Modern Refactoring
Il Escola de Informática Teórica e Métodos Formais

Volker Stolz\textsuperscript{1}, Larissa Braz\textsuperscript{2}, Anna M. Eilertsen\textsuperscript{3}, Fernando Mací\~{a}s\textsuperscript{1}, Rohit Gheyi\textsuperscript{2}

Western Norway University of Applied Sciences, Universidade Federal de Campina Grande, University of Bergen, Norway

Supported by the bilateral SIU/CAPES project “Modern Refactoring” 2017/18
Overview

• What are refactorings? What are they good for?
• Examples in common IDEs
• Implementation of refactorings
• Research challenges & prerequisites
It seems kinda important...

(Everybody’s doing it; you should as well!)
protected void Product.typeCheck(SemanticErrorList e) {
    HashSet<String> featureNames = new HashSet<String>();
    for (Feature f : getModel().getProductLine().getFeatures()) {
        featureNames.add(f.getName());
        Model m = getModel();
        if (m.hasProductLine()) {
            for (Feature f : m.getProductLine().getFeatures()) {
                featureNames.add(f.getName());
            }
        }
    }
    HashSet<String> productNames = new HashSet<String>();
    for (Product prod : getModel().getProducts()) {
        for (Product prod : m.getProducts()) {
            productNames.add(prod.getName());
        }
    }
    HashSet<String> deltaNames = new HashSet<String>();
    for (DeltaDecl delta : getModel().getDeltaDecs()) {
        for (DeltaDecl delta : m.getDeltaDecs()) {
            deltaNames.add(delta.getName());
        }
    }
}
Refactoring: how to do it? Why does everyone hate it?

I refactored once
It was horrible
THAT'D BE GREAT
What is Refactoring? (1)

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [Fowler]

From mathematical term “factor”: finding multiple occurrences of similar code and factoring it into a single reusable function

Motivation:

- keep the code clean
- avoid technical debt
Motivation

Code Quality Measurement: WTFs/Minute

Good Code

Code Review

Bad Code

Code Review

http://commadot.com
What is Refactoring? (2)

- Two different schools:
  - *anything goes* (agile)
  - *behaviour preserving*

- Corner cases:
  - changing complexity class, e.g. replacing bubble sort with quicksort still a refactoring?
Refactoring Process

• Developer inspects code.
• She selects part of it…
• …and chooses refactoring action from menu.
• Refactorings usually modify the Abstract Syntax Tree (AST) in memory…
• … and then synchronize the source code file.
Abstract Syntax Tree (AST)

- In-memory representation of parsed source code
- Semantic information available (Where was this variable declared? What are the superclasses?)

```java
public class C {
    public X x = new X();
    public void f(X x) {
        x.m(this);
    }
}
```
Refactoring: Origins

- Opdyke’s PhD thesis [1992]
- Smalltalk Refactoring Browser [Roberts, Brant, Johnson ’97]
- “Refactoring: improving the design of existing code” [Fowler ’99]
- 30% of changes are refactorings [Soares et al., 2011]
- Extract Method most popular — but performed manually [Murphy et al., 2006]
Literature

Refactoring: Improving the Design of Existing Code
Martin Fowler with Kent Beck, John Brant, William Opdyke, Jon Roberts
Addison Wesley, 1999

Class 1
aMethod()

Class 2

Class 1

Class 2
aMethod()
Adoption of Refactorings

- Agile: fully embraced refactorings
- Developers usually sceptical of automated changes
- Study: developers more confident when they can predict changes
- Problem in OO languages: refactoring touches on multiple contexts
Adoption: Software Engineering Studies

- Kim et al. (2010): survey on more than 300 engineers who had used refactoring during Microsoft Windows development
- Tempero et al. (C.ACM, 2017):
  - Survey on 3785 developers in 2009
  - They understand benefits of refactoring, but they see costs and risks as well.
Related Topics: Patterns

• “Design Patterns: Elements of Reusable Object-Oriented Software” [Gamma, Helm, Johnson, Vlissides, 1994]

• “Refactoring to patterns” [Kerievsky, 2005]

• “Anti-patterns” and “code smells”: indicators of design deficiencies

• Ignoring exceptions (AP), magic strings (AP), repeated code (CS), long functions (CS)

• Detection partially automated

• Refactoring to more structured solutions
Software Quality Metrics

• How “good” is your code?
• Often subjective, but some guidelines:
  • high cohesion/low coupling between classes
  • long method body
  • class with too many methods
• Refactorings affect those metrics:
  • Extract Method reduces length of method and cyclometric complexity…
  • …but obviously increases number of methods.
Software Quality Metrics (2)

- Tools like Findbugs, Checkstyle, SonarQube identify problems
- Developers still need to act on that info
- Problem with automation:
  - large search-space
  - often many (overlapping) possibilities
  - Extract Method ↔ Inline Method “competing” against each other
- Our attempt: Kristensen/Stolz, “Search-based composed refactorings”, NIK 2014

```java
class C {
    A a; B b; boolean bool;
    void method(int val) {
        if (bool) {
            a.foo();
            a = new A();
            a.bar();
        }
        a.foo();
        a.bar();
        switch (val) {
            case 1:
                b.a.foo();
                b.a.bar();
                break;
            default:
                a.foo();
        }
    }
}
```
Reducing Coupling

- Coupling Between Object Classes (CBO) of class C improves from 4 to 3…
- …but sometimes introduces additional coupling into the receiving class!
Related Topics: Source Code Rejuvenation

“Source Code Rejuvenation”
[Pirkelbauer, Dechev, Stroustrup ’10]

- automated migration of legacy code
- leverages enhanced program language/library facilities
- “reverse (some forms of) (software) entropy”
- “preserves or improves a program’s behavior”
# Source Code Rejuvenation

<table>
<thead>
<tr>
<th></th>
<th>Source Code Rejuvenation</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation</td>
<td>Source-to-source</td>
<td>Source-to-source</td>
</tr>
<tr>
<td>Behavior preserving</td>
<td>Behavior <em>improving</em></td>
<td>Behavior preserving</td>
</tr>
<tr>
<td>Directed</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>Raises the level of abstraction</td>
<td></td>
</tr>
<tr>
<td>Drivers</td>
<td>Language / library evolution</td>
<td>Feature extensions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Design changes</td>
</tr>
<tr>
<td>Indicators</td>
<td>Workaround techniques / idioms</td>
<td>Code smells</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Anti-patterns</td>
</tr>
<tr>
<td>Applications</td>
<td>One-time source code migration</td>
<td>Recurring maintenance tasks</td>
</tr>
</tbody>
</table>

From: Pirkelbauer, Dechev, Stroustrup, SOFSEM 2010
Source Code Rejuvenation

vector<int> vec;

// three consecutive push backs
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);

Inefficient!

// copying from an array
int a[] = {1, 2, 3};
vector<int> vec(a, a+sizeof(a)/sizeof(int));

Sizeof() what again?!

Now isn’t that pretty:

// rejuvenated source code in C++0x
vector<int> vec = {1, 2, 3};
Refactoring in IDEs

- All major IDEs support some form of refactoring
- Here: C, C++, Java
- Special case: command line tools for scripting (Go?)
- Support for scripting languages like Python, JavaScript, ...
- Refactoring of UML models (semantical overlap with OO-refactoring)
Tool Support for Java

- Common IDEs:
  - Eclipse JDT
  - IntelliJ (Android)
  - NetBeans
- Other object-oriented languages similar:
  - Visual Studio
Refactoring: Common Java Examples

Encapsulate Field: avoid direct field access

1) introduce setter & getter methods;
2) replace all field accesses with calls to new methods;
3) make field private.
Encapsulate Field

Right-click on a field and find the “Refactor” menu.
Encapsulate Field

IDEs will often have a helpful dialog, because further input is required.
Encapsulate Field

Enjoy your result!
Encapsulate Field

IDEs will even try to be helpful!
Refactoring: Common Java Examples

**Encapsulate Field**: avoid direct field access
1) introduce setter & getter methods;
2) replace all field accesses with calls to new methods;
3) make field private.

Let’s assume you have to *program* this refactoring. Can you see what happens if you swap steps 2 & 3? We will come back later to that.
Refactoring: Extract Local Variable

Before

```java
public void f() {
    a.b.c.d.m();
    a.b.c.d.n();
    a.b.foo(a.b.c.d);
    a.b.bar();
a.b.c.d.m();
}
```

After

```java
public void f() {
    D temp = a.b.c.d;
    temp.m();
    temp.n();
    a.b.foo(temp);
    a.b.bar();
    temp.m();
}
```

Compute complex (expensive) expression only once.
Extract Local Variable: Formally

\[
\text{input : } e \text{ – an expression of non-void type } E \\
\quad : S \text{ – a selection, as a list of consecutive statements} \\
\quad : \text{context} \text{ – the outermost, non-type scope containing } S \\
\text{output: context with } e \text{ extracted to a local variable in } S
\]

1. \( v \leftarrow \) fresh variable name;
2. \textbf{for } s \in S \textbf{ do}
3. \quad \text{in } s \text{ replace all occurrences of } e \text{ with } v;
4. \textbf{end}
5. \text{add a new variable declaration } E v = e \text{ context just before } S;
Other OO Refactorings

Fowler has a pretty loooong list of refactorings in his book:

https://refactoring.com/catalog/
Eclipse Refactoring: General Lifecycle

- Tooling in Eclipse through the Language Toolkit (LTK)
- Refactoring launched by user or script on a context
- Initial check whether refactoring is applicable in this context: checkInitialConditions()
- Configuration details provided by user interactively or through script
- Check if parameters are okay: checkFinalConditions()
- Calculate changes: createChange()
- Preview dialog; change application after confirmation
- Record action in Undo-history
Refactoring for C/C++

- We will see that processing C/C++ has its very own special challenges...
- ...refactoring even more.
- Opdyke: first thesis on “refactoring” in 1992
  - proposed a number of refactorings for C++
  - with conditions to guarantee behaviour preservation
- “Refactoring” first showed up in 1984
A. change_class_name

Change the name of a class.

Arguments: class C, string S.

This name change is reflected throughout the program (i.e. in class and subclass declarations, constructor and destructor functions, and the declarations of instances of the class).

Preconditions:

1. \( \forall \text{ class } \in \text{ Program.classes}, \)
   \[ \text{class.name} \neq S. \]
   (the new name doesn’t clash with an already existing class)

The precondition ensures distinct class names (satisfying program property two). Changing the name of a class does not change its behavior (satisfying program property seven). Other program properties are trivially preserved.

Challenges of Refactoring C Systems

• Typical C file contains not only terminals defined in the C grammar but also preprocessor directives.

• **Conditional compilation:** include code selectively depending on the value of conditions evaluated during compilation.

• Example:

```c
#ifdef A
int x = 0;
#else
int x = 1;
#endif
```

*Code before preprocessing*
A Preprocessing Odyssey…

- The C preprocessor takes as input a C file with directives and outputs pure C code, where directives have been stripped out and substituted accordingly:

```c
#ifdef A
  int x = 0;
#else
  int x = 1;
#endif
```

- Refactoring needs to deal with preprocessor directives!
Challenges of Refactoring C Systems

- Tools that work on preprocessed C code have many disadvantages:
  - allowed refactorings very restricted;
  - directives can be irrecoverable lost or may no longer apply.
- If directives are irrecoverable, the code may become unmaintainable.


Code That Keeps You Awake At Night: Disciplined vs undisciplined conditional annotations

```
#ifdef A
int x = 0;
#else
int x = 1;
#endif
```

Disciplined

```
int x =
#ifdef A
  0;
#else
  1;
#endif
```

Undisciplined

(Don’t write that code!)
Undisciplined Conditional Annotations Refactoring Example

Refactoring improves code quality

However, there is no automatic tool to apply them

F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca:
Research Challenges

• Refactoring at Scale
• Correctness of Refactorings
• Repository mining
• Refactoring *models*
• Rearchitecting (forward engineering)
• (Rearchitecting to concurrency)
• (Machine Learning for refactoring)
Refactoring at Scale

- Refactoring of large code bases
- Example: migration to new APIs
- Semi-Automated refactoring:
  - candidates identified automatically, e.g. during Continuous Integration
  - automated application of suggestions to developers (e.g. through ticketing system)
ClangMR at Google

- Moving from old APIs to new ones…
- …or to C++11 standard constructs.
- Source code rejuvenation!
- “Large-Scale Automated Refactoring Using ClangMR”, Wright et al., ICSM 2013.
ClangMR: Example & Statistics

old internal API

```c++
void SplitStringUsing(const string& full,
    const char* delimiters,
    vector<string>* result);

void foo() {
    string input;  
    vector<string> output;
    ... 
    SplitStringUsing(input, "-", &output);
}
```

new `strings::Split()` API

```c++
namespace strings {
    template <typename Delimiter, Predicate>
    Split(StringPiece text, Delimiter d, Predicate p);
    struct SkipEmpty {
        bool operator()(StringPiece sp) const {
            return !sp.empty();
        }
    };

    void foo() {
        string input;
        ... 
        vector<string> output =
            strings::Split(input, "-", strings::SkipEmpty());
    }
```

- 45,000 callers
- 35,000 transformed
- 3,100 chunks
- largest part reviewed within 2 minutes
ClangMR Video

https://www.youtube.com/watch?v=mVbDzTM21BQ
Correctness of Refactorings

Is it really a refactoring?

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [Fowler]

Challenge:
Can we predict the effect of applying a refactoring?
What could possibly go wrong here?

```java
protected void Product.typeCheck(SemanticErrorList e) {
    Hashset<String> featureNames = new Hashset<String>();
    for (Feature f : getModel().getProductLine().getFeatures()) {
        featureNames.add(f.getName());
    }
}
```

```
Model m = getModel();
```

```java
for (Product prod : m.getProductNames().getProducts()) {
    for (Product prod : m.getProductNames().getProducts()) {
        productName.add(prod.getName());
    }
} Hashset<String> deltaNames = new Hashset<String>();
for (DeltaDecl delta : getModel().getDeltaDecs()) {
    deltaNames.add(delta.getName());
```
A Problem With Extract Local Variable...

```java
public class C {
    public X x = new X();
    {//initializer
        x.myC = this;
    }

    public void f(){
        x.n();
        x.m();
        x.n();
    }
}
```

```java
public class X{
    public C myC;

    public void m(){
        myC.x = new X();
    }

    public void n(){
        System.out.println(  
            this.hashCode());
    }
}
```

Output:
1735600054 ← Method n() called on two different objects
21685669
Refactored...

Output now:
1735600054 ← Method \text{n()} called on \text{the same} object!
1735600054 (changed behaviour)
Does Refactoring Need A Safety Net?

```java
public class C {
    public X x = new X();
    {//initializer
        x.myC = this;
    }
    public void f() {
        X temp = x;
        assert temp == x;
        temp.n();
        assert temp == x;
        temp.m();
        assert temp == x;
        temp.n();
    }
}

public class X {
    public C myC;
    public void m() {
        myC.x = new X();
    }
    public void n() {
        System.out.println(
            this.hashCode());
    }
}
```

Output now:
1735600054
Exception in thread ”main” java.lang.AssertionError
An Isolated Incident?
Meet Extract-And-Move Method

- Combination of two refactorings
- Extract has same effect of evaluating target only once
- But: different implementations actually get this right by using additional parameter instead of `this`!
Test Case Generation for Refactored Code

- General issue: need test cases for (modified) code
- Here: refactoring often introduces new units that need to be subjected to additional tests
- No direct call to refactored method in tests → less likely to discover fault
- High branch coverage → better chances of discovery

[Alves et al., 2017: Test coverage of impacted code elements for detecting refactoring faults: An exploratory study]
Correctness of Refactorings
Is it implemented correctly?

- Implementation of refactorings often difficult
- Many syntactical elements in AST:
  - missing corner cases
  - mistakes
- Eclipse JDT bugtracker issues on refactorings
- Nice separation between UI and parameter objects
- What was the specification of your refactoring anyway?
Correct implementation

- Schäfer et al.: JRRT (based on JastAdd)
- Natural language description vs. implementation
- (Issue: refactoring with same name slightly different implemented)

```java
class A {
    void m() {
        int f = 23;
        ...
    }
}
```

```
class A {
    int f;
    void m() {
        f = 23;
        ...
    }
}
```

```
class A {
    int f(int y) {
        if(y <= 1)
            return 1;
        int x = y;
        return f(y-1) * x;
    }
}
```

- Example: Promote-variable-to-field
- Not always correct/possible
Better Implementations

**Algorithm 1** PROMOTE TEMP TO FIELD($d : LocalVar$)

<table>
<thead>
<tr>
<th>Input Language: Java</th>
<th>Output Language: Java ∪ locked dependencies</th>
</tr>
</thead>
</table>

1. **[SPLIT DECLARATION]**($d$)
2. $d' \leftarrow$ new **private** field of same type and name as $d$
3. make $d'$ **static** if $d$ is in static context
4. **[INSERT FIELD]**($\text{hostType}(d), d'$)
5. **for all** uses $u$ of $d$ **do**
6. lock $u$ onto $d'$
7. lock reaching definitions of $u$
8. **REMOVE DECL**($d$)

Implemented through JastAdd meta-compilation system

```java
public void VariableDeclaration
    .promoteToField()
{
    split();
    Modifiers mods = new Modifiers("private");
    if (inStaticContext())
        mods.addModifier("static");
    TypeAccess ta = type().createLockedAccess();
    FieldDeclaration f
        = new FieldDeclaration(mods, ta, name());
    hostType().insertField(f);
    for (VarAccess va : uses()) {
        va.lock(f);
        va.lockReachingDefs();
    }
    flushCaches();
    remove();
}
```

```java
public void VariableDeclaration
    .doPromoteToField()
{
    Program root = programRoot();
    promoteToField();
    root.eliminate(LOCKED_NAMES, LOCKED_FLOW);
}
```
Repository Mining

- Analyse (open source) repositories
- Example: GitHub (good API?)
- Analyse individual changes:
  - classify type of change — development, or refactoring?
  - “understand” commit message?
  - behaviour preserving?
Repository Mining

G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, T. Massoni: Analyzing Refactorings on Software Repositories

• compare two versions of the software via test suite
• transformation considered a refactoring if no behavioural difference detected between source and target programs
• requires execution
• classification?
Repository Mining

Average of:
refactoring frequency, granularity, scope, and test coverage

<table>
<thead>
<tr>
<th>Subject</th>
<th>Refactoring (%)</th>
<th>Low Level (%)</th>
<th>Local (%)</th>
<th>Average Test Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>20.32</td>
<td>55</td>
<td>61</td>
<td>35</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>33</td>
<td>70</td>
<td>75</td>
<td>30</td>
</tr>
<tr>
<td>jEdit</td>
<td>23.48</td>
<td>45</td>
<td>60</td>
<td>27</td>
</tr>
<tr>
<td>SweetHome 3D</td>
<td>29.30</td>
<td>52</td>
<td>62</td>
<td>35</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>25.70</td>
<td>71</td>
<td>80</td>
<td>31</td>
</tr>
<tr>
<td>Average</td>
<td>27.27</td>
<td>63.83</td>
<td>71.05</td>
<td>30.95</td>
</tr>
</tbody>
</table>

Soares et al.: Analyzing Refactorings on Software Repositories
Refactoring Models

- Refactoring = Model Transformation
- Refactoring models — we’re working e.g. on UML diagrams.
- Refactoring through models — we don’t work on the source, but an intermediate representation. Example: class diagram derived from Java code.
- Refactoring metamodels and instances.
Refactoring in MDSE

• Based on Graph Theory
• Higher level of abstraction, hence simpler reasoning
• Refactorings as model-to-model transformations
• Applicable to general-purpose modelling languages (e.g. UML) and domain-specific modelling languages
• Applicable to low-level or big-scale refactoring (reverse engineering and modernisation)
• Easier specification and visualisation of results
Reverse Engineering of Legacy Systems

- MoDisco Text-to-Model transformations
- Three-phase reengineering process (horseshoe)

Source: Erik Philippus
Model-based Refactoring of Source Code

- Based on the analysis of annotated UML models
- Coordination of refactoring operations based on dependencies
  - Variable accesses
  - Method calls

Refactoring of General-Purpose Languages

- Support in EMF through EMF Refactor
- Refactorings based on class diagrams (Ecore metamodels)
- Extensible framework for any EMF-based model

Source: eclipse.org
Refactoring of DSMLs

strategy freeStarvation()
{
    declare dstList:modelList;
    declare dst1,dst2,p,q:model;
    declare static num:Integer;
    dstList:=findOutConnections();
    assert(dstList.size()==2);
    dst1:=dstList.get(0); dst2:= dstList.get(1);
    p:=createModel("InitMarker","P"+intToStr(num));
    q:=createModel("Place", "Q" + intToStr(num));
    num:=num + 1;
    addConnection(dst1,p);  addConnection(p,dst2);
    addConnection(dst2,q);  addConnection(q,dst1);
}

J. Zhang, Y. Lin, J. Gray: “Generic and domain-specific model refactoring using a model transformation engine”, 2005
Refactoring of Metamodels and Their Instances

- AKA co-evolution AKA model migration

Mantz et al. “Co-evolving meta-models and their instance models: A formal approach based on graph transformation”, 2015
Refactoring an UML Sequence Diagram

[Li et al., Interactive Transformations from Object-Oriented Models to Component-Based Models]
Refactorings = Model Transformation?

- Many existing approaches for graph (tree!) -based models
- AST = model
- Can use OCL and other MDD-tools to describe transformations

Limitation:
- many syntactical elements → complex transformations
- formal reasoning difficult
  - *refactoring always correct* vs. *instance of refactoring correct* (with proof obligations)
Interested?

• Refactorings are an important software engineering topic.

• Tool support for refactoring always needs improvement.

• Requirements:
  
  ● *compiler construction* (to work with programs as input: syntax & semantics, grammars & parsing, ASTs, typing rules)

  ● *logic/discrete maths* (∀,∃,∊,… to read & write specifications)

  ● optional: *static analysis* (information flow analyses etc., also useful for security analysis)

  ● most of all: interest in *coding*!
Foundations:

- E. Gamma, J. Helm, R. Johnson, R. Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994
- J. Kerievsky: “Refactoring to patterns”, Addison-Wesley, 2005
“Modern” Reading:


Bibliography (3)


- D. Li, X. Li, Z. Liu, V. Stolz: Interactive Transformations from Object-Oriented Models to Component-Based Models. FACS 2011, LNCS, Springer 2011.
