Automated Support for Program Refactoring Using Invariants

Yoshio Kataoka (Toshiba)
Michael Ernst (MIT)
William Griswold (UCSD)
David Notkin (UW)

Goal:
Automatically identify refactoring candidates
Refactoring

(Local) program restructuring
Enhance readability, performance, abstraction, maintainability, flexibility, ...
Beloved of Extreme Programming
Example: Extract Method
  • find repeated code
  • replace each occurrence by call to a new method
Refactoring steps

Select a refactoring
   Typically done by hand or via lexical analysis

Apply the refactoring
   Some tool support exists
Identifying refactoring opportunities

Pattern of invariants $\Rightarrow$ refactoring is applicable

An invariant is a program property (as in asserts or specifications)

- $x > \text{abs}(y)$
- $x = 16*y + 4*z + 3$
- array $a$ contains no duplicates
- for each node $n$, $n = n.\text{child.parent}$
- graph $g$ is acyclic
- if $\text{ptr} \neq \text{null}$ then $*\text{ptr} > i$

Invariants are rarely present in practice
Tool architecture

Original program

Invariant Detector (Daikon)

Test suite

Invariant patterns

Invariants

Refactoring Candidate Detector

Refactoring candidates
Dynamic invariant detection

Goal: recover invariants from programs

Technique: run the program, examine values
  • postulate potential invariants
  • check for each set of variable values

Results are likely invariants
Dynamic invariant detection

Implementation: Daikon

http://sdg.lcs.mit.edu/daikon

Experiments indicate *accuracy* and *usefulness*

Recover/prove formal specs, aid programmers

Dynamically detected invariants may identify more refactoring opportunities

Static analysis fails for pointers
Refactorings examined

- Remove Parameter
- Eliminate Return Value
- Separate Query from Modifier
- Encapsulate Downcast
- Replace Temporary Variable by Query

Refactoring catalogs [Opdyke 92, Fowler 99] focus on simple lexical transformations
Remove Parameter

Applicable when parameter is constant or unused

- \( \text{param} = \text{constant} \), or
- \( \text{param} = f(a, b, \ldots) \), where \( a, b, \ldots \) are in scope

Examples:

- \( \text{height} = \text{width} \) for all icons
- \( \text{isAutomaticAspect} = \text{true} \) in \text{Aspect} constructor
- \( \text{SetFirstItemFlag} \) called with constant argument
Eliminate Return Value

Applicable if return value is constant or unused

- \( \text{return} = \text{constant} \), or
- \( \text{return} = f(a, b, ...) \), where \( a, b, ... \) are in scope

Example:

- \( \text{return} = \text{true} \) in \text{MakeObjectObey}
Separate Query from Modifier

Applicable when a method returns a value and has a side effect

• \( \text{return} \neq \text{constant} \), and
• \( a \neq \text{orig}(a) \) for some \( a \) in scope

Example:

• \( \text{mCurrentIndex} = \text{orig(mCurrentIndex)} + 1 \) in \text{CursorHistory.GetNextItem}
Encapsulate Downcast

Applicable when return value needs to be downcasted by the caller

- \( LUB(\text{return.class}) \neq \text{declaredtype}(\text{return}) \)

Approximation:

- \( \text{return.class} = \text{constant} \), and
- \( \text{return.class} \neq \text{declaredtype}(\text{return}) \)

Example:

- \( \text{comboBoxItems.class} = \text{AspectTraverseListItem[]} \) in \text{AspectTraverseComboBox}
Replace Temp. Var. by Query

Applicable when a temporary variable holds the value of an expression

- \( \text{temp} = \text{orig(temp)}, \) and
- \( \text{a} = \text{orig(a)} \) for all vars \( \text{a} \) in initializer of \( \text{temp} \)

Examples found after adding wrapper functions
Case study: Nebulous

A component of Aspect Browser [Griswold 01]
Visualizes cross-cutting aspects of a program
Manages changes to such aspects
Uses pattern matching and the map metaphor
78 files, 7000 non-comment non-blank lines
Case study methodology

Wrote a Perl script to identify invariant patterns in Daikon output
Ran Daikon over Nebulous executions
Ran script to identify refactoring opportunities
Nebulous programmer evaluated the recommendations
## Programmer assessment

<table>
<thead>
<tr>
<th></th>
<th>yes</th>
<th>maybe</th>
<th>no</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove Parameter</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Eliminate Return Value</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Sep. Query from Modifier</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Encapsulate Downcast</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>26</td>
</tr>
</tbody>
</table>

Remove Parameter: singletons, flags (another refactoring)
Eliminate Return Value: test suite, convenience
Separate Query from Modifier: style
Encapsulate Downcast: static count
Evaluation

Tool suggestions revealed architectural flaws, prompted redesign and code simplification.

Easy to filter out poor suggestions:
- No set of rules is right for all users and tasks
- Some are a matter of degree or of style

Maintainer had not previously identified these refactoring opportunities:
- Suggestions orthogonal to clone detection tool
Future work

Add patterns for more refactorings
Perform more case studies
Combine with static analysis
  • Static analysis better for "large method", "variable never used"
  • Refactorings requiring static and dynamic info
  • Compare dynamic and static counts
Combine with tool for applying refactorings
Conclusions

Program invariants effectively identify refactoring candidates

Automatic technique

Justified in terms of run-time properties

Programmer assessment demonstrates utility and ease of use
Questions?