Automated Refactoring of Legacy Java Software to Default Methods

RAFFI KHATCHADOURIAN
CITY UNIVERSITY OF NEW YORK

HIDEHIKO MASUHARA
TOKYO INSTITUTE OF TECHNOLOGY

INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2017
Background: Default Methods in Java 8

- methods with bodies in interfaces
- introduced in Java 8
- useful to improve skeletal implementations
Background: default methods

- methods with bodies in interfaces
- (originally for interface evolution)

```java
Collection
   add(){ ... }

ImmutableList

default void add(E elm) {
    throw new UnsupportedException();
}
```
Background: usefulness of default methods

- alternative to skeletal impl. [Goetz, 2011]

**classical**
* many duplication

**skeletal**

```
Collection
add()
```

```
AbsCollect
add(){...}
```

```
Exception
```

**default methods**
* simple / no dup.

```
Collection
add()
```

```
D
add(){...}
```

```
Exception
```
Background: usefulness of default methods

- alternative to skeletal impl. [Goetz, 2011]

Problems
- Inheritance: single tree only
- Modularity: need to find this
- Bloat: +1 class

skeletal

default methods

* simple / no dup.
Problem: Migration can be Difficult

requiring significant manual effort because

- ubiquitous
- subtle semantic restrictions
  - type-correctness
  - multiple inheritance
  - diff. between class and interface
- tie-breakers
Related: Pull-Up Method Refactoring?
[Fowler99, Tip+11]

- moves methods from a subclass into a super class
  - for reducing redundancy

— Not directly. as it is interfaces
  - multiple inheritance
  - “competition” with classes (tie-breaking)
Related: "Move Original Method to Super Class"? [Borba+04]

- is a law expresses transformational semantic equivalence

- Not for method bodies.

- In our case, no method declarations are being moved but rather bodies
Contributions: a Refactoring Tool

- developed a refactoring tool
- as an Eclipse plugin
- migrates into default methods
- conservative; preserves semantics
- tested with open-source projects
- to count successful/failed cases by applying the tool
- to inquire developers' opinions by sending pull-requests
Approach

- For each candidate method and target interface
  - move the method
  - check preconditions for type-safety and semantic preservation
  - remove the methods with the same body in sibling classes
Contributions: Target Methods with Multiple Source Methods

- Safe to migrate any of them
- Which one to migrate?
- Choose the largest number of “equivalent” source methods

```
Collection
  isEmpty()
  return this.size() == 0;

AbsList
  isEmpty
  return this.size() == 0;

AbsStack
  isEmpty
  return this.size() == 0;

AbsSet
  isEmpty
  int size = this.size();
  return size == 0;
```
Interfaces cannot Declare Instance Fields

Q: In general, how can we guarantee that migration results in a type-correct transformation?
A: Use type constraints to check refactoring preconditions.

[References: Palsberg & Schwartzbach 94, Tip et al. 11]
Preconditions for safety &
semantic preconditions

<table>
<thead>
<tr>
<th>program construct</th>
<th>implied type constraint(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment $E_i = E_j$</td>
<td>$[E_j] \subseteq [E_i]$ (1)</td>
</tr>
</tbody>
</table>
| method call $E.m(E_1, \ldots, E_n)$ to a virtual method $M$ (throwing exceptions $E_{x_1}, \ldots, E_{x_l}$) | $[E.m(E_1, \ldots, E_n)] \triangleq [M]$ (2)  
$[E_j] \subseteq [\text{Param}(M, i)]$ (3)  
$[E] \leq \text{Decl}(M) \lor \ldots \lor [E] \leq \text{Decl}(M_l)$ (4)  
where $\text{RootDefs}(M) = \{M_1, \ldots, M_k\}$  
$\forall E_{x_i} \in \{E_{x_1}, \ldots, E_{x_l}\}$  
$\exists E_{x_h} \in \text{Handle}(E.m(E_1, \ldots, E_n)) \land [E_{x_i}] \leq [E_{x_h}]$ (5) |
| access $E.f$ to field $F$ | $[E.f] \triangleq [F]$ (6)  
$[F] \leq \text{Decl}(F)$ (7) |
| return $E$ in method $M$ | $[E] \subseteq [M]$ (8) |
| $M'$ overrides $M$, $M' \neq M$ | $\text{Param}(M', i) = \text{Param}(M, i)$ (9)  
$[M'] \leq [M]$ (10)  
$\text{Decl}(M') < \text{Decl}(M)$ (11)  
$E \subseteq \text{java.lang.Object}$ (12) |
| for every class (an interface) $I$ | $I \not\subseteq \text{java.lang.Object} \land \forall M([\text{Decl}(M) \triangleq \text{java.lang.Object} \land (\exists \text{interface method } M, \text{overrides } (M', M)]))$ (13) |
| for every interface $I$ | $I \not\subseteq \text{java.lang.Object} \land \forall M([\text{Decl}(M) \triangleq I \land \text{Abstract}(M) \land \forall M'([\text{Decl}(M') \triangleq I \land M' \neq M \implies \neg \text{Abstract}(M')])$ (14) |
| implicit declaration of the method $M$ | $M[D](E) \triangleq \text{Decl}(M)$ (15) |
| implicit declaration of $\text{super}$ in method $M$ | Interface $D$, $D[I]$  
$\text{super} \triangleq \text{super}(\text{Decl}(M))$ (16) |
| implicit declaration of $\text{super}$ in method $M$ | $\text{Decl}(M) < I \implies [\text{I.supei}] \triangleq I$ (17)  
$\forall E_{x_1}, \ldots, E_{x_n} \ldots \subseteq [T]$ (18) |
| expression $\text{new } T(E_1, \ldots, E_n)$ | $\text{Decl}(M) \triangleq T$ (19)  
$\text{Decl}(F) \triangleq T$ (20) |
| declaration of method $M$ (declared in type $T$) | $\text{Decl}(M) \triangleq T$ (21)  
$\text{Decl}(F) \triangleq T$ (22)  
$[F] \triangleq T$ (23)  
$[(T)E] \triangleq T$ (24) |
| declaration of method $M$ declared in interface $I$ | $\exists J, M''([\text{Interface}(J) \land J \not\subseteq I \land J \not\subseteq \text{Decl}(M') \land \forall \text{NoOverides}(M'', M) \land (\text{Default}(M') \lor \text{Default}(M)]) \implies \forall C \mid \text{Class}(C) \land C < I \land C < J \land [\text{M''}[M'' \neq M' \land M'' \neq M] \land \text{Class}(\text{Decl}(M'')) \land C \leq \text{Decl}(M'') \land \text{Public}(M'') \land \neg \text{NoOverides}(M'', M')])$ (25) |
| declaration of concrete type $T$ implementing interface $I$ declaring method $M$ | $\exists M''([T \leq \text{Decl}(M') \land \text{NoOverides}(M, M') \land \neg \text{Abstract}(M') \land \forall M''([T \leq \text{Decl}(M'') < \text{Decl}(M') \land \text{NoOverides}(M'', M')]) \implies \neg \text{Abstract}(M'')]$ (26) |
Preserving Semantics in Light of Multiple Inheritance

new AbsQueue{}
  .removeLast()

where does

dispatches to?

Collection
  
AbsList
  removeLast

Queue
  removeLast
  setSize

AbsQueue
  removeLast

if (!isEmpty())
  this.setSize(
  this.size()-1);

throw
Exception

AbsQueue
  removeLast

if (!isEmpty())
  this.setSize(
  this.size()-1);

throw
Exception

AbsList
  removeLast

Queue
  removeLast
  setSize
Eclipse Plug-in and Case Study

- Implemented as an Eclipse plug-in
- Applied to 19 Java programs
  - how many methods can be migrated?
  - efficient enough?
  - when methods cannot be migrated?
# Eclipse Plug-in and Case Study (Result)

<table>
<thead>
<tr>
<th>subject</th>
<th>KL</th>
<th>KM</th>
<th>cnds</th>
<th>dflts</th>
<th>fps</th>
<th>δ</th>
<th>-δ</th>
<th>tm (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArtOfIllusion</td>
<td>118</td>
<td>6.94</td>
<td>16</td>
<td>1</td>
<td>34</td>
<td>1</td>
<td>0</td>
<td>3.65</td>
</tr>
<tr>
<td>Azureus</td>
<td>599</td>
<td>3.98</td>
<td>717</td>
<td>6</td>
<td>421</td>
<td>4</td>
<td>2</td>
<td>61.83</td>
</tr>
<tr>
<td>Colt</td>
<td>36</td>
<td>3.77</td>
<td>69</td>
<td>4</td>
<td>140</td>
<td>3</td>
<td>0</td>
<td>6.76</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>585</td>
<td>47.87</td>
<td>339</td>
<td>69</td>
<td>644</td>
<td>21</td>
<td>4</td>
<td>83.30</td>
</tr>
<tr>
<td>Java8</td>
<td>291</td>
<td>30.04</td>
<td>247</td>
<td>98</td>
<td>775</td>
<td>25</td>
<td>10</td>
<td>64.66</td>
</tr>
<tr>
<td>JavaPush</td>
<td>6</td>
<td>0.77</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1.02</td>
</tr>
<tr>
<td>JGraph</td>
<td>13</td>
<td>1.47</td>
<td>16</td>
<td>2</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>3.12</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>32</td>
<td>3.60</td>
<td>181</td>
<td>46</td>
<td>282</td>
<td>8</td>
<td>0</td>
<td>1.18</td>
</tr>
<tr>
<td>JUnit</td>
<td>26</td>
<td>3.58</td>
<td>9</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0.79</td>
</tr>
<tr>
<td>MWDumper</td>
<td>5</td>
<td>0.40</td>
<td>11</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0.29</td>
</tr>
<tr>
<td>oscli</td>
<td>11</td>
<td>1.81</td>
<td>13</td>
<td>0</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0.76</td>
</tr>
<tr>
<td>ruphij</td>
<td>2</td>
<td>0.66</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1.10</td>
</tr>
<tr>
<td>setting</td>
<td>506</td>
<td>43.51</td>
<td>776</td>
<td>150</td>
<td>1459 50</td>
<td>13</td>
<td>91.68</td>
<td></td>
</tr>
<tr>
<td>Tomcat</td>
<td>25</td>
<td>3.25</td>
<td>336</td>
<td>31</td>
<td>399</td>
<td>13</td>
<td>0</td>
<td>13.81</td>
</tr>
<tr>
<td>verbose</td>
<td>4</td>
<td>0.55</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.55</td>
</tr>
<tr>
<td>method</td>
<td>11</td>
<td>1.43</td>
<td>181</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0.36</td>
</tr>
<tr>
<td>Violet</td>
<td>27</td>
<td>2.06</td>
<td>104</td>
<td>40</td>
<td>102</td>
<td>5</td>
<td>1</td>
<td>3.54</td>
</tr>
<tr>
<td>Wezzle2D</td>
<td>35</td>
<td>2.18</td>
<td>87</td>
<td>13</td>
<td>181</td>
<td>5</td>
<td>0</td>
<td>4.26</td>
</tr>
<tr>
<td>ZKoss</td>
<td>185</td>
<td>15.95</td>
<td>394</td>
<td>76</td>
<td>684</td>
<td>0</td>
<td>0</td>
<td>33.95</td>
</tr>
<tr>
<td>Totals:</td>
<td>2677</td>
<td>232.2</td>
<td>3321</td>
<td>652</td>
<td>6180 166</td>
<td>30</td>
<td>383.17</td>
<td></td>
</tr>
</tbody>
</table>

18% (30/166) classes can be removed

7 KLOC/s runtime

Automatically migrated 19.6% candidates (652/3321 methods)
Refactoring Precondition Failure Distribution

- Many fails on different preconditions
- Major reason: inaccessible/nonexistent fields/methods
(Preliminary) Pull Request Study

Q: "Is it useful in practice?"

Procedure:
1. Choose GitHub projects
2. Apply refactorings
3. Send pull requests
4. Wait

Result:
- 19 pull requests
  - 4 merged
  - 5 still open
  - 10 rejected

Reasons of rejection:
- no Java 8 yet
- support older clients (Android)
- fear of performance
- ...
# List of Projects in Pull Request Study

<table>
<thead>
<tr>
<th>Merged</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ JSilhouette</td>
<td>▶ Blueocean</td>
</tr>
<tr>
<td>▶ Eclipse Collections</td>
<td>▶ JUnit</td>
</tr>
<tr>
<td>▶ Cyclops React</td>
<td>▶ RxJava</td>
</tr>
<tr>
<td>▶ Bootique</td>
<td>▶ ElasticSearch</td>
</tr>
<tr>
<td>Still open</td>
<td>▶ Guava</td>
</tr>
<tr>
<td>▶ QBit</td>
<td>▶ Spring Framework</td>
</tr>
<tr>
<td>▶ JGit</td>
<td>▶ jOOQ</td>
</tr>
<tr>
<td>▶ Java8 Commons</td>
<td>▶ Java Design Patterns</td>
</tr>
<tr>
<td>▶ Koral</td>
<td>▶ Jetty</td>
</tr>
<tr>
<td>▶ Dari</td>
<td></td>
</tr>
<tr>
<td>▶ Binnavi</td>
<td></td>
</tr>
</tbody>
</table>
A Thought: Evaluation Methods of New Language Features

**Autopsy**
- Investigating GitHub repo's.
  - state of the art
  - scales
  - can see adopted cases only

**Proactive**
- Sending pull requests
  - this work
  - manual effort
  - can learn reasons of rejection
Summary

- A refactoring approach from skeletal implementation to default methods
  - efficient, fully-automated, semantics-preserving
  - based on type constraints
  - implemented as an Eclipse IDE plug-in

- Evaluated
  - refactored 19.63% of methods in 19 projects
  - 4 pull requests merged into 19 projects