Implementing evolution: Refactoring

Alexander Serebrenik
Last week

• **How to implement** evolution
  • Last week: evolution strategies and decision making
  • Today: refactoring
Problem: changing code is difficult

• Correct code can be far from perfect:
  • Bad structure
  • Code duplication
  • Bad coding practices
  • …

• We need to change it
  • Undisciplined code modification may introduce bugs
  • … and does not guarantee that the code will actually be improved!
  • Manual work, not clear how to support it beyond “copy/paste” and “replace all”
Refactoring

• Refactoring – a disciplined technique for restructuring code, altering its internal structure without changing its external behavior.

• External behavior not changed
  • New bugs are not introduced
  • Old ones are not resolved!

• Aims at improving
  • maintainability, performance
Examples of refactorings

• Extract method
  • If similar series or steps are repeatedly executed, create a separate method

• Rename method
  • If the method’s name no longer corresponds to its purpose/behaviour, rename the method

• Pull up
  • Move the functionality common to all subclasses to the/a superclass

• Push down
  • If the functionality is needed only in some subclasses move it to the subclass
Refactoring catalogue [Fowler]: Example

- **Name**: Inline Temp
- **Applicability**:
  - A temp is assigned to once with a simple expression, and it is getting in the way of other refactorings.
  - Replace all references with the expression
- **Motivation**: simplifies other refactorings, e.g., Extract Method
- **Steps (Java)**:
  - Declare the temp as final, and compile
  - Find references to the temp and replace them
  - Compile and test after each change
  - Remove the declaration and the assignment of the temp
  - Compile and test

Why would you declare the temp as final?
How many refactorings are there?

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Language</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fowler book and website</td>
<td>2000</td>
<td>Java</td>
<td>93</td>
</tr>
<tr>
<td>Thompson et al. website</td>
<td></td>
<td>Haskell</td>
<td>20 * 3 categories</td>
</tr>
<tr>
<td>Garrido</td>
<td>2000</td>
<td>C</td>
<td>29</td>
</tr>
<tr>
<td>Serebrenik, Schrijvers, Demoen</td>
<td>2008</td>
<td>Prolog</td>
<td>21</td>
</tr>
<tr>
<td>Fields et al.</td>
<td>2009</td>
<td>Ruby</td>
<td>&gt;70</td>
</tr>
</tbody>
</table>

- One has to organize refactorings by categories
- We will discuss some of the refactorings in more detail!
Categories of refactorings [Fowler]

- Composing methods (extract method, inline temp)
- Moving features between objects (move field, remove middle man)
- Organizing data (change value to reference)
- Simplifying conditional expressions
- Making method calls simpler (rename method)
- Dealing with generalization (pull up field)

- Big refactorings (separate domain from presentation)
Closer look: Pull Up Field / Push Down Field

• When would you use each one of the refactorings?
  • Pull Up: field is common to all subclasses
  • Push Down: field is used only in some subclasses
Pull Up: Seems simple...

• Inspect all uses of the candidate fields
  • Ensure they are used in the same way.
• If the fields do not have the same name, rename
  • The candidate fields should have the name you want to use for the superclass field.
• Compile and test.
• Create a new field in the superclass.
  • If the fields are private, protect the superclass field
    • The subclasses should be able to refer to it.
• Delete the subclass fields.
• Compile and test.
• Consider using Self Encapsulate Field on the new field.
static Order order;
static char name[ ];
void printOwing ( ) {
    Enumeration e = elements (order);
    double outst = 0.0;
    // print banner
    printf ( " ******************** \n" );
    printf ( " Customer Owes \n" );
    printf ( " ******************** \n" );
    // calculate outstanding
    while ( hasMoreElements ( e ) ) {
        Order each = nextElement (e);
        outst += getAmount ( each );
    }
    // print details
    printf ( "name %s \n" , name );
    printf ( "amount %s \n" , outst );
}

/* Print details for a specific customer */
void printOwing ( ) {
    Enumeration e = elements (order);
    double outst = 0.0;
    printBanner();
    // calculate outstanding
    while ( hasMoreElements ( e ) ) {
        Order each = nextElement (e);
        outst += getAmount ( each );
    }
    // print details
    printf ( "name %s \n" , name );
    printf ( "amount %s \n" , outst );
}
static Order order;
static char name[ ];

// print banner
void printBanner()
{
    printf(" ******************** \\
" );
    printf(" Customer Owes \\
" );
    printf(" ******************** \\
" );
}

void printOwing()
{
    Enumeration e = elements (order) ;
    double outst = 0.0 ;
    printBanner();
    // calculate outstanding
    while ( hasMoreElements ( e ) ) {
        Order each = nextElement (e) ;
        outst += getAmount ( each ) ;
    }
    // print details
    printf("name %s \\
" , name) ;
    printf("amount %s \\
" , outst) ;
}

// print details
void printDetails(double outst) {
    // calculate outstanding
    while ( hasMoreElements ( e ) ) {
        Order each = nextElement (e) ;
        outst += getAmount ( each ) ;
    }
    printf("name %s \\
" , name) ;
    printf("amount %s \\
" , outst) ;
}
static Order order;
static char name[ ];
// print banner

... // print details
void printDetails(double outst) {
    printf( "name %s \n", name );
    printf( "amount %s \n", outst );
}

void printOwing() {
    Enumeration e = elements(order);
    double outst = 0.0;
    printBanner();
    // calculate outstanding
    while ( hasMoreElements(e) ) {
        Order each = nextElement(e);
        outst += getAmount(each);
    }
    printDetails(outst);
}
static Order order;
static char name[ ];

// print banner
...

// print details
...

// calculate outstanding

double getOutst(Enumeration e, double outst) {
    while (hasMoreElements(e)) {
        Order each = nextElement(e);
        outst += getAmount(each);
    }
    return outst;
}

void printOwing() {
    Enumeration e = elements(order);
    double outst = 0.0;
    printBanner();
    outst = getOutst(e, outst);
    printDetails(outst);
}

static Order order;
static char name[ ];

// print banner
...

// print details
...

// calculate outstanding

double getOutst() {
    Enumeration e = elements(order);
    double outst = 0.0;
    while (hasMoreElements(e)) {
        Order each = nextElement(e);
        outst += getAmount(each);
    }
    return outst;
}

void printOwing() {
    printBanner();
    printDetails(getOutst());
}
But is the new program really better than the old one?

- Assume that we want to improve maintainability

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>18</td>
<td>26</td>
</tr>
<tr>
<td>Comments</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Ave McCabe</td>
<td>2</td>
<td>5/4</td>
</tr>
<tr>
<td>Halstead volume</td>
<td>156</td>
<td>226</td>
</tr>
<tr>
<td>Maintainability index</td>
<td>57</td>
<td>77</td>
</tr>
</tbody>
</table>

- Difficult to maintain
- Average maintainability
The refactoring process

- Select the maintainability metrics
  - Recall: Goal – Question – Metrics!

- Refactoring loop
  - Calculate maintainability metrics
  - Identify a problem: “bad smell”
  - Check that the refactoring is applicable
  - Refactor
  - Compile and test
    - Recall: “without changing its external behavior”
  - Recalculate the maintainability metrics
How to identify bad smells?

• **Software metrics**
  - Size: Large class, large method, long parameter list
  - Dependencies: feature envy, inappropriate intimacy
  - % comments: comments

• **Code duplication**

• **Changes (based on version control)**
  - Divergent change (one class is changed in different ways for different reasons)
  - Shotgun surgery (many small changes)
    - Parallel inheritance hierarchies
• Start with complex and tightly coupled classes
Feature envy [Simon, Steinbrückner, Lewerentz]

- Fields – boxes, methods – balls
- Green – Class A, blue – Class B
- Distance

\[ 1 - \frac{|p(X) \cap p(Y)|}{|p(X) \cup p(Y)|} \]

- p(X) – properties of X
  - Method: the method, methods called and fields used
  - Field: the field and methods that use it
How to identify bad smells?

• Structural problems: obsolete parameters, inappropriate interfaces, …

• Parameter $p$ of $C.m$ is obsolete if
  • Neither $C.m$ itself uses $p$
  • Nor any of the classes inheriting from $C$ and reimplementing $m$ uses $p$

• Naïve approach: check all parameters of all methods of all classes
  • Not feasible
  • Better ideas?
Querying the structure [Tourwe, Mens]

Query (a la Datalog):
\[
\text{obsolete(Class,Method,Parameter)}:\quad -
\text{classImplements(Class,Method)},
\text{parameterOf(Class,Method,Parameter)},
\forall (\text{subclassImplements(Class,Method,Subclass)},
\neg (\text{methodUsesParameter(Subclass,Method,Parameter)}))
\]

Advantage:
- Once the DB is populated one can look for different smells
Another example: Inappropriate interface

- **AbstractTerm** cannot be easily extended
  - not clear which subclass should implement terms
- **Query**

  \[
  \text{commonSubclInt(Class,Int,Subcs) :-}
  \]

  \[
  \text{classInterface(Class,ClassInt),}
  \]

  \[
  \text{allSubclasses(Class,SubcList),}
  \]

  \[
  \text{sharedInterface(SubcList,commonInt,Subcs),}
  \]

  \[
  \text{difference(commonInt,ClassInt,Int)}
  \]
How to chose appropriate refactorings?

<table>
<thead>
<tr>
<th>Bad smell</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Introduce assertion</td>
</tr>
<tr>
<td>Duplicated code</td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Extract class</td>
</tr>
<tr>
<td></td>
<td>Pull Up method</td>
</tr>
<tr>
<td></td>
<td>Form Template method</td>
</tr>
<tr>
<td>Feature envy</td>
<td>Move method</td>
</tr>
<tr>
<td></td>
<td>Extract method</td>
</tr>
<tr>
<td>Long method</td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Decompose conditional</td>
</tr>
</tbody>
</table>

- Akin to critique tables discussed last week!
Refactoring never comes alone

• Usually one can find many **different bad smells**
• And for each one many **different refactorings**…
• **Guidelines** when refactorings should be applied

• Still even with strict guidelines [DuBois 2004]
  • `org.apache.tomcat.{core,startup}`
  • 12 classes, 167 methods and 3797 lines of code
  • Potential refactorings
    – Extract Method 5
    – Move Method 9
    – Replace Method with Method Object 1,
    – Replace Data Value with Object 3
    – Extract Class 3
Refactoring never comes alone

- Which one is “better”?
- The most beneficial for the maintainability metrics we want to improve
  - We can do this *a posteriori* but the effort will be lost!
  - So we would like to assess this *a priori*

- Extract method from multiple methods
  - decreases LOC
  - decreases #dependencies on other classes
### Refactorings and metrics

<table>
<thead>
<tr>
<th>Meta-Pattern Name</th>
<th>Description</th>
<th>CDE</th>
<th>DAC</th>
<th>LCOM</th>
<th>NOM</th>
<th>RFC</th>
<th>TCC</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACTION</td>
<td>adds an interface to a class which enables another class to take a more abstract view of the first class by accessing via interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTENSION</td>
<td>constructs an abstract class from an existing class and creates an extends relationship between the two classes</td>
<td>N1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVEMENT</td>
<td>moves parts of an existing class to a component class and sets up a delegation relationship from the existing class to its component</td>
<td></td>
<td></td>
<td></td>
<td>N1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCAPSULATION</td>
<td>weakens the association between two classes by packaging the object creation statements into dedicated methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N1</td>
</tr>
<tr>
<td>BUILDRELATION</td>
<td>operates the relationship between the classes in a more abstract fashion via an interface</td>
<td></td>
<td></td>
<td></td>
<td>N1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRAPPER</td>
<td>wraps an existing receiver class with another class in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps and similarly any results of such requests are passed back by the wrapper</td>
<td></td>
<td></td>
<td></td>
<td>N1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CDE** – class definition entropy  
- **N** – number of strings in the class  
- **f<sub>i</sub>** – frequency of the string **i**  

\[
CDE = - \sum \frac{f_i}{N} \ln \left( \frac{f_i}{N} \right)
\]

**DAC** – number of ADTs in a class  
**WMC** – WMC/McCabe
The refactoring process

- Select the quality metrics
  - maintainability, performance, ...
  - Recall: Goal – Question – Metrics!

- Refactoring loop
  - Calculate the metrics value
  - Identify a problem: “bad smell”
  - Check that the refactoring is applicable
  - Refactor
  - Compile and test
    - Recall: “without changing its external behavior”
  - Recalculate the metrics value
Inconsistency

• Refactoring can introduce inconsistency
  • In tests by breaking the interfaces
  • In models by making them out-of-date

• We need to detect such inconsistencies
  • A priori: using classification of refactorings
    – We know when the things will go wrong
  • A posteriori:
    – Using a logic formalism
      – Inconsistency = unsatisfiability of a logic formula
    – Using change logs
      – eROSE
Interface preservation by refactorings

- Refactoring can violate the interface

- Classify refactorings [Moonen et al.]
  - Composite: series of small refactorings,
  - Compatible: interface is not changed
  - Backwards compatible: interface is extended
  - Make backwards compatible: interface can be modified to keep it backwards compatible
  - Incompatible: interface is broken, tests should be adapted
### Refactoring and tests

<table>
<thead>
<tr>
<th>Compatible</th>
<th>Inline temp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extract class</td>
</tr>
<tr>
<td></td>
<td>Decompose conditional</td>
</tr>
<tr>
<td>Backwards compatible</td>
<td>Extract method</td>
</tr>
<tr>
<td></td>
<td>Push down/Pull up field</td>
</tr>
<tr>
<td>Make backwards compatible</td>
<td>Add/Remove parameter</td>
</tr>
<tr>
<td></td>
<td>Rename/Move method</td>
</tr>
<tr>
<td>Incompatible</td>
<td>Inline method</td>
</tr>
<tr>
<td></td>
<td>Inline class</td>
</tr>
</tbody>
</table>

- To which group belong
  - Replace Exception with Test
  - Self Encapsulate Field (create **getters and setters**)
But tests are also code!

- **Smells [Moonen et al.]**
  - Mystery guest (dependency on an external resource)
  - Resource optimism (availability of resources)
  - Test run war (concurrent use of resources)
  - General fixture (too many things are set-up)
  - Eager test (several methods are tested together)
  - Lazy tests (the same method for the same fixture)
  - Assertions roulette (several assertions in the same test with no distinct explanation)
  - For testers only (production code used only for tests)
  - Sensitive equality (toString instead of equal)
  - Test code duplication
Smells are there, what about refactorings?

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Bad smell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline resource</td>
<td>Mystery guest</td>
</tr>
<tr>
<td>Setup External Resource</td>
<td>Resource optimism</td>
</tr>
<tr>
<td>Make resource unique</td>
<td>Test run war</td>
</tr>
<tr>
<td>Reduce data</td>
<td>General fixture</td>
</tr>
<tr>
<td>Add assertion explanation</td>
<td>Assertions roulette</td>
</tr>
<tr>
<td>Introduce equality method</td>
<td>Sensitive equality</td>
</tr>
</tbody>
</table>
Alternative: A posteriori inconsistency

• Sometimes we do not know what refactorings took place

• Van Der Straeten et al.: inconsistencies in UML models using encoding as logic formulas
  • Similar technique can be used for code/model
  • Syntax adapted:
    inconsistent(ClassDiagram,SeqDiagram,Class,Obj) :-
    class(Class),
    not(inNamespace(Class,ClassDiagram)),
    instanceof(Class,Obj),
    inNamespace(Obj,SeqDiagram)
Putting it all together: refactoring browsers

- IntelliJ IDEA – first popular commercial refactoring browser for Java
- Today: additional languages
- A number of alternatives
  - Eclipse
  - MS Visual Studio
  - …

Eclipse Europa 3.3
Refactoring browsers have a lot of potential but are they used?

• Students [Murphy-Hill and Black]
  • 16 used Eclipse, 2 used refactoring
  • 42 used Eclipse, 6 used refactoring

• Professionals
  • 112 agile programmers, 68% used refactoring
  • Traditional programmers are expected to be less enthusiastic

• Are refactoring browsers fit to what the developers want to do?
How do people refactor [Murphy-Hill and Black]

- **Floss refactorings**: frequent, intertwined with usual development activities
- **Root canal refactorings**: concentrated refactoring effort, infrequent, no usual development activities take place

- Regular flossing prevents root canal treatment
- Programmers prefer to floss [Weißgerber, Diehl]
We need to focus on floss refactorings

1. Choose the desired refactoring quickly,
2. Switch seamlessly between program editing and refactoring,
3. View and navigate the program code while using the tool,
4. Avoid providing explicit configuration information, and
5. Access all the other tools normally available in the development environment while using the refactoring tool.
Eclipse Europa revisited

Lengthy menus: refactoring selection is slow (Req. 1)
Eclipse revisited

Lengthy menus: refactoring selection is slow (Req. 1)
Configuration dialog (Req. 4) interrupts the regular development activities (Req. 2) and does not allow to view the code (Req. 3).
Lengthy menus: refactoring selection is slow (Req. 1) and configuration dialog (Req. 4) interrupts the regular development activities (Req. 2) and does not allow to view the code (Req. 3).

Preview (good idea) but the basic functionality is limited: no hover-on documentation (Req. 5).
Fast access means limited choice

- Which refactorings are actually used?
  - Column – refactoring
  - Row – developer
  - Colour – times used

- Leaders
  - Rename, move
  - Extract method, pull up
  - ModifyParameters
  - ExtractLocalVariable
  - ExtractInterface
  - ExtractConstant

/ SET / W&I
Eclipse Usage Data Collector (2009)

- Formatting changes excluded
- More or less the same leaders:
  - Rename, move
  - Extract method + getters/setters
  - Extract local variable
Proposed solution: Refactoring Cues

- Short menu (Req. 1)
- Switch is easy (Req. 2)
- Code is visible (Req. 3)
- Dialog is non-modal (Req. 5)

- Configuration (Req. 4) is an issue
No explicit configuration: X-Develop

```java
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == ((Long)obj).longValue();
    }
    return false;
}
```

- Up: Original source code
- Down: After the extract method refactoring

```java
public boolean equals(Object obj) {
    if (obj instanceof Long) {
        return value == m(obj);
    }
    return false;
}
```

- Default method name: m
- The name is pre-selected: the rename method refactoring is intended
How does this look now? Eclipse Indigo

```java
@SuppressWarnings("restriction")
public static String toHexString(float f) {
    if (Math.abs(f) < FloatConsts.MIN_NORMAL)
        // float subnormal
    else
        ...
}
```

- Ctrl+1
- Context-sensitive menu
- Results of refactoring are shown in the yellow box
  - Academic research ⇒ main-stream IDE
But there is more

- Direct shortcuts in the yellow box.
The same idea works for
- default constructors
- overridden methods from the superclass
Additional features of Eclipse Indigo

• Ctrl + Shift + O removes unused imports:
  • Why would this feature be interesting?

• Ctrl + Shift + F formats the code according to a predefined style.
  • length of the lines in a source code, placement of brackets, etc.
  • Why would this feature be interesting?
Conclusion

- Refactoring – a disciplined technique for restructuring code, altering its internal structure without changing its external behavior.

- Refactoring loop
  - Calculate maintainability metrics
  - Identify a problem: “bad smell”
  - Check that the refactoring is applicable
  - Refactor
  - Compile and test
  - Recalculate the maintainability metrics

- Refactoring browsers should better support flossing