Automated Refactoring of Legacy Java Software to Default Methods

Raffi T. Khatchadourian  
CUNY Hunter College

Hidehiko Masuhara  
Tokyo Institute of Technology

How does access to this work benefit you? Let us know!

Follow this and additional works at: http://academicworks.cuny.edu/hc_pubs

🔗 Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation


This Poster is brought to you for free and open access by the Hunter College at CUNY Academic Works. It has been accepted for inclusion in Publications and Research by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
Java 8 introduces enhanced interfaces, allowing for default (instance) methods that implementers will inherit if none are provided [3]. Default methods can be used [2] as a replacement for the skeletal implementation pattern [1], which creates abstract skeletal implementation classes that implementers extend. Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort due to subtle language and semantic restrictions. It requires preserving type correctness by analyzing complex type hierarchies, resolving issues arising from multiple inheritance, reconciling differences between class and interface methods, and ensuring type breakers with overriding class methods do not alter semantics.

We propose an efficient, fully-automated, semantics-preserving refactoring approach, based on type constraints [4,5] and implemented as an open source Eclipse plug-in, that assists developers in taking advantage of enhanced interfaces. It identifies instances of the pattern and safely migrates class method implementations to interfaces as default methods.

### Motivating Example

```
Consider the following interface:

interface Collection<E> {
    void add(E elem); // optional
    void removeLast();
    boolean isEmpty();
    int capacity();
}
```

And the following skeletal implementation classes:

```
abstract class AbstractCollection<E> implements Collection<E>,
    Object { // instance fields.
    int size = 0;
    E[] elems = new E[8]; // instance fields.
    }
```

```
@Override
public void add(E elem) {  // inherits skeletal implementations:
    if (size < elems.length) { // allows adding
        elems[size] = elem;
        size = size + 1;
    }
}
```

```
public void removeLast() { // inherits skeletal implementations:
    if (size > 0) { // allows removing
        E elem = elems[size-1];
        size = size - 1;
    }
}
```

```
public boolean isEmpty() {return this.size == 0;}
```

```
public int capacity() {return this.size.length();}
```

Implementers now extend AbstractCollection to inherit the skeletal implementations of Collection:

```
// inherit skeletal implementations
this.size = return new AbstractArrayList<>();
```

```
And override its methods as needed:

list.add(new int[1]) throws Exception;
list.add(x); // optional
```

```
This pattern has several drawbacks, including:
```

- Inheritance. Implementers extending AbstractCollection cannot further extend.
- Modularity. No syntactic path between Collection and AbstractCollection.
- Bloating libraries. Skeletal implementation classes must be separate from interfaces.
```

### Refactoring Approach

Can we refactor abstract skeletal implementation classes to instead utilize default methods?

```
public class AbstractCollection<E> implements Collection<E>,
    Object { // instance fields.
    int size = 0;
    E[] elems = new E[8]; // instance fields.
    }
```

```
@Override
public void add(E elem) { // inherits skeletal implementations:
    if (size < elems.length) { // allows adding
        elems[size] = elem;
        size = size + 1;
    }
}
```

```
@Override
public void removeLast() { // inherits skeletal implementations:
    if (size > 0) { // allows removing
        E elem = elems[size-1];
        size = size - 1;
    }
}
```

```
@Override
public boolean isEmpty() {return this.size == 0;}
```

```
@Override
public int capacity() {return this.size.length();}
```

1. Refactored version.

```
```

### Inferred Type Constraints

Type constraints are used to determine if a possible migration will either result in a type-incorrect or semantically different program. For example:

- Migrating size() and capacity() from AbstractCollection to Collection implies that [this.size] = Collection. violating constraint (6) that [this.size] ≤ [AbstractCollections.size]...

### Evaluation

<table>
<thead>
<tr>
<th>Subject</th>
<th>KC</th>
<th>BOM</th>
<th>cnds</th>
<th>dflts</th>
<th>fps</th>
<th>su</th>
<th>cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>118</td>
<td>0.94</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Avro</td>
<td>508</td>
<td>3.98</td>
<td>747</td>
<td>118</td>
<td>1306</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>Cola</td>
<td>30</td>
<td>3.77</td>
<td>49</td>
<td>4</td>
<td>143</td>
<td>3</td>
<td>0.76</td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>508</td>
<td>47.67</td>
<td>330</td>
<td>60</td>
<td>644</td>
<td>24</td>
<td>4.30</td>
</tr>
<tr>
<td>Jython</td>
<td>200</td>
<td>29.99</td>
<td>299</td>
<td>42</td>
<td>775</td>
<td>28</td>
<td>10</td>
</tr>
<tr>
<td>JavaPath</td>
<td>8</td>
<td>0.77</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1.02</td>
</tr>
<tr>
<td>JGraph</td>
<td>13</td>
<td>1.47</td>
<td>6</td>
<td>2</td>
<td>21</td>
<td>1</td>
<td>3.12</td>
</tr>
<tr>
<td>JDraw</td>
<td>32</td>
<td>3.50</td>
<td>151</td>
<td>46</td>
<td>282</td>
<td>2</td>
<td>7.75</td>
</tr>
<tr>
<td>JSoup</td>
<td>20</td>
<td>3.50</td>
<td>8</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0.79</td>
</tr>
<tr>
<td>MLlib</td>
<td>5</td>
<td>0.40</td>
<td>184</td>
<td>1</td>
<td>259</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>neo4j</td>
<td>10</td>
<td>1.81</td>
<td>13</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>0.76</td>
</tr>
<tr>
<td>nlp</td>
<td>2</td>
<td>0.36</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1.20</td>
</tr>
<tr>
<td>Spring</td>
<td>508</td>
<td>53.51</td>
<td>775</td>
<td>150</td>
<td>1458</td>
<td>53</td>
<td>12</td>
</tr>
<tr>
<td>Tarmac</td>
<td>178</td>
<td>16.15</td>
<td>233</td>
<td>31</td>
<td>389</td>
<td>13</td>
<td>13.62</td>
</tr>
<tr>
<td>uvm</td>
<td>4</td>
<td>0.95</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.95</td>
</tr>
<tr>
<td>Vfs</td>
<td>16</td>
<td>0.55</td>
<td>10</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>0.36</td>
</tr>
<tr>
<td>Violent</td>
<td>27</td>
<td>2.36</td>
<td>104</td>
<td>40</td>
<td>102</td>
<td>5</td>
<td>1.54</td>
</tr>
<tr>
<td>WeakIdc</td>
<td>35</td>
<td>2.18</td>
<td>87</td>
<td>13</td>
<td>181</td>
<td>5</td>
<td>0.46</td>
</tr>
<tr>
<td>Zhcon</td>
<td>108</td>
<td>15.66</td>
<td>594</td>
<td>76</td>
<td>26</td>
<td>6</td>
<td>0.19</td>
</tr>
<tr>
<td>Ztrack</td>
<td>2077</td>
<td>222.7</td>
<td>2152</td>
<td>513</td>
<td>4180</td>
<td>160</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 1: Experimental results.

Able to automatically migrate 19.63% (column diffs) of candidate methods despite of its conservatism.

### Preliminary Pull Request Study

Submitted 19 pull requests to Java projects on GitHub, of which 4 have been successfully merged, 5 are open, and 10 were closed without merging. Merged projects totaled 163 watches, 1071 stars, and 180 forks. Projects rejecting requests citing reasons such as that they needed to support older Java clients.

### Conclusion

We have presented an efficient, fully-automated, type constraint-based, semantics-preserving approach, featuring an exhaustive runtime selection algorithm that migrates the skeletal implementation pattern in legacy Java code into instead use default methods. It is implemented as an Eclipse IDE plug-in and was evaluated on 19 open source projects. The results show that our tool scales and was able to refactor, despite its conservativeness and language constraints, 19.63% of all methods possibly participating in the pattern with minimal intervention. Our study highlights pattern usage and gives insight to language designers on applicability to existing software.

### References


Automated Refactoring of Legacy Java Software to Default Methods

Raffi Khatchadourian¹ Hidehiko Masuhara²

¹Hunter College & the Graduate Center, City University of New York
²Tokyo Institute of Technology