "(point subject changed color).");
  }
}
```

ScreenObserver (AspectJ)

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

  protected pointcut subjectChange(Subject subject):
    call(void Screen.display(String)) && target(subject);

  protected void updateObserver(Subject subject, Observer observer) {
    ((Screen)observer).display("Screen updated " +
    "(screen subject displayed message).");
  }
}
```

Development Findings

- Common parts to all instances
  - Existence of Subject and Observer classes
  - Mapping from subjects to observers
  - Update logic: subject changes trigger observer updates
- Specific parts to each instance
  - Which classes are subjects and observers
  - Which changes trigger the observers
  - Meaning of updating for each observer

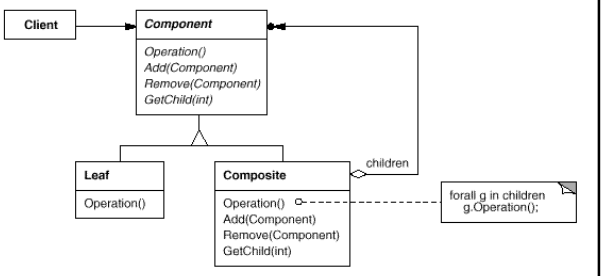
Comparison Findings (I)

- Locality
  - All code is in the abstract and concrete aspects
- Reusability
  - The core pattern code is abstracted and reusable
  - The base classes are cleaner and easier to reuse

Comparison Findings (II)

- Composition Transparency
  - Classes playing multiple pattern roles do not become more complicated
  - Each pattern instance can be reasoned about independently
- Pluggability
  - It is possible to switch between using a pattern or not

Composite



Java Implementation

```
public interface FileSystemComponent {

  public void add(FileSystemComponent component);

  public void remove(FileSystemComponent component);

  public FileSystemComponent getChild(int index);

  public int getChildCount();

  public int getSize();

}
```

Diretory (Java Impl.)

```
public class Directory implements FileSystemComponent {

  protected LinkedList children = new LinkedList();

  protected String name;

  public Directory(String name) {
    this.name = name;
  }

  public String toString() {
    return ("Directory: "+name);
  }

  public void add(FileSystemComponent component) {
    this.children.add(component);
  }

  public void remove(FileSystemComponent component) {
    this.children.remove(component);
  }

  public FileSystemComponent getChild(int index) {
    return (FileSystemComponent) children.get(index);
  }

  public int getChildCount() {
    return children.size();
  }

  public int getSize() {
     return 0;
  }

}
```

File (Java Impl.)

```
public class File implements FileSystemComponent {

  protected String name;

  protected int size;

  public File(String name, int size) {
    this.name = name;
    this.size = size;
  }

  public void add(FileSystemComponent component) {}

  public void remove(FileSystemComponent component) {}

  public FileSystemComponent getChild(int index) {
    return null;
  }

  public int getChildCount() {
    return 0;
  }

  public int getSize() {
    return size;
  }

  public String toString() {
    return ("File: "+name+" ("+size+" KB)");
  }
}
```

Diretory and File (AspectJ)

```
public class Directory {

  protected String name;

  public Directory(String name) {
    this.name = name;
  }

  public String toString() {
    return ("Directory: "+name);
  }
}
```

```
public class File {

  protected String name;
  protected int size;

  public File(String name, int size) {
    this.name = name;
    this.size = size;
  }

  public String toString() {
    return ("File: "+name+" ("+size+" KB)");
  }

  public int getSize() {
    return size;
  }
}
```

CompositeProtocol (AspectJ)

```
public abstract aspect CompositeProtocol {

  public interface Component {}

  protected interface Composite extends Component {}

  protected interface Leaf extends Component {}

  private WeakHashMap perComponentChildren = new WeakHashMap();

  private Vector getChildren(Component s) {
    Vector children = (Vector)perComponentChildren.get(s);
    if ( children == null ) {
      children = new Vector();
      perComponentChildren.put(s, children);
    }
    return children;
  }

}
```

```
public void addChild(Composite composite, Component c) {
  getChildren(composite).add(c);
}
public void removeChild(Composite composite, Component c) {
  getChildren(composite).remove(c);
}
public Enumeration getAllChildren(Component c)  {
  return getChildren(c).elements();
}
```

FileSystemComposition (AspectJ)

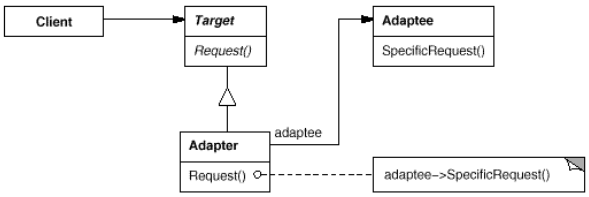
```
public aspect FileSystemComposition extends CompositeProtocol {

  declare parents: Directory implements Composite;

  declare parents: File implements Leaf;

}
```

HK solution uses the Visitor pattern to test, i.e., to navigate over the composite structure.

Roles used within aspect

- Observer and Composite have similar solution
  - As Command, Mediator and Chain of Responsibility
- AspectJ solution
  - Roles are realized with empty interfaces
  - One abstract aspect for each pattern
  - Concrete sub-aspects assign classes to roles

Adapter



Java Implementation

Target

```
public interface Writer {

  public void write(String s);
}
```

Adapter

```
public class PrinterAdapter implements Writer {

  private SystemOutPrinter adaptee = null;

  public PrinterAdapter(SystemOutPrinter adaptee) {
    this.adaptee = adaptee;
  }

  public void write(String s) {
    adaptee.printToSystemOut(s);
  }
}
```

Adaptee

```
public class SystemOutPrinter {

  public void printToSystemOut(String s) {
    System.out.println(s);
  }
}
```

AspectJ Implementation

Target

```
public interface Writer {

  public void write(String s);
}
```

Adapter

```
public aspect PrinterAdapter {

  declare parents: SystemOutPrinter implements Writer;

  public void SystemOutPrinter.write(String s) {
    printToSystemOut(s);
  }
}
```

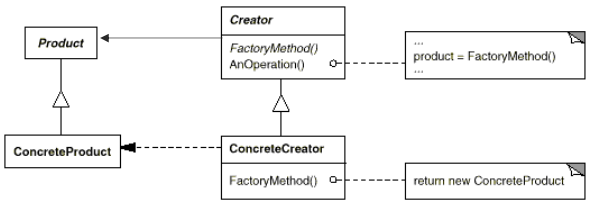
Adaptee

```
public class SystemOutPrinter {

  public void printToSystemOut(String s) {
    System.out.println(s);
  }
}
```

Language Constructs

- Decorator, Strategy, Visitor and Proxy also benefits from AspectJ constructs
- Adapter and Visitor can be realized by extending interface
  - AspectJ open class mechanisms
- Decorator, Strategy and Proxy are implemented based on attaching advice

Factory Method



GUIComponentCreator (Java)

```
public abstract class GUIComponentCreator {
  public abstract JComponent createComponent();
  public abstract String getTitle();
  private static Point lastFrameLocation = new Point(0, 0);
  public final void showFrame() {
    JFrame frame = new JFrame( getTitle() );
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });
    JPanel panel = new JPanel();
    panel.add( createComponent() );
    frame.getContentPane().add(panel);
    frame.pack();
    frame.setLocation(lastFrameLocation);
    lastFrameLocation.translate(75, 75);
    frame.setVisible(true);
  }
}
```

ButtonCreator (Java Impl.)

```
public class ButtonCreator extends GUIComponentCreator {
  public JComponent createComponent() {
    final JButton button = new JButton("Click me!");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button.setText("Thank you!");
      }
    });
    return button;
  }
  public String getTitle() {
    return "Example 1: A JButton";
  }
}
```

LabelCreator (Java Impl.)

```
public class LabelCreator extends GUIComponentCreator {
  public JComponent createComponent() {
    JLabel label = new JLabel("This is a JLabel.");
    return label;
  }
  public String getTitle() {
    return "Example 2: A JLabel";
  }
}
```

GUIComponentCreator (AspectJ)

```
public interface GUIComponentCreator {
  public JComponent createComponent();
  public String getTitle();
}
```

HK solution turns abstract class into interface.

ButtonCreator (AspectJ Impl.)

```
public class ButtonCreator implements GUIComponentCreator {

  public JComponent createComponent() {
    final JButton button = new JButton("Click me!");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button.setText("Thank you!");
      }
    });
    return button;
  }

  public String getTitle() {
    return "Example 1: A JButton";
  }
}
```

LabelCreator (AspectJ Impl.)

```
public class LabelCreator implements GUIComponentCreator {

  public JComponent createComponent() {
    JLabel label = new JLabel("This is a JLabel.");
    return label;
  }

  public String getTitle() {
    return "Example 2: A JLabel";
  }
}
```

LabelCreator (Java Impl.)

```
public aspect CreatorImplementation {

  private static Point lastFrameLocation = new Point(0, 0);

  public final void GUIComponentCreator.showFrame() {
    JFrame frame = new JFrame(getTitle());
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });
    JPanel panel = new JPanel();
    panel.add( createComponent() );
    frame.getContentPane().add(panel);
    frame.pack();
    frame.setLocation(lastFrameLocation);
    lastFrameLocation.translate(75, 75);
    frame.setVisible(true);
  }
}
```

Multiple Inheritance

- Abstract Factory, Template Method, Builder and Bridge have similar solutions
- AspectJ Implementations
  - Replace abstract class by interface
  - Classes are free to extend another application class
  - Do not provide reusable parts

Summary of Results (I)

	Modularity Properties			
Pattern Name	Locality(**)	Reusability	Composition Transparency	(Un)pluggability
Façade	Same implementation for Java and AspectJ			
Abstract Factory	no	no	no	no
Bridge	no	no	no	no
Builder	no	no	no	no
Factory Method	no	no	no	no
Interpreter	no	no	n/a	no
Template Method	(yes)	no	no	(yes)
Adapter	yes	no	yes	yes
State	(yes)	no	n/a	(yes)
Decorator	yes	no	yes	yes
Proxy	(yes)	no	(yes)	(yes)
Visitor	(yes)	yes	yes	(yes)

Summary of Results (II)

	Modularity Properties			
Pattern Name	Locality(**)	Reusability	Composition Transparency	(Un)pluggability
Command	(yes)	yes	yes	yes
Composite	yes	yes	yes	(yes)
Iterator	yes	yes	yes	yes
Flyweight	yes	yes	yes	yes
Memento	yes	yes	yes	yes
Strategy	yes	yes	yes	yes
Mediator	yes	yes	yes	yes
Chain of Responsibility	yes	yes	yes	yes
Prototype	yes	yes	(yes)	yes
Singleton	yes	yes	n/a	yes
Observer	yes	yes	yes	yes

Slides in English

Quantitative Comparison

A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena and A. von Staa. **Modularizing Design Patterns with Aspects: A Quantitative Study**. Proceedings of AOSD, 2005.

Study Replication

- HK assessment goals
  - Locality, Reusability, Composability, and Pluggability
- Our assessment goals
  - Separation of Concerns (SoC), Coupling, Cohesion and Size

Metrics Used

- Separation of Concerns (SoC)
  - Concern Diffusion over Components (CDC)
  - Concern Diffusion over Operations (CDO)
  - Concern Diffusions over LOC (CDLOC)
- Coupling, Cohesion and Size
  - Coupling Between Components (CBC)
  - Depth Inheritance Tree (DIT)
  - Lack of Cohesion in Operations (LCOO)
  - Lines of Code (LOC)
  - Number of Attributes (NOA)
  - Weighted Operations per Component (WOC)

Expanded HK Solution

- In addition to the original HK solution
  - We extend each pattern instance with additional participant classes
- Our goal was investigate the pattern crosscutting structure
  - Create more realistic situations where several classes play each pattern role

Example of Changes

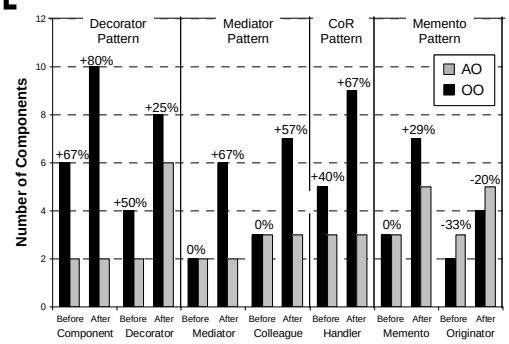
Design Pattern	Introduced Changes
Adapter	4 methods adaptees
Composite	4 Composites and 4 Leafs
Factory Method	4 Creators
Observer	4 Observers and 4 Subjects

Results: Separation of Concerns

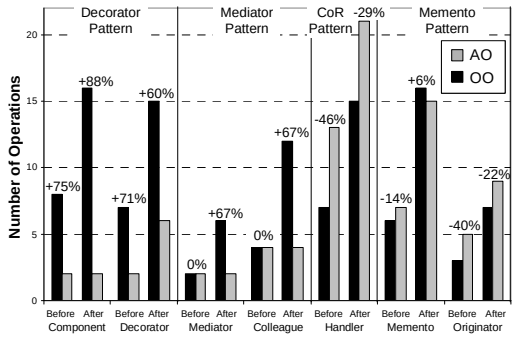
1 – Increased Separation

- 14 design patterns presented better SoC results in the AspectJ solution
  - Decorator, Adapter, Prototype, Visitor, Proxy, Singleton, Mediator, Composite, Observer, Command, Iterator, CoR (Chain of Responsibility), Strategy, and Memento
- Sorted list
  - Decorator presented the best results

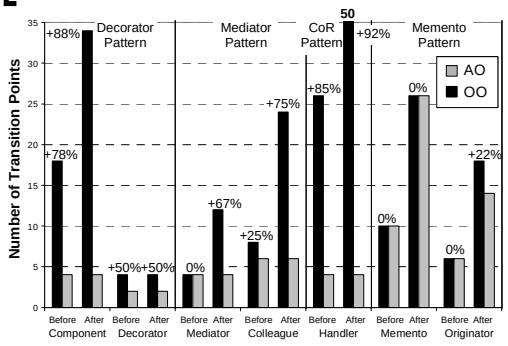
Measurement Results (CDC)



Measurement Results (CDO)



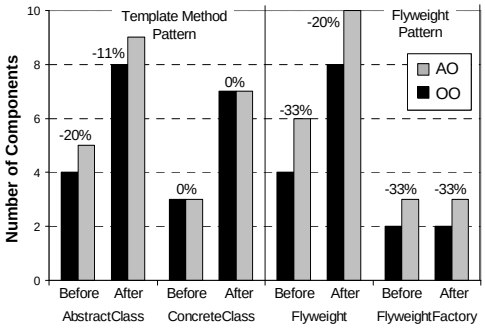
Measurement Results (CDLOC)



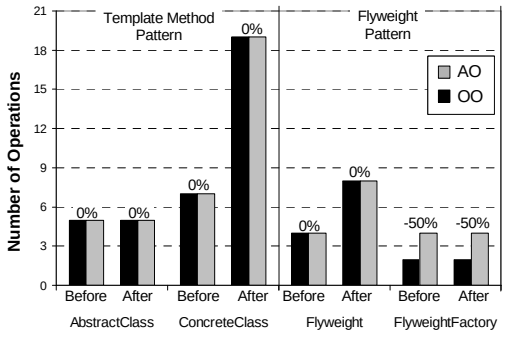
2 – Decreased Separation

- 6 design patterns presented worse SoC results in the AspectJ solution
  - Template Method, Abstract Factory, Factory Method, Bridge, Builder, and Flyweight
- AspectJ solution of Flyweight presented the worst result

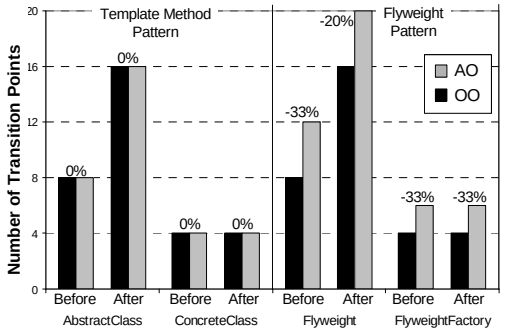
Measurement Results (CDC)



Measurement Results (CDO)



Measurement Results (CDLOC)



3 – No Effect

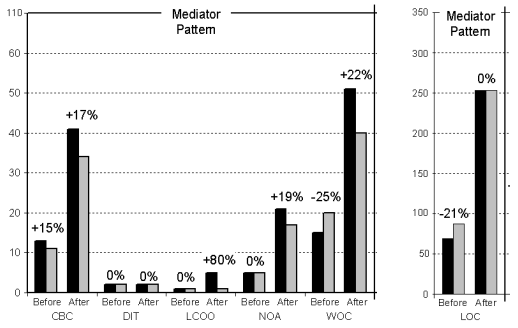
- 3 design patterns presented no significant difference
  - Façade, Interpreter, and State
- Façade used exactly the same implementation in Java AspectJ

Results: Coupling, Cohesion, and Size

1 – Better Results for AO

- 5 design patterns presented better results in the AspectJ solution
  - Composite, Observer, Adapter, Mediator and Visitor
- Sorted list
  - Composite presented the best results

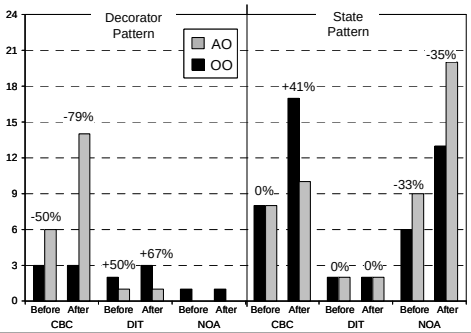
Measurement Results



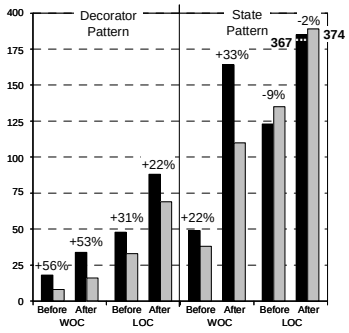
2 – Better Results in Most Metrics

- The AspectJ solutions of 4 design patterns presented better results in most metrics (except one or two)
  - Decorator, Proxy, Singleton and State

Measurement: CBC, DIT and NOA



Measurements: WOC and LOC



Additional Results

- Group 3: Better Results for OO with Exceptions
  - Chain of Responsibility, Command, Prototype and Strategy
- Group 4: Better Results for OO
  - Template Method, Abstract Factory, Bridge, Interpreter, Factory Method, Builder, Memento and Flyweight
- Group 5: No Effect
  - Iterator and Façade

Summary of Results

- Aspects improve separation of concerns in most design patterns
- Aspects help to reduce coupling and increase cohesion of only 9 patterns
- Aspects reduce the number of attributes in 10 design patterns
- Aspects reduce the number of operations+parameters in 12 patterns
- Only 4 patterns implemented in AspectJ exhibited significant reuse

HK Study vs. Our Study

Pattern	HK Study (Locality)	Our First Study (SoC)
Abstract Factory	No	OO
Adapter	Yes	AO
Bridge	No	OO
Builder	No	OO
CoR	Yes	AO
Command	Yes	AO
Composite	Yes	AO
Decorator	Yes	AO
Façade	Same Implementati Java and Aspec	
Factory Method	No	OO

Pattern	HK Study (Locality)	Our First Study (SoC)	
Flyweight	Yes	OO	←
Interpreter	No	=	
Iterator	Yes	AO	
Mediator	Yes	AO	
Memento	Yes	AO	
Observer	Yes	AO	
Prototype	Yes	AO	
Proxy	Yes	AO	
Singleton	Yes	AO	
State	Yes	=	←
Strategy	Yes	AO	
Template Method	Yes	OO	←
Visitor	Yes	AO	