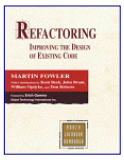
Advanced Object-Oriented Design
Lecture 5

Software Refactoring
Introduction

Bartosz Walter
<Bartek.Walter@man.poznan.pl>

Motto



„Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Martin Fowler

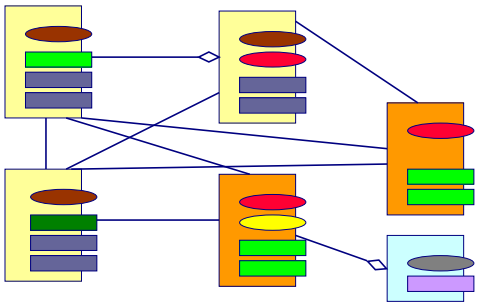
Agenda

- Motivation
- Introduction
- Cost of refactoring
- Correctness
- Bad code smells
- Methods of bad smell detection

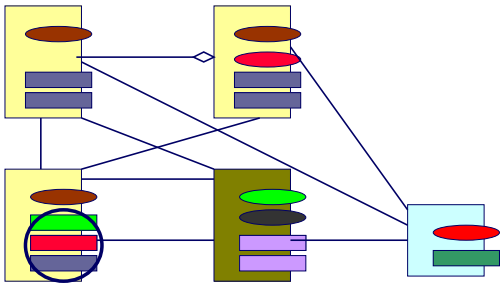
Agenda

Motivation

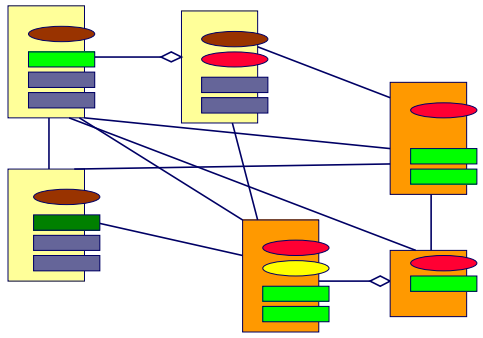
More functionality, quickly!



Bug fixing



We've got no time. Let's integrate...



Motivation for refactoring

- High cost of maintenance
  - 80% of TCO (Yourdon 1976)
  - legacy software problem (Boehm 1975: at Boeing $30 → $4000 per LOC)
- Software entropy increases during development
- Code becomes hardly understandable
- Design does not fit present requirements/functionality
- YAGNI = You Aren't Going to Need It

Agenda

Introduction

Intiuitive definition

Refactoring is:

- a change made to the internal structure of software
- to make it easier to understand and cheaper to modify
- without changing its observable behaviour (same input to a program yields same output)

Source: W. Opdyke, 1991

Formal definition

Refactoring is an ordered pair

$$R = (pre;\ T)$$

where

- *pre* is the precondition that the program must satisfy, and
- *T* is the program transformation

Source: D. Roberts, 1999

Example: Extract Method

Extract Method

```
void scalarProduct(String[] params) {
  int[] x = prepareX(params);
  int[] y = prepareY(params);
  int[] product = computeXY(x, y);
  // ...
  for (i = 0; i < x.length; i++) {
    out.println("X = " + x[i]);
    out.println("Y = " + y[i]);
    out.println("X * Y = " + product[i]);
  }
}
```

```
void scalarProduct(String[] params) {
  int[] x = prepareX(params);
  int[] y = prepareY(params);
  int[] product = computeXY(x, y);
  // ...
  printScalarProduct(x, y, product);
}
```

```
void printScalarProduct(int[] x, int[]y,
int[] product) {
  for (i = 0; i < x.length; i++) {
    out.println("X = " + x[i]);
    out.println("Y = " + y[i]);
    out.println("X * Y = " + product[i]);
  }
}
```

Preconditions:

- there exists no *printScalarProduct()* method in the class
- the class does not inherit neither *scalarProduct()* nor *printScalarProduct()*

Extended formal definition

Refactoring is an ordered triple

$$R = (pre; T; P)$$

where

- *pre* is an assertion that must be true on a program for *R* to be legal,
- *T* is the program transformation, and
- *P* is a function from assertions to assertions that transforms legal assertions whenever *T* transforms programs

Source: D. Roberts, 1999

Example: Rename Method

Rename Method

```
class ClassA
  void methodName1 () {
    // some code
  }
}
```

⇒

```
class ClassA
  void methodName2 () {
    // some code
  }
}
```

Preconditions:

- there exists no method *void methodName2()* in class *ClassA*
- class *ClassA* does not inherit neither *methodName1()* nor *methodName2()*

Postconditions:

- class *ClassA* does not define method *methodName1()*
- class *ClassA* does define method *methodName2()*

Agenda

Cost of refactoring

Cost of refactoring

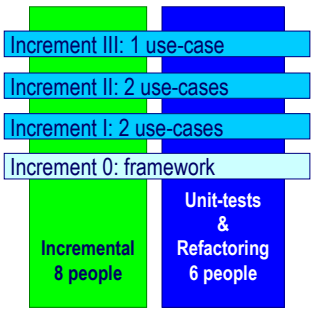
Refactoring <u>does not</u> add new functions to the system, while <u>it does</u> consume time and resources

- automation of localization
- automation of transformation
- automation of verification

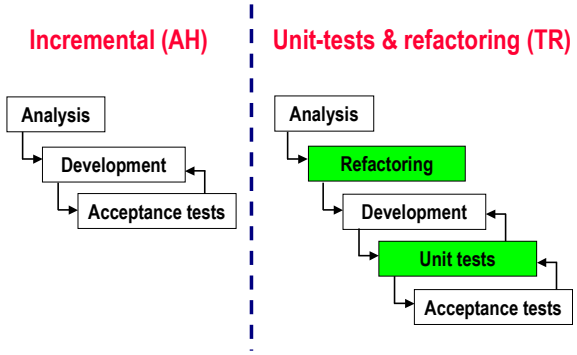
The cost depends on:

- the programming language used
- support from CASE tools
- the nature of refactorings performed
- number and quality of test cases

Experimental evaluation

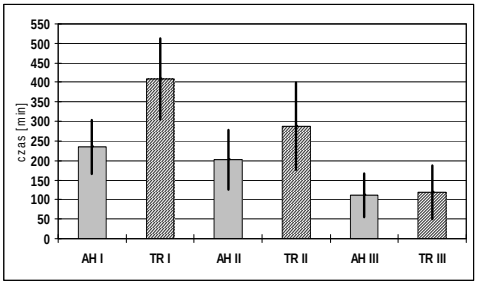


Comparison of development cycles



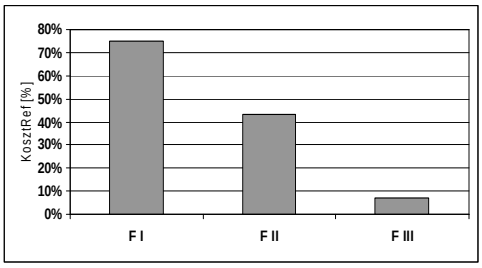
Results

Effort at every increment



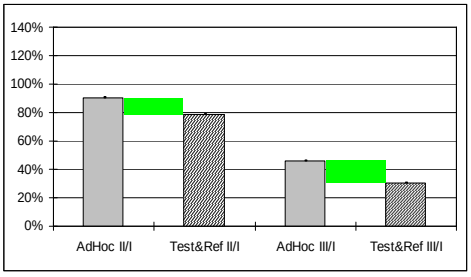
Cost of refactoring

CostRef = (time$_{Test\&Ref}$ / time $_{AdHoc}$) – 1



Relative cost of refactoring

Phase3/Phase1
Phase2/Phase1



When to do and not to do refactoring?

- Three strikes and refactor
- Before functional enhancement
- Variability reveals
- Regular reviews

- Fixed scope
- Run-once projects
- Prematurely published interfaces
- Unstable, rubbish code
- At close deadlines

Unfinished refactoring is like going into debt.
You can live with it, but it is costly.
Ward Cunningham

Agenda

Correctness of refactorings

Example: Inline Temp

```
i = 0;
...
array[0] = i++;   1
array[1] = i++;   2
array[2] = i++;   3
...
```

?

```
i = 0;
x = i++;
array[0] = x;   1
array[1] = x;   1
array[2] = x;   1
...
```

Predicate *noSideEffectsP*

Inputs:
- A program referencing a variable *Var* of initial value 1
- A function *F* potentially modifying the value of *Var*

Problem 1:
- Has a call to the function *F* a side-effect resulting in modifying the variable *Var*?

Lemma 1:
- Problem 1 (*no-side-effects*) is unsolvable

Predicate *noSideEffectsP*

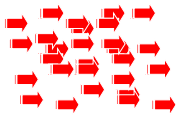
Inputs:
- A program referencing a variable *Var* of initial value 1
- A function *F* potentially modifying the value of *Var*

Problem 2:
- Does exist a inputs' set, which makes the function *F* to modify the variable *Var*?

Lemma 2:
- Problem 2 (modified no-side-effects) is NP-hard

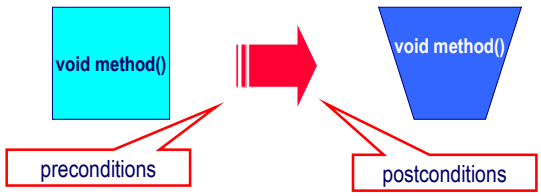
Correctness of refactorings

SIMPLE	HARD
Automated verification	Verification requires testing
Implemented in many IDEs	Tests need to be manually created



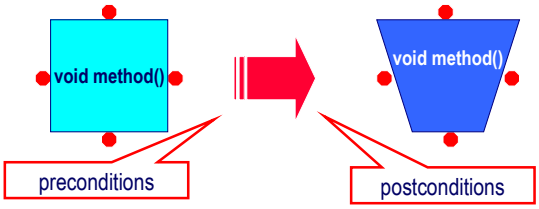
Simple refactorings

- Verified through static analysis
- Can be proved to be correct

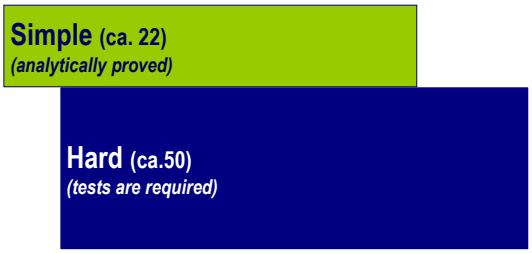


Hard refactorings

- Cannot be proved analitycally
- Role of unit tests



Classification of refactoring depending of verif. method



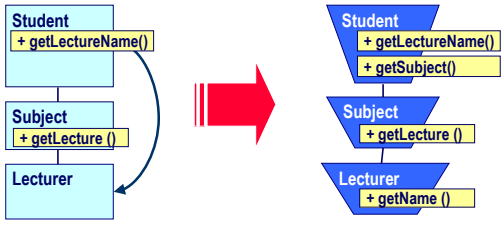
Classification of refactoring depending of verif. method

Simple (ca. 22)
(analytically proved)

Testable (ca. 25)
(tests required, but well-known)

Unknown (ca. 25)
(tests required, but unknown)

Example of a testable refactoring: Move Method



- ✓ check if creating getSubject() in class Student does not disturb inheritance
- ✓ check if creating getName() in class Lecturer does not disturb inheritance
- check if getSubject() in class Student returns appropriate object of class Subject?
- check if Subject object exists every time when Student references it
- check if getLectureName() call returns same name like prior the change

Agenda

Bad code smells

Bad code smells

If it stinks, change it

Kent Beck's grandma
discussing child-rearing philosophy

Bad code smell

A bad smell is a symptom of the poor code quality that indicates the need for refactoring

Martin Fowler
„Refactoring, Improving the design of existing code", Addison-Wesley, 1999

Comments

Symptoms
Comments unnecessarily replicate the control flow

Solutions
- move the over-commented parts to a new method (*Extract Method*)
- *Rename Method* to better reflect its purpose
- *Introduce Assertion* if it clears-out the code control flow

Long Method

Symptoms

- A method performs multiple activities (too many options)
- Not enough support from other methods, causing the method to do tasks at a lower level than it should
- Overly complicated exception handling

- *Extract Method*
- extract temporary variables to external methods with *Replace Temp with Query*
- *Introduce Parameter Object* or *Preserve Whole Object* to decrease the number of parameters
- *Replace Method with Method Object* to extract a method to a new class

Long Parameter List

Symptoms

A method is provided with more external information than it actually needs

Solutions

- *Replace Parameter with Method*, *Preserve Whole Object* or *Introduce Parameter Object* to decrease the number of formal parameters

Duplicated Code

Symptoms

Same or similar code appears all over the system

Solutions

- single class: extract out the common bits into their own method (*Extract Method*)
- sibling classes: *Extract Method* with a shared functionality and then *Pull-up the Method* to a common superclass
- unrelated classes: *Extract Class* with common behavior and delegate to it (possibly in static context)

Large class

Symptoms

- Class holds too much responsibility
- Numerous inner classes, static and instance methods
- Excessive numbers of convenience methods

Solutions

- *Extract Class* to split-up the class by reference
- *Extract Subclass*to split-up the class by inheritance
- *Extract Interface* to split-up the class by polymorphism
- *Extract Superclass* and *Pull-up Method*s to a superclass

Incomplete Library Class

Symptoms

Some functionality is missing from an existing library, while the library cannot be modified

Solutions

- create the methods in the client object (*Introduce Foreign Method*)
- create a subclass or a wrapper with required functionality (*Introduce Local Extension*)

Switch Statements

Symptoms

A method contains a complex, nested switch/conditional statement

Solutions

- extract out the common bits into their own method (*Extract Method*) if code is in same class
- *Replace Conditional with Polymorphism*/*State* to use polymorphism instead
- *Replace Conditional with Subclasses* to use inheritance instead

Speculative Generality

Symptoms

A class is designed to hold some responsibility in the future, but never ends up doing it.

Solutions

- Remove abstract classes with *Collapse Hierarchy*
- Remove unnecessary delegation with *Inline Class*
- Methods with unused parameters - *Remove Parameter*
- Methods named with odd abstract names should be simplified with *Rename Method*

Data Class

Symptoms

A class is merely holding data and offering no interesting methods (Data Transfer Objects)

Solutions

- Move some of clients' code to the data class via a combination of *Extract Method* and *Move Method*
- Split-up and *Inline Class*

Data Clumps

Symptoms

A set of data that's always hanging with each other (e.g. name, street, zip)

Solutions

- Turn the clump into a class with *Extract Class*
- Then continue the refactoring with *Introduce Parameter Object* or *Preserve Whole Object* in order to pass a single instance
- Related to *Long Parameter List*

Refused Bequest

Symptoms

Subclass does not need the inherited data and methods

Solutions

- Create a new sibling class and use *Push Down Method* and *Push Down Field*
- If a subclass is reusing behavior but does not want to support the interface of the superclass, use *Replace Inheritance with Delegation*

Inappropriate Intimacy

Symptoms

Directly getting in the internals of another class

Solutions

- *Move Method / Move Field* to an appropriate class
- restrict the references *Change Bidirectional Association to Unidirectional Association*
- *Extract Class* to hold the shared internals of both classes
- *Replace Inheritance with Delegation* to better separate former super- and subclasses

Lazy Class

Symptoms

A class has no or very limited responsibility

Solutions

- *Collapse Hierarchy* if subclasses are nearly vacuous
- *Inline Class* - move the methods and fields in the class that was using it and remove the original class

Feature Envy

Symptoms
- A method in one class uses lots of functionality from another class
- Low class cohesion

Solutions
- *Move Method* (possibly after *Extract Method* is applied)
- use *Visitor* or *Self Delegation* patterns

Message Chains

Symptoms
Long chain of *getAnotherObject()* calls

Solutions
- *Hide Delegate* to remove unnecessary indirection
- *Extract Method* and then *Move Method* to move it down the chain

Middle Man

Symptoms
A class delegates further most of its methods

Solutions
- *Remove Middle Man*
- If only a few methods aren't doing much, use *Inline Method*
- You could also consider turning the middle man into a subclass with *Replace Delegation with Inheritance*

Divergent Change

Symptoms
A class is commonly changed in different ways for different reasons

Solutions
- Identify everything that changes for a particular cause and use *Extract Class* to put them all together

Shotgun Surgery

Symptoms
A change results in the need to make a lot of little changes in several classes

Solutions
- use *Move Method* and *Move Field* to put all the changes into a single class
- use *Inline Class* to bring a whole bunch of behavior together

Parallel Inheritance Hierarchies

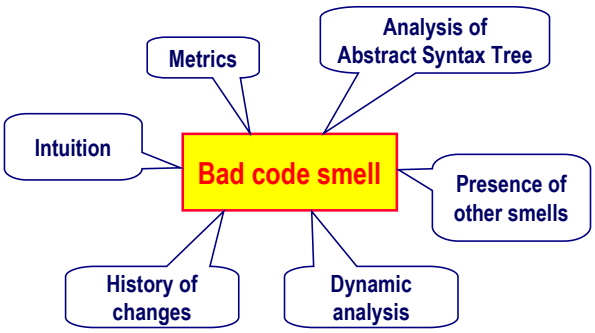
Symptoms
Every time a subclass of one class is created, a corresponding subclass of another is required

Solutions
- use *Move Method* and *Move Field* to combine the hierarchies into one

Agenda

Methods of bad smell detection

Symptoms of a bad smell



Relations among bad code smells

- **Simple compatibility**
  Smell S_1 is compatible with a smell S_2 ($S_1 \Rightarrow S_2$) if the presence of S_1 implies the presence of S_2 (with probability level higher than assumed).
- **Mutual compatibility**
  Mutual compatibility ($S_1 \Leftrightarrow S_2$) is a symmetric closure of a simple compatibilty relation.
- **Transitive compatibility**
  Smell S_1 is transitively compatible with a smell S_3 ($S_1 \Rightarrow \Rightarrow S_3$) if S_1 is compatible with a S_2, and the S_2 is compatible with S_3
- **Aggregate compatibility**
  Smells $S_1,..,S_n$ are compatible as an aggregate with smell S_m ($S_1,..,S_n \Rightarrow S_m$) if their simultaneous presence implies the existence of the S_m with higher probability that for any individual smell $S_1,..,S_n$.
- **Incompatibility**
  Smell S_1 is incompatible with S_2 ($S_1 \not\Rightarrow S_2$) if the presence of S_1 excludes the simultaneous presence of the smell S_2.

Summary

- Refactoring is costly at development phase, but decreases cost of maintenance
- Refactoring must preserve software behaviour
- Testing and analysis as methods of verification
- Code smells indicate a need for refactoring
- Code smell require sophisticated detection and identification mechanism

Readings

1. M. Fowler, *Refactoring. Improving Design od Existing Code.* Addison-Wesley, 1999.
2. Refactoring HomePage, http://www.refactoring.com/
3. W. Wake, *Refactoring Workbook.* Addison-Wesley, 2003.
4. J. Kerievsky, *Refaktoryzacja do wzorców.* Helion, 2005.
5. B. Pietrzak, *XSmells Eclipse Plug-in*

Q&A