Threats to the Validity of Refactoring Studies

Rainer Schaden
Freie Universität Berlin

June 30, 2016
Outline

Introduction

How to find Refactorings?

Repository Mining Tools

Planned Approach
Outline

Introduction

How to find Refactorings?

Repository Mining Tools

Planned Approach
What is Refactoring?

- Opdyke - PhD thesis (1992)
What is Refactoring?

- Opdyke - PhD thesis (1992)

Definition

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” — Martin Fowler
What is Refactoring?

- Opdyke - PhD thesis (1992)

**Definition**

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

— Martin Fowler
Why Refactoring?

- Prevents software aging
Why Refactoring?

- Prevents software aging
- Improves:
  - Design
  - Understandability
  - Maintainability
  - Developer’s productivity
Refactoring Tactics

- Floss Refactoring:
  - Frequent
  - Mixed with other program changes
Refactoring Tactics

- **Floss Refactoring:**
  - Frequent
  - Mixed with other program changes

- **Root-Canal Refactoring:**
  - Infrequent
  - Lengthy
  - Hardly any other program changes
Goal of this Thesis

- Examine Refactoring studies with regards to:
  - Credibility
  - Relevance
  - Validity
Goal of this Thesis

- Examine Refactoring studies with regards to:
  - Credibility
Goal of this Thesis

- Examine Refactoring studies with regards to:
  - Credibility
  - Relevance
Goal of this Thesis

- Examine Refactoring studies with regards to:
  - Credibility
  - Relevance
  - Validity
Outline

Introduction

How to find Refactorings?

Repository Mining Tools

Planned Approach
How to find Refactorings?

- Goal: empirical refactoring data
How to find Refactorings?

- Goal: empirical refactoring data
- Four Methods: (Murphy-Hill)
How to find Refactorings?

- Goal: empirical refactoring data
- Four Methods: (Murphy-Hill)
  - Mining the Commit Log
  - Analyzing Code Histories
  - Observing Programmers
  - Logging Refactoring Tool Use
Mining the Commit Log

- Search for "refactor", "rename", etc. in commit messages

Problems:
- Bad commit messages
- Floss refactoring
- Trend towards certain, large refactorings

Assumptions:
- Developer recalls refactoring
- And describes it accurately
Mining the Commit Log

- Search for "refactor", "rename", etc. in commit messages
- Problems:
  - Bad commit messages
  - Floss refactoring
  - Tendency towards certain, large refactorings
Mining the Commit Log

- Search for "refactor", "rename", etc. in commit messages
- Problems:
  - Bad commit messages
  - Floss refactoring
  - Tendency towards certain, large refactorings
- Assumptions:
  - Developer recalls refactoring
  - And describes it accurately
Analyzing Code Histories

- Analyzing versions of source code
Analyzing Code Histories

- Analyzing versions of source code
- Problems:
  - Manual inspection is slow and error-prone
  - Comparing non-consecutive versions
  - Tools only recognize limited number of refactorings
  - Tools use heuristics
Analyzing Code Histories

- Analyzing versions of source code
- Problems:
  - Manual inspection is slow and error-prone
  - Comparing non-consecutive versions
  - Tools only recognize limited number of refactorings
  - Tools use heuristics
- Assumptions:
  - Adequate granularity in code history
  - Appropriate window of observation
  - Heuristics are parameterized correctly
Observing Programmers

- Direct observation: controlled experiment

- Problems:
  - Controlled experiments are expensive
  - Bad external validity of controlled experiments

- Indirect observation relies on memory of developers

- Assumptions:
  - Developers recall refactorings accurately
  - Frequent refactorings

FU Berlin, Threats to the Validity of Refactoring Studies, June 30, 2016
Observing Programmers

- Direct observation: controlled experiment
- Indirect observation: survey
Observing Programmers

- Direct observation: controlled experiment
- Indirect observation: survey
- Problems:
  - Controlled experiments are expensive
  - Bad external validity of controlled experiments
  - Indirect observation relies on memory of developers
Observing Programmers

- Direct observation: controlled experiment
- Indirect observation: survey
- Problems:
  - Controlled experiments are expensive
  - Bad external validity of controlled experiments
  - Indirect observation relies on memory of developers
- Assumptions:
  - Developers recall refactorings accurately
  - Frequent refactorings
  - External validity
Logging Refactoring Tool Use

- Use log files of automated refactoring tools
Logging Refactoring Tool Use

- Use log files of automated refactoring tools
- Problems:
  - Requires extensive log files
  - Can’t track manual refactorings
Logging Refactoring Tool Use

- Use log files of automated refactoring tools
- Problems:
  - Requires extensive log files
  - Can’t track manual refactorings
- Assumptions:
  - Developers use tools for refactoring
Strength and Weaknesses

- Implicit/Explicit
- Accuracy and Precision
- Context
- Fidelity
Outline

Introduction

How to find Refactorings?

Repository Mining Tools

Planned Approach
Ref-Finder

- Prete et al. (2010)
- Analyzing versions of source code
- Supports 63 refactorings
- Logic rules
Ref-Finder Description

- Input: Two versions of a program
Ref-Finder Description

- Input: Two versions of a program
- Basic facts
  - i.e. method(methodFullName, methodShortName, typeFullName)
Input: Two versions of a program

Basic facts
  - I.e. method(methodFullName, methodShortName, typeFullName)

Deleted_ und added_ facts
Ref-Finder Description

- Input: Two versions of a program
- Basic facts
  - I.e. method(methodFullName, methodShortName, typeFullName)
- Deleted_ und added_ facts
- Similarbody facts
Ref-Finder Description

- Input: Two versions of a program
- Basic facts
  - I.e. method(methodFullName, methodShortName, typeFullName)
- Deleted und added facts
- Similarbody facts
- Refactorings are rules
Ref-Finder Description

- Input: Two versions of a program
- Basic facts
  - i.e. method(methodFullName, methodShortName, typeFullName)
- Deleted_ und added_ facts
- Similarbody facts
- Refactorings are rules
- Topological sort
Ref-Finder Example Rule

Example

- deleted_method(mFullName, mShortName, t1FullName) ∧
Ref-Finder Example Rule

Example

- deleted_method(mFullName, mShortName, t1FullName) ∧
- added_method(newmFullName, mShortName, t2FullName) ∧
Ref-Finder Example Rule

Example

- deleted_method(mFullName, mShortName, t1FullName) ∧
- added_method(newmFullName, mShortName, t2FullName) ∧
- similar_body(newmFullName, newmBody, mFullName, mBody) ∧
Ref-Finder Example Rule

Example

- `deleted_method(mFullName, mShortName, t1FullName)` ∧
- `added_method(newmFullName, mShortName, t2FullName)` ∧
- `similar_body(newmFullName, newmBody, mFullName, mBody)` ∧
- `NOT(equals(t1FullName, t2FullName))`
Ref-Finder Example Rule

Example

- deleted_method(mFullName, mShortName, t1FullName) ∧
- added_method(newmFullName, mShortName, t2FullName) ∧
- similar_body(newmFullName, newmBody, mFullName, mBody) ∧
- NOT(equals(t1FullName, t2FullName))
- → move_method(mShortName, t1FullName, t2FullName)
Nine of 72 refactorings are not detected
- Including “Substitute Algorithm”
Ref-Finder Problems: Wrong precision

- \( \text{Precision} = \frac{|E \cap R|}{|R|} \)
Ref-Finder Problems: Wrong precision

- Precision $= \frac{|E \cap R|}{|R|}$
- Recall $= \frac{|E \cap R|}{|E|}$

Figure: Ref-Finder precision (Source: Prete et al.)
Ref-Finder Problems: Wrong precision

- **Precision** = \( \frac{|E \cap R|}{|R|} \)
- **Recall** = \( \frac{|E \cap R|}{|E|} \)

**TABLE VI: Results from jEdit, Columba, and Carol (\( \sigma = 0.85 \))**

| Versions | \( |R| \) | \( |E| \) | Prec. | Recall | Types of identified refactorings | Time (min) |
|----------|-------|-------|------|-------|-------------------------------|------------|
| 3.0-3.0.1 | 10    | 9     | 0.75 | 0.78  | remove parameter(1), add parameter(2), **replace magic number with constant**(4), extract method(3) | 0.87       |
| 3.0.1-3.0.2 | 1    | 1     | 1.00 | 1.00  | remove parameter(1)           | 0.67       |
| 3.0.2-3.1  | 214   | 10    | 0.45 | 1.00  | **change unidirectional to bidirectional association**(3), replace magic number with constant(7), replace parameter with method(4), replace nested conditional with guard clauses(3), hide delegate(4), introduce null object(4), change bidirectional to unidirectional association(1), separate query from modifier(1), consolidate conditional expression(7), remove middle man(4), replace exception with test(1), inline method(5), remove parameter(89), add parameter(73), extract method(4), move method(4) | 12.37      |

**Figure:** Ref-Finder precision (Source: Prete et al.)
Ref-Finder Problems: Wrong precision

- Precision = \( \frac{|E \cap R|}{|R|} \)
- Recall = \( \frac{|E \cap R|}{|E|} \)

| \( |R| \) | \( |E| \) | Prec. | Recall |
|---|---|---|---|
| 10 | 9  | 0.75 | 0.78 |
| 1  | 1  | 1.00 | 1.00 |
| 214 | 10 | 0.45 | 1.00 |

**Figure**: Ref-Finder precision (Source: Prete et al.)
Ref-Finder Problems: Questionable Recall

Recall:

“Since it is hard to find known refactorings, we ran REF-FINDER using similarity threshold $\sigma = 0.65$ and manually inspected randomly chosen refactorings until we found 10 correct refactorings. We then measured a recall against this data set at a more reasonable threshold, $\sigma = 0.85$.”

— Prete et al.
Xing and Strouliia (2005)
- Structural-differencing algorithm
- Name-similarity and Structure-similarity heuristics
- Supports 33 Refactorings
RefactoringCrawler

- Dig et al. (2006)
- Syntactic analysis using Shingles encoding
- Semantic analysis
- Supports seven Refactorings
Outline

Introduction

How to find Refactorings?

Repository Mining Tools

Planned Approach
Planned Approach

- Examine precision and recall of Ref-Finder

- Analyze results of studies using Ref-Finder

- Examine precision and recall of other tools and methods

- Analyze results of other studies
Planned Approach

- Examine precision and recall of Ref-Finder
- Analyze results of studies using Ref-Finder
Planned Approach

- Examine precision and recall of Ref-Finder
- Analyze results of studies using Ref-Finder
- Examine precision and recall of other tools and methods
Planned Approach

- Examine precision and recall of Ref-Finder
- Analyze results of studies using Ref-Finder
- Examine precision and recall of other tools and methods
- Analyze results of other studies
Further possibilities: Time pressure and releases

Recommendation

“The other time you should avoid refactoring is when you are close to a deadline. At that point the productivity gain from refactoring would only appear after the deadline and thus be too late.”

— Martin Fowler

Studies equate time pressure with major releases

Questionable for Open-source software
Recommendation

“The other time you should avoid refactoring is when you are close to a deadline. At that point the productivity gain from refactoring would only appear after the deadline and thus be too late.”

— Martin Fowler

- Studies equate time pressure with major releases
- Questionable for Open-source software
Further possibilities: Bug-inducing vs. Bug-fix-inducing changes

- Are refactorings really responsible for bugs?
- Or do they help finding bugs?
Summary

- There are four basic methods to gather empirical data about refactoring
- Tools to detect refactorings are necessary but flawed
- Goal: How valid are the results of refactoring studies?
For Further Reading

Martin Fowler
*Refactoring: Improving the Design of Existing Code.*

Murphy-Hill, Black, Dig, Parnin
Gathering Refactoring Data: A Comparison of Four Methods

Prete, Rachatasumrit, Sudan, Kim
Template-based Reconstruction of Complex Refactorings
Thank You!