Polishing Design Flaw Definitions

Code Quality Knowledge Engineering
over
The Quest for the Ultimate Good
Overview

• Prelude: Why study design flaw definitions and design ideas together? [Not in the paper]
• General setup of the exploratory case study
• The concrete case study
• Results
  – Different reasons for adaptations
  – Design ideas of different generality
  – Basis for code quality knowledge engineering
WHY DESIGN FLAW DEFINITIONS VS DESIGN IDEAS
Feature Envy Considered Bad

- Objects bring data and behavior together.
- A method has *feature envy*, if it operates for the major part on data of another class.
- Breaks encapsulation.
Visitor Pattern Considered Good

• The visitor pattern places functionality that operates on the data of certain objects (elements) in separate classes (visitors).

• Reasons
  – Elements build a complex object structure
  – Functionality belongs rather to the whole structure than to single elements.
  – Functionality is expected to change more frequently.
  – Functionality is used only in specific configurations.
Intended Data-Behavior Separation

• Separation of data into an extra object to share this data
  – *data class*: extrinsic state in flyweight, context in interpreter
  – *feature envy*: flyweight in flyweight, expressions in interpreter

• Separation of data to store it independently (in memento)
  – *data class*: memento
  – *feature envy*: originator

• Separation of the behavior to change it dynamically
  – *feature envy*: concrete state, concrete strategy

• Separation of the behavior to localize logic
  – *feature envy*: concrete mediator
Further Pattern Leading to Smells

• *factory, builder, singleton, facade, mediator*
  – central responsibility
  – strong afferent coupling
    = *shotgun surgery* [LM06]

• *concrete factory, builder*
  and its *director, facade, concrete mediator, concrete visitor, context in state*
  – coordinate/configure objects
  – *intensive/dispersed coupling*
  – might attract high complexity.

• *adapter, composite, abstract decorator, concrete decorator*
  – inserted into type hierarchies
  – *refused parent bequest*
  – *tradition breaker*

• *adapter, abstraction in bridge, abstract decorator, facade, proxy, concrete mediator, context in state*
  – add level of indirection
  – *middle man* [FBB+99]
CASE STUDY: LEVERAGING EXPRESSED INTENTIONS
Intensive / Dispersed Coupling

• Intensive: Many dependencies to a few other classes
• Dispersed: Dependencies to many other classes
• “Additionally, based on our practical experience, we impose a minimal complexity condition on the function, to avoid the case of configuration operations (e.g., initializers, or UI configuring methods) that call many other methods. These configuration operations reveal a less harmful (and hardly avoidable) form of coupling [...].” [LM06, p.121]
• "Method has few nested conditionals" measured by MAXNESTING > 1
Result

• Ignoring flat methods increases efficiency:
  – $8086/13297 = 61\%$ are flat

• Ignoring configuration methods and test methods increases effectiveness!
  – Config/test: $704/778 = 90\%$ or $340/1153 = 29\%$.
  – Flat methods: $102/778 = 13\%$ or $202/1153 = 18\%$

$=>$ Smell detection should make use of expressed intentions.
known structure of good design

unknown good design

known structure of bad design

unknown bad design
GENERAL SETUP OF THE EXPLORATORY CASE STUDY
Research questions

• What is the nature of design ideas that are related to design flaws?

• How are both related?

• What are the consequences for operational definitions of design ideas and design flaws?
Case selection

- Select
  - a rather rigid classical design flaw and
  - a thoughtfully designed software system
  in the hope to generate many false positives.

- Select relevant qualities related to the flaw.
- List the possible refactorings to consider.
For each potential design flaw

1. Is it really bad?
2. Could it be improved?
3. Decision based on 1 and 2:
   – Is the code good enough so that the potential violation is actually a false positive or should it be refactored?
4. For true positives:
   – How to refactor?
5. For false positives:
   – How do we need to adapt the definition of the Law?
   – Can we describe the reason for the adaptation as a design idea?
   – How can we identify the program elements that belong to the design idea? (Extension of the idea)
THE CONCRETE CASE STUDY:
LAW OF DEMETER VS JHOTDRAW
public class Paperboy {
    public void sellPaper(Customer customer) {
        float payment = 2.00f; // “I want my two dollars!”
        Wallet wallet = customer.getWallet();
        if (wallet.getTotalMoney() > payment) {
            wallet.subtractMoney(payment);
        } else {
            // come back later and get my money
        }
    }
}

Bad in terms of encapsulation, coupling, and understandability
public class Paperboy {
    public void sellPaper(Customer customer) {
        float payment = 2.00f; // "I want my two dollars!"
        float paidAmount = customer.getPayment(payment);
        if (paidAmount == payment) {
            // say thank you and give customer a receipt
        } else {
            // come back later and get my money
        }
    }
}
public class Customer {
    public float getPayment(float bill) {
        if (wallet != null) {
            if (wallet.getTotalMoney() > bill) {
                wallet.subtractMoney(bill);
                return bill;
            }
        }
        return 0;
    }
}
public class Paperboy {
    public void sellPaper(Customer customer) {
        float payment = 2.00f; // “I want my two dollars!”
        if (customer.getTotalMoney() > payment) {
            customer.subtractMoney(payment);
        } else {
            // come back later and get my money
        }
    }
}
Law of Demeter

Java, class form’s strict version (based on [LH89])

In all executables (methods, constructors, initializer blocks, field initialization expressions) M of class C, you may use only members (methods and fields) of the following types (classes and interfaces) and their supertypes:

- C,
- types of fields of C,
- types of parameters of M, or
- classes that are instantiated in M.
Qualities, Refactorings

• Coupling:
  – A method M can only be affected by changes to its “friend” types.

• Understandability:
  – “[W]hen reading a method M, the programmer has only to be aware of the [friend] classes of M”

• Encapsulation:
  – Limiting the access to classes returned by method calls that might reveal information about the internal structure of the “friend” objects.

• Potential Refactorings:
  – Pushing back, Lifting forward
JHotDraw 5.1

[Image of a JHotDraw 5.1 screenshot with a diagram created with JHotDraw 5.1]
JHotDraw 5.1 (as documented)
JHotDraw 5.1 (as documented)
Why 5.1?

(a) JHotDraw 5.1
(b) JHotDraw 5.2
(c) JHotDraw 5.3
public class TriangleFigure extends RectangleFigure {

    protected int fRotation = 0;

    public Polygon polygon() {
        Rectangle r = displayBox();
        Polygon p = new Polygon();
        switch (fRotation) {
        case 0:
            p.addPoint(r.x + r.width/2, r.y);
            p.addPoint(r.x + r.width, r.y + r.height);
            p.addPoint(r.x, r.y + r.height);
            break;
        case 1:
            p.addPoint(r.x + r.width, r.y);
            p.addPoint(r.x + r.width, r.y + r.height);
            p.addPoint(r.x, r.y);
            break;
        // [ ... 6 more cases of this kind ... ]
        }
        return p;
    }
}
Review of this potential flaw

- Understandability: Rectangle is elementary common knowledge
- Coupling: Rectangle is shallow and the class is very stable.
- Encapsulation: There is nothing to hide about Rectangles.

- Lifting refactoring would mean to replace every method returning one Rectangle with four methods returning an int.
- Pushing the calculation back up into the class RectangleFigure nonsense since functionality would not be used by other classes.

- Lit.: Like Complex in [LH89]. Empirically no problem [Yam14].

=> Rectangle should be seen as “everybody’s friend”, i.e. every method should be allowed to access their members.
public class ElbowHandle extends AbstractHandle {
    private int fSegment;

    public ElbowHandle(LineConnection owner, int segment) {
        super(owner);
        fSegment = segment;
    }

    private int constrainX(int x) {
        LineConnection line = ownerConnection();
        Figure startFigure = line.start().owner();
        Figure endFigure = line.end().owner();
        Rectangle start = startFigure.displayBox();
        Rectangle end = endFigure.displayBox();
        Insets i1 = startFigure.connectionInsets();
        Insets i2 = endFigure.connectionInsets();

        int r1x, r1width, r2x, r2width;
        r1x = start.x + i1.left;
        r1width = start.width - i1.left - i1.right -1;
        r2x = end.x + i2.left;
        r2width = end.width - i2.left - i2.right -1;

        if (fSegment == 0)
            x = Geom.range(r1x, r1x + r1width, x);
        if (fSegment == line.pointCount() -2)
            x = Geom.range(r2x, r2x + r2width, x);
        return x;
    }
}
DrawingView holds the Selection

```java
public interface DrawingView extends ImageObserver, DrawingChangeListener {
    // [...]
    public Vector selection();
    public FigureEnumeration selectionElements();
    public Vector selectionZOrdered();
    public int selectionCount();
    public void addToSelection(Figure figure);
    public void addToSelectionAll(Vector figures);
    public void removeFromSelection(Figure figure);
    public void toggleSelection(Figure figure);
    public void clearSelection();
    public FigureSelection getFigureSelection();
    // [...]
}
```
public interface DrawingEditor {

  DrawingView view();
  Drawing drawing();
  Tool tool();

  void toolDone();
  void selectionChanged(DrawingView view);
  void showStatus(String string);
}

DrawingEditor
the sad said mediator
public interface DrawingView extends ImageObserver, DrawingChangeListener {

    public void setEditor(DrawingEditor editor);
    public Tool tool();
    public Drawing drawing();
    public void setDrawing(Drawing d);
    public DrawingEditor editor();
    // [...]

}
RELATIONS OF DESIGN IDEAS TO DESIGN FLAW DEFINITIONS
Means to Make Amendments

- Extend the original categories of exceptions:
  - Constructor parameters like fields
  - ... like parameters, like instantiations
- Add a free pass category:
  - Data classes are “everybody’s friend”
- Add a new category of exception:
  - Consider the methods returning the selection to be an “introducing member”
- Adapt the meaning of “type of”
- Recursion

Since we started with a very strict rule, all amendments are relaxations.
Elaboration of the Original

• Completion of the coverage
  – Some facets of Java needed to be covered, e.g. for anonymous inner classes the friends of the enclosing method should be as well considered its friends.

• Improved inference
  – Some type information was not immediately enough available.

• Added interpretation
  – Creational patterns are considered as alternative ways of instantiation
Conflicts With the Original

• Trade for our preference
  – Easily understandable data classes
  – Reasonable choice to localize the information about the selected figures in the `DrawingView`

• Resign because of our ability
  – Code not under our control
  – Code followed very established conventions

• Postpone because of our current ability
  – The undutiful mediator `DrawingEditor` exposing colleagues needs improvement, but is too deeply entrenched in the current design.
DESIGN IDEAS
From generic to specific design ideas

• design ideas from very generic to very specific
  – “anonymous inner class”
  – “DrawingView holding Figure selection”

• adaptations with very high confidence to high enough
  – “creation method access is like instantiation”
  – “data classes are everybody’s friend”
Java Ideas 1/2

• public static members (270)
• Collection types (137)
• java.lang types (128)
• array “field” .length (8)
  => consider as everybody’s friends
• anonymous inner class (23)
  => has friends of the enclosing method
Java Ideas 2/2

• Constructor parameters (86)
• Inferred type in collection field (40)
  => like fields
• Immediate downcasts of parameters (16)
• Designated accessors in the JDK (5)
  => like parameters
• System.out (11)
  => introducing members
Design Ideas

• access to singleton     (24)
• call to creation method (2)
• call to factory method  (1)

  => like instantiation

• data class               (540)

  => consider as everybody’s friends

• [covariant owner type, family polymorphism]

  => like fields
JHotDraw 5.1 Ideas 1/2

- Composites with specific components (5)
- Collection parameters containing Figures (3)
- Clipboard contains Figures (2)
  => taking the specific types into account
- Accessors to the selected Figures in DrawingView (20)
  => introducing member
The alleged mediator `DrawingEditor` exposes the colleagues `Tool`, `Drawing`, `DrawingView` and similarly `DrawingView` exposes `Tool`, `Drawing`, `DrawingEditor` => introducing member
Contributions to Precision

1215 potential design flaws, 67 true positives, 3 undecided
=> Very low precision (as intended): 67/1215 = 5.5%

Quite high precision of 67/(1215-472-149-499-25) = 67/70 = 95.7%
(or if the 3 undecided are true positives: 70/70=100%).
Layers of generality / confidence

Temporary definitions

JHotDraw 5.1 specific definitions

General definition of the Law of Demeter
Elements to locate on different layers

- general qualities
- general ideas
- code
- knowledge configuration

- DATA
- CLASS
- Law of Demeter

- Design Idea
- Design Flaw Definition

- low coupling
- understand-ability

- Rectangle
- Subsumption
- Exception

- silently is
- subsume

- Definition of Idea Extension
- Adaptation of Flaw to Idea
