Multi-Objective Optimization for Software Refactoring and Evolution

Research Proposal
in partial fulfillment of the requirements
for the degree Philosophiæ Doctor (Ph.D.) in computer science

Ali Ouni

Advisor : Houari Sahraoui, Université de Montréal
Co-advisor: Marouane Kessentini, Missouri University of Science and Technology

December 7, 2012
Outline

• Context
• Research methodology
• Proposal
  – Detection of design defects
  – Correction of design defects
  – Refactoring effect
• Research schedule
• Conclusion
Context

- Software is complex
- It changes frequently
  - Add new functionalities
  - Correcting bugs
  - Adaptation to environment changes
- Easiness to accommodate changes depends on software quality
  - Refactoring
Refactoring

• Refactoring
  – The process of improving a code after it has been written by changing its internal structure without changing the external behavior (Fowler et al., ‘99)
  – Examples: Move method, extract class, move attribute, ...

• Two refactoring steps
  1. detection of code fragments to improve (e.g., design defects)
  2. identification of refactoring solutions
Step 1: Design defects detection

- **Design defect** introduced during the initial design or during evolution
  
  - Anomalies, anti-patterns, bad smells…
  
  - Design situations that adversely affect the development of a software

  - Examples: *Blob, spagheti code, functional decomposition, ...*
The Blob example

• Definition
  – Procedural-style design leads to one object with numerous responsibilities and most other objects only holding data or executing simple processes.

• Symptoms
  – A Blob is a controller class, abnormally large, with almost no parents and no children. It mainly uses data classes, i.e. very small classes with almost no parents and no children (Brown et al. ’98).
Step 2: Refactoring

Move method
Extract class
Move field
Add association
…
Thesis objectives...

• Generate design defects detection rules
• Find refactoring solutions taking into consideration many objectives
  – Quality
  – Effort
  – Semantics correctness
  – Similarity with good refactorings applied in the past to similar contexts
• Understand the impact of refactoring on design defects during software evolution
RQ1. To what extent do refactorings induce new design defects?

RQ2. To what extent does fixing specific design defect types can induce correcting other defects implicitly?
**Detection step**

- **Generation of detection rules (GA)**
  - Input: List of possible refactorings
  - Output: Proposed refactorings

- **Correction step**
  - **Search-based Refactoring (NSGA-II)**
    - Input: Source code + call graph
    - Output: Semantic measures
  - **Effort approximation**
    - Input: Refactoring efforts
    - Output: Recorded refactorings
  - **Semantic similarity measures**
    - Input: Design defects detection rules
    - Output: Recorded refactorings

**Effect study**

- **RQ1.** To what extent do refactorings induce new design defects?
- **RQ2.** To what extent does fixing specific design defect types can induce correcting other defects implicitly?
Existing work

• Design defects detection
  – Manual  (Brown et al. ‘98, Fowler and Beck ‘99)
  – Metrics-based  (Marinescu et al. ’04, Salehie et al. ’06, Maiga et al.’12)
  – Visual  (Dhambri et al. ’08, Langelier et al. ’05)
  – Symptoms-based  (Moha et al. ’08, Murno et al. ‘08)

Definition ➔ symptoms ➔ detection algorithm
Problem

• Detection issues
  – No consensual definition of symptoms
  – The same symptom could be associated to many defect types
  – Difficulty to automate symptom’s evaluation
  – Require an expert to manually write and validate detection rules
• Detection issues
  – Large exhaustive list of quality metrics
  – Huge number of possible threshold values
  – Large systems

Search problem to explore this huge space
Approach overview

Detection step

Examples of defects → Generation of detection rules (GA) → Detection rules

Quality metrics
Genetic algorithm

START

Population of solutions

Evaluation

Optimal or “good” solution found?

Yes

Selection

Crossover Mutation

Detection rules

No

END
Genetic algorithm

- Genetic Algorithm
  1. Generate/refine rule sets randomly
     - Rule = Conditions on metrics
  2. Apply rules
     - Comparing between the detected defects and expected ones
  3. Repeat step 1 and 2 Until (stopping criteria)
- Key elements
  - Representing sets of rules
  - Evaluating a set of rules
  - Deriving a set of rules from other sets of rules
Solution Representation

Defect types

1 : Blob
2 : Spaghetti code
3 : Functional decomposition

New solution generation

1 : If (LOCCCLASS ≥ 4) AND (NOM ≥ 20) OR (WMC > 10) Then Blob
2 : If (LOCMETHOD ≥ 151) Then Spaghetti code
3 : If (NOA ≥ 4) AND (WMC = 16) Then Functional decomposition

Quality metrics

WMC
LCOM
NOM
CBO

NOA

[1..100]
[1..1000]

Solution Representation

Fitness Function

• Average of precision and recall
  – $p$: number of detected classes
  – $t$: number of defects in the base of examples

$$f_{norm} = \frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}$$

$$a_i = \begin{cases} 
1, & \text{if the ith detected classes exists in the base of examples (with the same defect type)} \\
0, & \text{else.} 
\end{cases}$$
Crossover


University of Montreal
Mutation

Validation

• Studied systems

<table>
<thead>
<tr>
<th>Systems</th>
<th>Number of classes</th>
<th>KLOC</th>
<th>Number of defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject v1.10.2</td>
<td>245</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>Xerces-J v2.7.0</td>
<td>991</td>
<td>240</td>
<td>66</td>
</tr>
<tr>
<td>ArgoUML v0.19.8</td>
<td>1230</td>
<td>1160</td>
<td>89</td>
</tr>
<tr>
<td>Quick UML v2001</td>
<td>142</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>LOG4J v1.2.1</td>
<td>189</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>AZUREUS v2.3.0.6</td>
<td>1449</td>
<td>42</td>
<td>93</td>
</tr>
</tbody>
</table>

• Three types of defect
  – Blob, Spaghetti code, Functional decomposition
## Validation

### Detection results

<table>
<thead>
<tr>
<th>System</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject</td>
<td>Blob: 100% SC: 93% FD: 91%</td>
<td>100% 97% 94%</td>
</tr>
<tr>
<td>Xerces-J</td>
<td>Blob: 97% SC: 90% FD: 88%</td>
<td>100% 88% 86%</td>
</tr>
<tr>
<td>ArgoUML</td>
<td>Blob: 93% SC: 88% FD: 82%</td>
<td>100% 91% 89%</td>
</tr>
<tr>
<td>QuickUML</td>
<td>Blob: 94% SC: 84% FD: 81%</td>
<td>98% 93% 88%</td>
</tr>
<tr>
<td>AZUREUS</td>
<td>Blob: 82% SC: 71% FD: 68%</td>
<td>94% 81% 86%</td>
</tr>
</tbody>
</table>
Results & Comparison

• Comparison with DECOR (Moha et al. 2008)

<table>
<thead>
<tr>
<th></th>
<th>Search-based approach</th>
<th>DECOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision-Gantt</td>
<td>87%</td>
<td>59%</td>
</tr>
<tr>
<td>Precision-Xerces</td>
<td>81%</td>
<td>67%</td>
</tr>
</tbody>
</table>
Limitations

• Detection rules depends on the base of examples

• To have good detection rules, we need to have examples from similar contexts

**Detection step**

- **Generation of detection rules (GA)**

**Correction step**

- **Search-based Refactoring (NSGA-II)**
  - Input: Source code + call graph
  - Output: Semantic similarity measures
  - Input: List of previous program versions
  - Output: Similarity with good recorded refactorings
  - Input: Proposed refactorings
  - Output: refactoring efforts

**Effect study**

- **RQ1.** To what extent do refactorings induce new design defects?
- **RQ2.** To what extent does fixing specific design defect types can induce correcting other defects implicitly?
Existing work

• Metric-based approaches
  – Search-based techniques
    • Find the best sequence of refactorings (Harman et al. '07, O’Keeffe et al. '08)

  – Analytic approaches
    • Study of relations between some quality metrics and refactoring changes
      (Sahraoui et al. '00, Du Bois et al. '04, Moha et al. '08)

• Graph-based approaches
  – Graph transformation
    • Software is represented as a graph
    • Refactorings activities as graph production rules
      (Kataoka et al. ’01, Heckel et al. ’95)
Existing work

• Limits
  – Difficult to define "standard" refactoring solutions
  – Difficulty to propose refactoring solutions for each defect type
  – Correct defects separately
  – Do not consider the impact of refactoring
    • Correcting a defect may produce other defects
  – Do not consider the effort dimension
  – The semantic coherence is not concerned
Approach overview

Correction step

A. List of possible refactorings
   Input: Refactoring efforts

B. Design defects detection rules

C. Effort approximation
   Output: refactoring efforts

D. Semantic similarity measures
   Input: Source code + call graph
   Output: semantic measures

E. Similarity with good recorded refactorings
   Input: List of previous program versions
   Output: recorded refactorings

Search-based Refactoring (NSGA-II)

Proposed refactorings
Multi-Objective Refactoring

• See the refactoring task as a multi-objective optimization problem
  – Improve software quality
  – Minimize the effort
  – Preserve the semantics
  – Consider similarity with good recorded code changes to similar contexts

Meta Heuristic Search Using Multi-Objective Optimization (NSGA-II)
NSGA-II overview

- NSGA-II: Non-dominated Sorting Genetic Algorithm (K. Deb et al., '02)

Diagram:
- Parent Population
- Offspring Population
- Non-dominated sorting
- Front F1
- Front F2
- Front F3
- Objective 1
- Objective 2

NSGA-II adaptation

- Representation of the **individuals**
- Creation of a **population** of individuals
- Creation of new individuals using genetic operators (**crossover and mutation**)
- Definition of **fitness functions**
Representation of individuals

• Individual = Refactoring solution
• Sequence of refactoring operations

<table>
<thead>
<tr>
<th>RO1</th>
<th>moveMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO2</td>
<td>pullUpAttribute</td>
</tr>
<tr>
<td>RO3</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO4</td>
<td>inlineClass</td>
</tr>
<tr>
<td>RO5</td>
<td>extractSuperClass</td>
</tr>
<tr>
<td>RO6</td>
<td>inlineMethod</td>
</tr>
<tr>
<td>RO7</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO8</td>
<td>moveMethod</td>
</tr>
</tbody>
</table>
Representation of individuals

- Specify the controlling parameters
  - Random selection for the initial population

<table>
<thead>
<tr>
<th>Refactorings</th>
<th>Controlling parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>move method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>move field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>pull up field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>pull up method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>push down field</td>
<td>(sourceClass, targetClass, field)</td>
</tr>
<tr>
<td>push down method</td>
<td>(sourceClass, targetClass, method)</td>
</tr>
<tr>
<td>inline class</td>
<td>(sourceClass, targetClass)</td>
</tr>
<tr>
<td>extract class</td>
<td>(sourceClass, newClass)</td>
</tr>
</tbody>
</table>
Population creation

- **Population**: set of refactoring solutions

<table>
<thead>
<tr>
<th>RO1</th>
<th>moveMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO2</td>
<td>pullUpAttribute</td>
</tr>
<tr>
<td>RO3</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO4</td>
<td>inlineClass</td>
</tr>
<tr>
<td>RO5</td>
<td>extractSuperClass</td>
</tr>
<tr>
<td>RO6</td>
<td>inlineMethod</td>
</tr>
<tr>
<td>RO7</td>
<td>extractClass</td>
</tr>
<tr>
<td>RO8</td>
<td>moveMethod</td>
</tr>
</tbody>
</table>

**Context.** Methodology. Detection. **Correction.** Refactoring effect. **Schedule.** Conclusion.
1. **Crossover**

- **Solution A**
  - `moveMethod`
  - `pullUpAttribute`
  - `extractClass`
  - `moveMethod`
  - `moveMethod`
  - `PushDownMethod`
  - `inlineClass`

- **Solution B**
  - `extractClass`
  - `moveAttribute`
  - `pullUpAttribute`
  - `moveMethod`
  - `moveMethod`
  - `PushDownMethod`

- **Crossover** (k=3)

- **Child A**
  - `moveMethod`
  - `pullUpAttribute`
  - `extractClass`
  - `moveMethod`
  - `moveMethod`
  - `moveAttribute`

- **Child B**
  - `moveMethod`
  - `moveMethod`
  - `PushDownMethod`
  - `inlineClass`
  - `extractClass`
  - `moveAttribute`
  

2. **Mutation**

- **Mutation** (i=3, j=5)

- **Solution**
  - `moveMethod`
  - `pullUpAttribute`
  - `extractClass`
  - `moveMethod`
  - `moveMethod`
  - `pushDownMethod`
  - `inlineClass`
Fitness functions

1. Quality
   - Minimize the number of detected defects

\[
Quality = \frac{\#\text{detected_defects}}{\#\text{detected_defects\_before\_refactoring}}
\]
Fitness functions

2. Effort

- Minimize code changes
- Given effort values for low-level refactoring
- Aggregated values for high-level refactoring

\[
\text{Effort}_{HLR} = \sum_{i=1}^{p} (n_i \ast \text{Effort}_{LLRi})
\]
## 2. Effort

<table>
<thead>
<tr>
<th>Low Level Refactorings</th>
<th>Create class</th>
<th>Delete class</th>
<th>Add method</th>
<th>Delete method</th>
<th>Add Field</th>
<th>Delete Field</th>
<th>Create Relationship</th>
<th>Delete Relationship</th>
<th>Hide Method</th>
<th>Add Parameter</th>
<th>Remove Parameter</th>
<th>Modify Cardinality</th>
<th>Rename method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effort Value</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Extract Class</td>
<td>l</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Extract interface</td>
<td>l</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Extract method</td>
<td>l</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Extract subclass</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Extract superclass</td>
<td>l</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Inline class</td>
<td>1</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>l</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Inline method</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move class</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Move Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parametrize_Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pull up Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pull up Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push down Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push_down_Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace_subclass_with_field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extract class example

\[ E_{Extract\_Class} = 1 \times E_{Create\_Class} + 2 \times E_{Move\_Field} + 1 \times E_{Create\_Relationship} + 1 \times E_{add\_Method} \]

\[ = 1 \times 2 + 2 \times (E_{add\_Field} + E_{delete\_Field}) + 1 \times 1 + 1 \times 1 \]

\[ = 2 + 2 \times (1 + 2) + 1 + 1 = 10 \]
3. Semantics

- Minimize semantic errors
  - Vocabulary-based similarity (cosine similarity)
  - Dependency-based similarity (shared method call, shared fields)

• Intuition:
  - The correctness of proposed refactorings increase when applied to semantically connected elements.
Fitness Functions

4. Similarity with good recorded code changes applied in the past

\[ Sim\_refactoring\_history(RO_i) = \sum_{j=0}^{n-1} w \times m \]

where \( n \) is the number of possible refactorings to use
\( m \) is the number of times that refactoring has been applied in the past
\( w = \begin{cases} 
2 & \text{if the same refactoring has been applied in the past} \\
1 & \text{if a compatible refactoring has been applied in the past} \\
0 & \text{otherwise} 
\end{cases} \)

• Intuition :
  – Recorded code changes can be used to propose new refactoring solutions in similar contexts.
Evaluation

• Considered Objectives
  – Quality (ICPC 2011)
  – Quality vs Effort (JASE 2012)
  – Quality vs Semantics (ICSM 2012)
  – Quality vs Recorded Changes (CSMR 2013)
  – Quality vs Semantics vs Recorded Changes (CSMR 2013)
Evaluation

• Four research questions:
  
  – **RQ1**: To what extent can the proposed approach correct design defects? *(ICPC 2011)*
  
  – **RQ2**: To what extent can the proposed approach minimizes correction efforts with similar correction rate? *(JASE 2012)*
  
  – **RQ3**: To what extent the proposed approach preserves the semantics with similar correction rate? *(ICSM 2012)*
  
  – **RQ4**: To what extent the used of recorded changes improve the automaton of refactoring with similar correction rate? *(CSMR 2013)*
Evaluation

- Data: Three medium/large open source Java projects

<table>
<thead>
<tr>
<th>Systems</th>
<th>Number of classes</th>
<th>KLOC</th>
<th>Number of Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject v1.10.2</td>
<td>245</td>
<td>31</td>
<td>41</td>
</tr>
<tr>
<td>Xerces-J v2.7.0</td>
<td>991</td>
<td>240</td>
<td>66</td>
</tr>
<tr>
<td>JHotDraw v6.0b1</td>
<td>585</td>
<td>23</td>
<td>21</td>
</tr>
</tbody>
</table>
Evaluation

• Method: Three metrics
  – defect correction ratio (DCR)

  \[
  DCR = \frac{\text{# detected defects}}{\text{# defects before applying refactorings}}
  \]

  – refactoring precision (RP)

  \[
  RP = \frac{\text{# meaningful refactorings}}{\text{# proposed refactorings}}
  \]

  – reused refactorings (RR)

  \[
  RR = \frac{\text{# used refactorings in the base of code changes}}{\text{# refactorings in the base of code changes}}
  \]
# Results

<table>
<thead>
<tr>
<th>Systems</th>
<th>Objectives</th>
<th>DCR</th>
<th>RP</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Quality</td>
<td>Semantics</td>
<td>Refactoring Reuse</td>
<td></td>
</tr>
<tr>
<td>GanttProject v1.10.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>95% (39</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>87% (36</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>93% (38</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>85% (35</td>
</tr>
<tr>
<td>Xerces-J v2.7.0</td>
<td>x</td>
<td>x</td>
<td></td>
<td>89% (59</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>77% (51</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>89% (59</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>76% (50</td>
</tr>
<tr>
<td>JHotDraw v6.0b1</td>
<td>x</td>
<td>x</td>
<td></td>
<td>90% (19</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>81% (17</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>86% (18</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>81% (17</td>
</tr>
</tbody>
</table>
## Results & Comparison

### Comparison

- To what extent the semantics preservation improve refactoring correctness?

<table>
<thead>
<tr>
<th>Systems</th>
<th>Approach</th>
<th>Corrected defects (DCR)</th>
<th>Meaningful refactoring (RP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject v1.10.2</td>
<td>Multi-objective (NSGA-II)</td>
<td>87% (36</td>
<td>41)</td>
</tr>
<tr>
<td></td>
<td>Single-objective (GA)</td>
<td>95% (39</td>
<td>41)</td>
</tr>
<tr>
<td></td>
<td>Kessentini et al. 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Harman et al. 07</td>
<td>N.A</td>
<td>69% (218</td>
</tr>
<tr>
<td>Xerces-J v2.7.0</td>
<td>Multi-objective (NSGA-II)</td>
<td>78% (51</td>
<td>66)</td>
</tr>
<tr>
<td></td>
<td>Single-objective (GA)</td>
<td>89% (59</td>
<td>66)</td>
</tr>
<tr>
<td></td>
<td>Kessentini et al. 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Harman et al. 07</td>
<td>N.A</td>
<td>63% (262</td>
</tr>
</tbody>
</table>
Results

• Effort NSGA-II vs effort GA results comparison
Approach sensitivity

An example of multiple executions on Xerces-J
**Effect study**

**RQ1.** To what extent do refactoring efforts induce new design defects?

**RQ2.** To what extent does fixing specific design defect types can induce correcting other defects implicitly?

---

**Detection step**

**Generation of detection rules (GA)**

- **Input:** List of previous program versions
- **Output:** Detection rules

**Correction step**

**Search-based Refactoring (NSGA-II)**

- **A** List of possible refactorings
- **B** Design defects detection rules
- **C** Effort approximation
- **D** Semantic similarity measures
- **E** Similarity with good recorded refactorings

**Output:**
- Refactoring effort
- Semantic measures
- Recroded refactorings
Related Work

• Relation between refactoring and bugs correction
  – The refactoring is less error-prone than other changes? Correlation between refactoring and bugs correction (Weissgerber et al. ’06)
  – Use of refactoring history information to support bugs prediction (Ratzinger et al. 08)
Problem statement

• Do not investigate correlation between refactoring operation and specific bug types.

• No work to investigate the correlation between refactoring and design defects
  – Do refactoring introduce new design defects?
  – Do refactoring correct implicitly existing design defects?

Correlation between refactoring and design defects?
Questions to answer

• **RQ1**: To what extent do refactorings introduce new design defects?

• **RQ2**: To what extent does fixing specific design defect types can induce correcting other defects implicitly?
Empirical investigation

• **Method:**

  1. **Detection of applied refactorings**
  2. **Apply refactoring**
  3. **Apply detection rules**
  4. **Evaluate the impact of refactoring of design defects**
## Research schedule

<table>
<thead>
<tr>
<th>Step</th>
<th>Period</th>
<th>Status</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course IFT2125, IFT2015</td>
<td>H11</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td>Course IFT6251: Sujets en Génie Logiciel</td>
<td>A11</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td>Course IFT6315: Compréhension et analyse de programmes</td>
<td>H12</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td><strong>Design defects detection and correction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single-objective detection and correction (prove of concepts)</td>
<td>H11</td>
<td>Completed</td>
<td>ICPC 2011: accepted</td>
</tr>
<tr>
<td>Predoc I</td>
<td>A11</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td><strong>Multi-objective search-based refactoring</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-objective search-based refactoring: semantics preservation</td>
<td>H12</td>
<td>Completed</td>
<td>ICSM 2012: accepted</td>
</tr>
<tr>
<td>Search-based Refactoring Using Code Changes</td>
<td>E12 to A12</td>
<td>Completed</td>
<td>CSMR 2013: accepted</td>
</tr>
<tr>
<td>Automating Software Refactoring Using Multiple Criteria: Quality, Semantic Preservation, Effort and Recorded Code Changes</td>
<td>A12 to H13</td>
<td>80%</td>
<td>IEEE Transactions on Software Engineering</td>
</tr>
<tr>
<td>Predoc II</td>
<td>H12</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td><strong>Empirical study: Refactoring Impact on design defects</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empirical study to investigate the relationship between refactoring and correcting/introducing new design defects</td>
<td>H13</td>
<td>0%</td>
<td>FSE 2013</td>
</tr>
<tr>
<td>Empirical study: Refactoring impact on design defects</td>
<td>E13 to A13</td>
<td>0%</td>
<td>ICSE2014; Transactions on Software Engineering and Methodology</td>
</tr>
<tr>
<td>Thesis writing</td>
<td>A13 to H14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Publication list

• Refereed Articles in International Journals

• Refereed Articles in International Conferences
  – Ali Ouni, Marouane Kessentini and Houari Sahraoui, Search-based Refactoring Using Recorded Code Changes, in the 17th European Conference on Software Maintenance and Reengineering (CSMR), march 5-8, 2013, Genova, Italy. (accepted)
  
  

• Publication in progress
Conclusion

• A novel approach to the problem of design-defect detection
  – Generate detection rules from real defect example using GA

• A Multi-objective approach for software refactoring (defects correction)
  – Generate detection rules from real defect example using GA

• Validation
  – Set of java open-source java systems
  – Comparison with existing approaches
  – Promising Results

• Still to do
  – Comparative study with our different objectives
  – Improve the semantics preservation
  – Conduct an empirical study to understand the impact of refactoring
Thanks for your attention
### Semantics problem

#### Employee
- ID
- Name
- FamilyName
- Natinality
- DateOfBirth
- Sex
  -...
- ...

- getPhoneNumber()
- calculateLocalTax()
- getAge()
- calculateSalary()
- setMaritalStatus()
- getCurrentNatinality()
  -...
- ...

#### Car
- IdNumber
- TowingCapacity
- OwnerName
  -...

- getHistoryReport()
- getTowingCapacity()
- setInsuranceNum()
  -...

#### Position
- PositionId
- Grade
- CompanyName
  -...

- getPosition()
- setGrade()
  -...

**defect:** Blob
Refactoring metamodel