CSE 70: Refactoring

Ingolf Krueger

Based on slides by Macneil Shonle at UTSA (http://www.cs.utsa.edu/~mshonle/) and Michael Godfrey at U Waterloo: http://plg.uwaterloo.ca/~migod/

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114, USA

California Institute for Telecommunications and Information Technologies
La Jolla, CA 92093-0405, USA
Learning Goals for Today
Learning Goals

• Understand the term “Refactoring” and its applications
• Understand the notion of “bad smells”
• Be able to identify the #1 bad smell in software
• Be able to identify opportunities for and perform mechanics of key refactorings:
  – Encapsulate Field
  – Extract variable/method/class
  – Pull up/Push down method
  – Replace type-checking code with State/Strategy
• Be able to respond to bad smells via refactoring
Something “smells funny”

```java
int[][] a = ...;
for (int i = 0; i < a.length; ++i) {
    for (int j = 0; j < a[i].length; ++j) {
        a[i][j] += i * c;
    }
}
```

• What could it be?
Let’s Try It: **Extract Local Variable** in Eclipse

- Select an expression in Eclipse, then right-click on it, select Refactor->Extract Local Variable

- Selecting “up” the code structure in Eclipse:
  - Alt+Shift+Up

- Selecting “down” in Eclipse:
  - Alt+Shift+Down (typically if you went up too far)

- Move the current line up one line:
  - Alt+Up

- Move the current line down one line:
  - Alt+Down
Let’s try another

```java
if (windowWidth == 0) {
    titleWidth = 0;
}
else {
    titleWidth = windowWidth;
}

• Now what?
```
What do you think about this?

```java
if (message instanceof SendTextMessageTo) {
    INotification notification = new 
        SendTextNotificationFrom(
            ((SendTextMessageTo) message).getText(),
            ((SendTextMessageTo) message).getSender());
    SendNotificationTo(notification,
        getClientReference(
            ((SendTextMessageTo) message).getSender()),
            ((SendTextMessageTo) message).getAddressees());
}
```
Is this an Improvement?

```java
if (message instanceof SendTextMessageTo) {
    SendTextMessageTo stmt = (SendTextMessageTo) message;
    Node sender = stmt.getSender();
    Text content = stmt.getText();
    AddressList addressees = stmt.getAddressees();

    INotification notification = new SendTextNotificationFrom(content, sender);
    SendNotificationTo(notification,
                        GetClientReference(sender, addressees);
}
```
“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.” *

*M. Fowler et al.: Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999*
The idea is that you should improve the code in some significant way. For example:

- Reducing near-duplicate code
- Improved cohesion, lessened coupling
- Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, etc...
Refactoring Techniques

Refactoring:

– “minimally invasive” modifications to system structure
  • Strategy of small steps

– Set up adequate test suite before changing the system

– Carry out tests during and after performing the change
  • Increases confidence in correctness
  • Goal: no change of observable behavior
Example: Move method to superclass

```
MessagePrinter

StandardPrinter
  print()

PrettyPrinter
  print()
```

“Pull Up Method”

```
MessagePrinter
  print()

StandardPrinter

PrettyPrinter
```
Refactoring Techniques

• Refactoring proceeds in extremely small steps

• Each individual change is manageable

• Refactoring fits particularly well with design patterns and architectural patterns:
  1. Evaluate current state
  2. Select target pattern
  3. Identify sequence of refactoring steps leading to target pattern
  4. Perform refactoring until target pattern is reached
Refactoring, The Book

• Reference:

• Fowler, Beck, *et al.* are big names in the OOA&D crowds
  – Smalltalk experience
  – OO design patterns
  – XP (extreme programming)

• See also:
Book is very influential in spreading the ideas
  – Experienced OO programmers will have a pretty good handle on most of the concepts

Mostly it’s a catalog of transformations that you can perform on your code, with motivation, explanation, variations, and examples
  – e.g., Extract interface, Move method, Pull up constructor body

Refactorings often come as duals
  – e.g., Replace inheritance with delegation and
  – Replace delegation with inheritance
Refactoring

• Book has a catalogue of:
  – 22 “bad smells”, *i.e.*, things to look out for, “anti-patterns”
  – 72 “refactorings”, *i.e.*, what to do when you find them
    • *This color indicates the name of a refactoring.*
  – As with the Design Patterns book, there is overlap between the catalogue items, and the ideas start to blur.
  – We will look at some of the bad smells and what to do about them.
Duplicated Code

• **Duplicated Code:** “The #1 bad smell”

• Same expression in two methods in the same class?
  – Make it a private ancillary routine and parameterize it

  *(Extract method)*

• Same code in two related classes?
  – Push commonalities into closest mutual ancestor and parameterize
  – Use *template method* DP for variation in subtasks

  *(Form template method)*
Duplicated Code

- Same code in two unrelated classes?
  - Ought they be related?
  - Introduce abstract parent (Extract class, Pull up method)
  - Does the code really belongs to just one class?
  - Make the other class into a client (Extract method)
  - Can you separate out the commonalities into a subpart?
    - Make the method into a sub-object of both classes.
    - Strategy DP allows for polymorphic variation of methods-as-objects
    - (Replace method with method object)
Bad Smell: Long method

• Often a sign of:
  – Trying to do too many things
  – Poorly thought out abstractions and boundaries
  – Micromanagement anti-pattern

• Best to think carefully about the major tasks and how they inter-relate. Be aggressive!
  – Break up into smaller private methods within the class (Extract method)
  – Delegate subtasks to subobjects that "know best" (i.e., template method DP)
    (Extract class/method, Replace data value with object)
Bad Smell: Long method

• Fowler’s heuristic:
  – *When you see a comment, make a method.*
  – Often, a comment indicates:
    • The next major step
    • Something non-obvious whose details detract from the clarity of the routine as a whole.
    • A workaround might be better placed in a Façade DP
  – In either case, this is a good spot to “break it up”.

© Ingolf Krueger, 2010
Bad Smell: Large Class

- *i.e.*, too many different subparts and methods
- Two step solution:
  1. Gather up the little pieces into aggregate subparts.  
     *(Extract class, Replace data value with object)*
  2. Delegate methods to the new subparts.  
     *(Extract method)*
- Likely, you’ll notice some unnecessary subparts that have been hiding in the forest!
- Resist the urge to micromanage the subparts!
Bad Smell: Large Class

• Counter example:
  – Library classes often have large, fat interfaces (many methods, many parameters, lots of overloading)
  – If the many methods exist *for the purpose of flexibility*, that’s OK in a library class.
Bad Smell: Long Parameter List

- Long parameter lists make methods difficult for clients to understand
- This is often a symptom of
  - Trying to do too much
  - ... too far from home
  - ... with too many disparate subparts
Bad Smell: Long Parameter List

Solution:

- Trying to do too much?
  - Break up into sub-tasks (*Extract method*)

- ... too far from home?
  - Localize passing of parameters; don’t blithely pass down several layers of calls
    (*Preserve whole object, Introduce parameter object*)

- ... with too many disparate subparts?
  - Gather up parameters into aggregate subparts
  - Your method interfaces will be much easier to understand!
    (*Preserve whole object, Introduce parameter object*)
Bad Smell: Divergent Change

- Occurs when one class is commonly changed in different ways for different reasons
- Likely, this class is trying to do too much and contains too many unrelated subparts
- Over time, some classes develop a “God complex”
  - They acquire details/ownership of subparts that rightly belong elsewhere
- This is a sign of *poor cohesion*
  - Unrelated elements in the same container
- Solution:
  - Break it up, reshuffle, reconsider relationships and responsibilities (*Extract class*)
Bad Smell: Feature Envy

- A method seems more interested in another class than the one it’s defined in
  
  *e.g.*, A’s method \( m() \) calls lots of get/set methods of class \( B \)

- Solution:
  
  - Move \( m() \) (or part of it) into \( B \)!
  
  - The values were accessed/set for a reason: there’s probably a higher level idea that can become a method

    *(Move method/field, Extract method)*

  - E.g., “I was getting \( x \)’s and \( y \)’s to calculate distances between points: so instead I created a \texttt{getDistance()} method”
Bad Smell: Data Clumps

• You see a set of variables that seem to “hang out” together
e.g., passed as parameters, changed/accessed at the same time

• Usually, this means that there’s a coherent sub-object just waiting to be recognized and encapsulated

```java
void setTitle(string titleText, int titleX, int titleY, Colour titleColor){...}
```
Example

quitCommand(message, command, commandArray);
connectCommand(message, command, commandArray);
signonCommand(message, command, commandArray);
...

• All subparts of an object are instances of primitive types (int, String, boolean, double, etc.)
  e.g., dates, currency, SIN, tel.#, ISBN, special string values

• Often, these small objects have interesting and non-trivial constraints that can be modelled
  e.g., fixed number of digits/chars, check digits, special values

• Solution:
  – Create some “small classes” that can validate and enforce the constraints.

  • Types make the program easier to understand
  • Avoids “puns”

(Replace data value with object, extract class, introduce parameter object)
Learning Goals for Today
Learning Goals

• Understand the term “Refactoring” and its applications
• Understand the notion of “bad smells”
• Be able to identify the #1 bad smell in software
• Be able to identify opportunities for and perform mechanics of key refactorings:
  – Encapsulate Field
  – Extract variable/method/class
  – Pull up/Push down method
  – Replace type-checking code with State/Strategy
• Be able to respond to bad smells via refactoring