Design Rules for Increasing Modularity with CaesarJ

Carlos Eduardo Pontual

Advisor: Paulo Borba
Aspect-Oriented Software Development

- Better modularize the *crosscutting*-concerns
  - Distribution, persistence, transaction management
- Aims to increase Software Modularity
Does AOP increase Modularity?

- Compromises class comprehensibility
- Changes in a class might break aspects intents
- Parallel development is difficult
So, AOP…

Increases Crosscutting Modularity

But…

Harms Class Modularity.

Is it possible to avoid that?
Improving Modularity OO/AO

- Brief specification of relations/restrictions between classes and aspects
- Design Rules
  - Interfaces: Information Hiding
  - Decouple classes and aspects
  - Guide developers
- Existing solutions (XPIs, LSD)
  - AspectJ
CaesarJ

• Increase support for aspect modularity
  • Aspects implementation and binding
• No difference between classes and aspects
  • Aspect: cclass with pointcuts/advice
  • Aspectual polymorphism
• Aspect Collaboration Interface (ACI)
  • ACIs can be partially implemented
  • Composition using mixins
public cclass Graph {
    public cclass Edge {
        Node start, end;
    }

    public cclass DirectedEdge extends Edge {
        int direction;
    }

    public abstract cclass Node {
        public abstract void getState();
    }
}

The specification of proper DRs in CaesarJ allow a modular (independent) development of OO and AO code.

Is it possible to specify proper DRs with CaesarJ constructs/mechanisms?
Motivating Example

- Tetris games SPL for mobile phones
  - Difficulty feature: Easy and normal variations, among others.
Variation details

TetrisCanvas

<table>
<thead>
<tr>
<th>NextPiece np;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td><strong>void</strong> init() { ...</td>
</tr>
<tr>
<td>np.updatePiece(); ... }</td>
</tr>
<tr>
<td><strong>void</strong> sideBoxes() { ...</td>
</tr>
<tr>
<td>np.updatePiece(); ... }</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

NextPiece

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void</strong> paint() { ...</td>
</tr>
<tr>
<td>paintBox(); ... }</td>
</tr>
<tr>
<td><strong>void</strong> drawPiece() { ... }</td>
</tr>
<tr>
<td><strong>void</strong> updatePiece() { ... }</td>
</tr>
<tr>
<td><strong>void</strong> paintBox() { ...</td>
</tr>
<tr>
<td>drawPiece(); ... }</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Invariant Members

Variant Members
Implementations

- Multiple implementations of the Difficulty Feature
- Different modularization techniques / mechanisms
  - Template Method
  - Extended
  - Mixin Composition
- For each mechanism, two implementations: with or without using ACIs
Enhanced TM

Roles not well defined on the interface (P2)
- Not enough for parallel (independent) development
- No independent compilation of common/variant code (inheritance) (P3)
**Mixin composition – ACI**

<table>
<thead>
<tr>
<th>NextPieceACI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NextPieceBase</strong></td>
</tr>
<tr>
<td>void paint();</td>
</tr>
<tr>
<td>void drawPiece();</td>
</tr>
<tr>
<td>void updatePiece();</td>
</tr>
<tr>
<td>void paintBox();</td>
</tr>
<tr>
<td><strong>NextPieceVariation</strong></td>
</tr>
<tr>
<td>void updatePiece();</td>
</tr>
<tr>
<td>void paintBox();</td>
</tr>
</tbody>
</table>

- Common and Variation code are independent (solve P3)
- Virtual class overhead (P4)
ACI – Mixin Composition

• Common and variation code are independent (solve P3)
• Overhead of virtual classes – P4
• Developers handle instances of Mixin classes – P5
• Mixin allow the specification of roles on different hierarchies
• CaesarJ compiler does not enforce all ACI roles – P6

<table>
<thead>
<tr>
<th>NextPieceACI-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextPieceBase</td>
</tr>
<tr>
<td>void paint();</td>
</tr>
<tr>
<td>void drawPiece();</td>
</tr>
<tr>
<td>void updatePiece();</td>
</tr>
<tr>
<td>void paintBox();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NextPieceVariation</th>
</tr>
</thead>
<tbody>
<tr>
<td>void updatePiece();</td>
</tr>
<tr>
<td>void paintBox();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TetrisCanvas</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init();</td>
</tr>
<tr>
<td>void sideBoxes();</td>
</tr>
</tbody>
</table>
According to the implementations, the CaesarJ interfaces have modularity problems.

Previous work propose the use of Design Rules to increase modularity, but not for CaesarJ.

How can we provide support for increasing CaesarJ modularity?
Our Proposal

- CaesarJ+: Extension that aims to increase CaesarJ modularity

- Set of constructs on top of CaesarJ
  - Specification of Design Rules
  - Syntax-based rules (restrictions)

- CaesarJ+ restrictions are automatically verified
Design Rule construct

- Establish a contract
  - Roles must have an implementation
- Design purpose
- Parameters abstracting roles
  - Increase reuse

```java
import NextPieceDR
import Common, Canvas

abstract class Common {
    ...
}

abstract class Canvas {
    ...
}
```
Complements construct

- Implementation of a class on multiple modules
  - Develop parts of a class in parallel (independently)
- Bi-directional class relationship: mutual visibility
- Only the complemented role can be instantiated
  - One concept implemented on more than one module

```java
dr NextPieceDR [Common, Variation, Canvas] {
    abstract cclass Common {
        abstract void paint(); …
    }

    abstract cclass Variation complements Common {
        abstract void updatePiece();
        …
    }

    abstract cclass Canvas {
        abstract void init()
    }
}
```
Design Rule implementation

- Associates the contract (DR) with classes responsible for roles implementations
- Checks the restrictions

```java
public class NextPiece implements NextPieceDR [Common] {
    void paint() {
    }
}

public class EasyVariation implements NextPieceDR [Variation] {
    void updatePiece() {
    }
}

public class TetrisCanvas implements NextPieceDR [Canvas] {
    void initialization();
}

Must provide init()!
```
Implementation – OO Template Method

abstract class NextPiece {
    void paint() { … paintBox(); … }
    void drawPiece() { … }
    abstract void updatePiece();
    abstract void paintBox();
}

class EasyVariation extends NextPiece {
    void updatePiece() { … }
    void paintBox() { … }
}
class NormalVariation extends NextPiece {
    void updatePiece() { … }
    void paintBox() { … }
}

• Design (abstract signatures) and implementation tangling (P1)
  • The variation part can only be implemented after the implementation of the common code
First idea of interface using ACI (TM)

- States that two roles must be implemented (solves P1)
  - Common (NextPieceBase) and Variation (NextPieceVariation)
- No mutual role dependency
  - Common role methods cannot see variation’s
Signatures duplicated

Roles not well defined
### Implementation – TM ACI

<table>
<thead>
<tr>
<th>NextPieceACI</th>
<th>NextPieceBase</th>
<th>NextPieceVariation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>void</strong> paint();</td>
<td><strong>void</strong> updatePiece();</td>
</tr>
<tr>
<td></td>
<td><strong>void</strong> drawPiece();</td>
<td><strong>void</strong> updatePiece();</td>
</tr>
<tr>
<td></td>
<td><strong>void</strong> updatePiece();</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>void</strong> paintBox();</td>
<td><strong>void</strong> paintBox();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Problems – TM Implementation

<table>
<thead>
<tr>
<th>Problems</th>
<th>No ACI</th>
<th>ACI</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 - Design and implementation tangling</td>
<td>O</td>
<td>-</td>
</tr>
<tr>
<td>P2 - Roles not well defined on the interface</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>P3 - No separated compilation</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>P4 - Overhead of virtual classes</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>
**DR Instantiation**

- Different classes on different contexts can be responsible for a DR
- Instantiation: specifies these classes at a given context
  - A specific DR is generated for each DR instance

```
Instance Name                             Classes implementing DR roles

dr NextPieceEasy = NextPieceDR [NextPiece, EasyVariation, TetrisCanvas]

dr NextPieceNormal = NextPieceDR [NormalVariation, NextPiece, TetrisCanvas]
```

Invalid instance: wrong parameters
Enhancing Expressiveness

- *Exact requisites* x *Minimum requisites*
- Wildcard on method signatures (*, ..)
- Advice restrictions

```
  displayExampleDR [...] {
    ...
    public abstract cclass Point extends FigureElement {
      public void set*(int);
    }

    public abstract cclass LogRole {
      pointcut pointChange(Point p): target(p) && (call(void Point.set*(..));
      after(Point p, ..): pointChange(p) .. ;
    }
    ...
  }
```
Implementation details

- Evaluate the proposed constructs
  - Proof of concept
- CaesarJ+ compiler
  - Transforms CaesarJ+ constructs into CaesarJ valid code
  - Stratego/XT language and toolkit for program transformation
Transforming Design Rules

```java
abstract cclass Common {
    abstract void paint();
    update(): call(* Canvas.*sideBoxes());
}

abstract cclass Variation
    complements Common {
    abstract void updatePiece();
}

abstract cclass Canvas {
    abstract void init();
}
}
```

CaesarJ+ DR

```
abstract cclass NextPieceDR {
    abstract cclass Common {
        abstract void paint();
        update(): call(* Canvas.*sideBoxes());
    }

    abstract cclass Variation extends Base {
        abstract void updatePiece();
    }

    abstract cclass Canvas {
        abstract void sideBoxes();
    }
}
```

Generic ACI
Transforming DR Implementation

- DRs mapped to ACIs ➔ DR roles are virtual classes
- Create collaboration
- Organize inheritance
  - Created collaboration inherits from DR – Class from role
- Update references

```java
class NextPiece implements NextPieceDR [Common] {
    void paint() { ... }
}
```

```java
class NextPiece_Cjp extends NextPieceDR {
}
class NextPiece extends Common {
    void paint() { ... }
}
```
Transforming DR Instantiation

• Generates specific ACI for each instance
• Create a new mixin for each instantiation
  • Inner mixins for complemented roles
• Update references

```java
public cclass NextPieceEasy extends NextPiece_Cjp & EasyVariation_Cjp & TetrisCanvas_Cjp {
    private static NextPieceEasy nextPieceEasy;
    public static NextPieceEasy getInstance() {
        if (nextPieceEasy == null)
            nextPieceEasy = new NextPieceEasy();
        return nextPieceEasy;
    }
}
```

```
dr NextPieceEasy = NextPieceDR [NextPiece, EasyVariation, TetrisCanvas]
```

```java
public cclass NextPice extends EasyVariation {} }
}
Evaluation

• Comparison between CaesarJ+ and CaesarJ
• Tetris SPL difficulty feature
• Observer design pattern
• Health Watcher concern
  • Transaction concern
  • LSD comparison
Evaluation Criteria

• **Language expressiveness:** quantifies the degree in which a language is able to express a restriction
  - It is a three level factor - a language **supports**, does **not support**, or **partially supports** a specific rule

• **Language conciseness:** measures how simple is to express a restriction in a language.
  - Number of tokens required to express a restriction
### Tetris SPL Difficulty Feature

<table>
<thead>
<tr>
<th></th>
<th>Expressiveness</th>
<th>Conciseness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>CaesarJ ACI</td>
<td>SC</td>
<td>PC</td>
</tr>
<tr>
<td>CaesarJ+ DR</td>
<td>SC</td>
<td>SC</td>
</tr>
</tbody>
</table>

SC = Statically checked, PC = Partially checked
N = Not checked
* = Compromised Comprehensibility
Tetris Restrictions

**R5** – An advice which uses at least pointcut `matchCanvas` (R4) must be implemented on the invariant operations.

- Might be an *after advice*

**R6** – Classes and methods used by `matchCanvas` pointcut must exist (have an implementation)
```java
public abstract class Common {
    protected int width, height;
    protected TetrisCanvas tc;

    public abstract void setup(int width, int height, TetrisCanvas tc);
    protected abstract void paint();
    protected abstract void drawPiece(int type);

    pointcut matchCanvas(): execution(* Canvas+.init(..)) || execution(* Canvas+.sideBoxes(..));
    after(..): matchCanvas() ..;
}

public abstract class Variation complements Common {
    protected abstract void paintBox();
    public abstract void updatePiece();
}

public abstract class Canvas {
    protected abstract void init(..); //Used by the pointcuts
    protected abstract void sideBoxes(..); //Used by the pointcuts
}
```
### Observer Design Pattern

**R1** – Two abstractions (roles) on the aspects: Subject and Observer

**R3** – Aspect implementation separated from binding

- Subject implementation – `addObserver`, `removeObserver`, `update`; Binding – `getState`
- Observer Binding - `notify`

<table>
<thead>
<tr>
<th></th>
<th>Expressiveness</th>
<th>Conciseness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>CaesarJ ACI</td>
<td>PC</td>
<td>SC</td>
</tr>
<tr>
<td>Extended CaesarJ ACI</td>
<td>PC</td>
<td>SC</td>
</tr>
<tr>
<td>CaesarJ+ DR</td>
<td>SC</td>
<td>SC</td>
</tr>
</tbody>
</table>
```java
public abstract class Subject {
    public abstract String getState();
}

class SubjectImplementation extends Subject {
    // Implementation details...
}

public abstract class Observer {
    public abstract void notify(Subject s);
}
```
HW Transaction Concern

<table>
<thead>
<tr>
<th></th>
<th>Expressiveness</th>
<th>Conciseness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>LSD DR</td>
<td>SC</td>
<td>SC</td>
</tr>
<tr>
<td>CaesarJ ACI</td>
<td>PC</td>
<td>SC</td>
</tr>
<tr>
<td>CaesarJ+ DR</td>
<td>SC</td>
<td>SC</td>
</tr>
</tbody>
</table>

**R4** – *Advice* must exist call specific transaction methods (behavior)

**R5** – Calls to specific methods must occur only inside the aspect (behavior)
CaesarJ+ allows the specification of restrictions which could not be expressed with ACIs (expressiveness)
  - Static checking
  - Similar number of tokens (concision)
Related work

- AspectJ abstract aspects
  - Aspects restrictions only
  - No advice restrictions
- XPIs
  - Complex specifications
  - Natural language
- Language for Specifying Design Rules – LSD
  - More expressive (behavioral, quantification)
  - AspectJ only
Conclusion

• An analysis of the use of CaesarJ ACIs focusing on the specification of modular interfaces for OO and AO code

• CaesarJ+, an extension that enables the specification of design rules for increase CaesarJ modularity

• A compiler for CaesarJ+, which allows the static checking of the restrictions specified on the DRs
Future work

• Finish compiler implementation
  • Complements *pointcuts* updates
  • Accepts only a single file
  • *wildcards / advice* restrictions
• More examples and evaluation
• Introduce new constructs
  • LSD behavioral rules
  • Quantification mechanisms for rules (*all, none, one*)
  • DR inheritance
Future work (2)

- Improve tool support
  - Eclipse plugin
- Integration with other works
  - DR generator
  - ECaesarJ
Thank you! Questions

Design Rules for Increasing Modularity with CaesarJ

Carlos Eduardo Pontual
ceplc@cin.ufpe.br