Spring 5-24-2017

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

AUTOMATED REFACTORING OF LEGACY JAVA SOFTWARE TO DEFAULT METHODS

Raffi Khatchadourian¹ Hidehiko Masuhara²
International Conference on Software Engineering, 2017

¹Computer Science, Hunter College & the Graduate Center, City University of New York, USA
²Mathematical and Computing Science, Tokyo Institute of Technology, Japan
MOTIVATION
Traditionally, an interface is a Java type that lists method declarations.

interface Collection<E> {
    int size();
    void add(E elem);
    boolean isEmpty();
    int capacity();
    abstract boolean atCapacity();
}
Traditionally, an **interface** is a Java type that lists method declarations.

Clients are guaranteed that concrete interface **implementers** provide **implementations** for all listed methods.

```java
interface Collection<E> {
    int size();
    void add(E elem);
    boolean isEmpty();
    int capacity();
    abstract boolean atCapacity();
}
```
· Interface methods can be listed as **optional** operations.

```java
interface Collection<E> {
    // ...
    void add(E elem); /* optional */
}
```
- Interface methods can be listed as **optional** operations.
- Implementers may choose to support them or not.

```java
interface Collection<E> {
    // ...
    void add(E elem); /* optional */
}
```

```java
class ImmutableList<E> implements Collection<E> {
    // ...
}
```
Some Interface Methods are Optional

- Interface methods can be listed as **optional** operations.
- Implementers may choose to support them or not.
- If operations are unsupported, they conventionally **throw** an UnsupportedOperationException.

```java
interface Collection<E> {
    // ...
    void add(E elem); /* optional */
}
```

```java
class ImmutableList<E> implements Collection<E> {
    // ...
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}
```
SKELETAL IMPLEMENTATION CLASSES HELP IMPLEMENT INTERFACES

- The **skeletal implementation** design pattern [Bloch, 2008] is used to make implementing interfaces easier.

```java
abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) { throw new UnsupportedOperationException(); }
}

class ImmutableList<E> extends AbstractImmutableList<E> {
    // ...
    @Override public void add(E elem) { throw new UnsupportedOperationException(); }
}
• The **skeletal implementation** design pattern [Bloch, 2008] is used to make implementing interfaces easier.

• Abstract skeletal implementation class provides partial implementations.

```java
abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}
```

```java
class ImmutableList<E> extends AbstractImmutableList<E> {
    // ...
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}
```
The **skeletal implementation** design pattern [Bloch, 2008] is used to make implementing interfaces easier.

Abstract skeletal implementation class provides partial implementations.

Implementers extend the skeletal implementation class rather than directly implementing the interface.

```java
abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}

class ImmutableList<E> extends AbstractImmutableList<E> {
    // ...
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}
```
The skeletal implementation pattern has several drawbacks:

**Inheritance** `ImmutableList` cannot:

- Subclass another class.
- Inherit skeletal implementations split over multiple classes [Horstmann, 2014].
- Inherit skeletal implementations for multiple interfaces.

**Modularity** No syntactic path between `Collection` and `AbstractCollection` (may require global analysis [Khatchadourian et al., 2016]).

**Bloat**

- Separate classes can complicate libraries, making maintenance difficult.
- Method declarations needed in both interface and abstract class.
Java 8 default methods can replace skeletal implementations.

- Java 8 enhanced interfaces allow both method declarations and definitions.

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();
    }
}
```
Java 8 enhanced interfaces allow both method declarations and definitions.
Implementers inherit the (default) implementation if none provided.

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {}
```
Java 8 enhanced interfaces allow both method declarations and definitions.
- Implementers inherit the (default) implementation if none provided.
- Original motivation to facilitate interface evolution.

```java
interface Collection<E> {
    default void add(E elem) {
        // optional.
        throw new UnsupportedOperationException();
    }
}

class ImmutableList<E> implements Collection<E> {
}
```
Java 8 enhanced interfaces allow both method declarations and definitions.
Implementers inherit the (default) implementation if none provided.
Original motivation to facilitate interface evolution.
Can also be used as a replacement of the skeletal implementation pattern [Goetz, 2011].

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {}

abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();}
}
Using default methods:

**Inheritance** `ImmutableList` can:
- Subclass another class.
- Inherit centralized default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does not require **global** analysis).

**Bloat**
- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in **both** interface and abstract class.
Using default methods:

**Inheritance** `ImmutableList` can:
- Subclass another class.
- Inherit centralized default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does not require global analysis).

**Bloat**
- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in both interface and abstract class.
Using default methods:

**Inheritance** `ImmutableList` can:
- Subclass another class.
- Inherit *centralized* default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does not require *global* analysis).

**Bloat**
- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in both interface and abstract class.
Using **default** methods:

**Inheritance** *ImmutableList* can:
- Subclass another class.
- Inherit *centralized* default methods for an interface.
- Inherit default methods for *each* interface.

**Modularity** No need to find default implementations (does not require *global* analysis).

**Bloat**
- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in *both* interface and abstract class.
Using default methods:

**Inheritance** `ImmutableList` can:

- Subclass another class.
- Inherit *centralized* default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does not require *global* analysis).

**Bloat**

- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in both interface and abstract class.
Using default methods:

**Inheritance** *ImmutableList* can:

- Subclass another class.
- Inherit *centralized* default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does *not* require *global* analysis).

**Bloat**

- No separate classes to complicate libraries, making maintenance *easier*.
- No method declarations needed in *both* interface and abstract class.
Using default methods:

**Inheritance** `ImmutableList` can:
- Subclass another class.
- Inherit *centralized* default methods for an interface.
- Inherit default methods for each interface.

**Modularity** No need to find default implementations (does not require *global* analysis).

**Bloat**
- No separate classes to complicate libraries, making maintenance easier.
- No method declarations needed in both interface and abstract class.
Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort, especially in large and complex projects.

- Skeletal implementation pattern is ubiquitous, particularly in frameworks.
- Subtle language and semantic interface restrictions.
- Requires:
  - Preserving type-correctness by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort, especially in large and complex projects.

- Skeletal implementation pattern is ubiquitous, particularly in frameworks.
- Subtle language and semantic interface restrictions.
- Requires:
  - Preserving type-correctness by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort, especially in large and complex projects.

- Skeletal implementation pattern is ubiquitous, particularly in frameworks.
- Subtle language and semantic interface restrictions.
- Requires:
  - Preserving type-correctness by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
MIGRATION TO INTERFACES AS DEFAULT METHODS CAN BE DIFFICULT

Migrating **legacy code** using the skeletal implementation pattern to **instead** use default methods can require **significant manual effort**, especially in large and complex projects.

- Skeletal implementation pattern is **ubiquitous**, particularly in frameworks.
- Subtle language and semantic interface **restrictions**.
- Requires:
  - Preserving **type-correctness** by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating **legacy code** using the skeletal implementation pattern to **instead** use default methods can require **significant manual effort**, especially in large and complex projects.

- Skeletal implementation pattern is **ubiquitous**, particularly in frameworks.
- Subtle language and semantic interface **restrictions**.
- Requires:
  - Preserving **type-correctness** by analyzing possibly **complex type hierarchies**.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort, especially in large and complex projects.

- Skeletal implementation pattern is ubiquitous, particularly in frameworks.
- Subtle language and semantic interface restrictions.
- Requires:
  - Preserving type-correctness by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating legacy code using the skeletal implementation pattern to instead use default methods can require significant manual effort, especially in large and complex projects.

- Skeletal implementation pattern is ubiquitous, particularly in frameworks.
- Subtle language and semantic interface restrictions.
- Requires:
  - Preserving type-correctness by analyzing possibly complex type hierarchies.
  - Resolving issues arising from multiple inheritance.
  - Reconciling possibly minute differences between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not alter semantics.
Migrating **legacy code** using the skeletal implementation pattern to **instead** use default methods can require **significant manual effort**, especially in large and complex projects.

- Skeletal implementation pattern is **ubiquitous**, particularly in frameworks.
- Subtle language and semantic interface **restrictions**.
- Requires:
  - Preserving **type-correctness** by analyzing possibly **complex type hierarchies**.
  - Resolving issues arising from **multiple inheritance**.
  - Reconciling possibly **minute differences** between class and interface methods.
  - Ensuring tie-breakers with overriding class methods do not **alter semantics**.
· Pull Up Method refactoring [Fowler, 1999; Tip et al., 2011] safely moves methods from a subclass into a super class.
· Goal is solely to reduce redundant code.
· Java has **multiple interface** inheritance.
· More complicated type hierarchy involving interfaces.
· “Competition” with classes (tie-breaking).
· Differences between class method headers (sources) and corresponding interface method declarations (targets).
“Move Original Method to Super Class” law [Borba et al., 2004] expresses transformational semantic equivalence.

In our case, no method declarations are being moved but rather bodies.
OUR CONTRIBUTION
interface Collection<E> {
    boolean isEmpty();}

abstract class AbsList<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;}
}

abstract class AbsStack<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;}
}

abstract class AbsSet<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        int size = this.size(); return size == 0;}
}
TARGET METHODS WITH MULTIPLE SOURCE METHODS

interface Collection\<E\> {  
    boolean isEmpty();}

abstract class AbsList\<E\> implements Collection\<E\> {  
    @Override public boolean isEmpty() {  
        return this.size() == 0;}}

abstract class AbsStack\<E\> implements Collection\<E\> {  
    @Override public boolean isEmpty() {  
        return this.size() == 0;}}

abstract class AbsSet\<E\> implements Collection\<E\> {  
    @Override public boolean isEmpty() {  
        return size == 0;}}

- May not have a one-to-one correspondence between source and target methods.
TARGET METHODS WITH MULTIPLE SOURCE METHODS

```java
interface Collection<E> {
    boolean isEmpty();
}

abstract class AbsList<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;
    }
}

abstract class AbsStack<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;
    }
}

abstract class AbsSet<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        int size = this.size(); return size == 0;
    }
}

- May not have a one-to-one correspondence between source and target methods.
- Migrating any of the source methods passing preconditions would be safe.
```
interface Collection<E> {
    boolean isEmpty();
}

abstract class AbsList<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;
    }
}

abstract class AbsStack<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;
    }
}

abstract class AbsSet<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        int size = this.size();
        return size == 0;
    }
}

- May not have a one-to-one correspondence between source and target methods.
- Migrating any of the source methods passing preconditions would be safe.
- Choose the largest number of “equivalent” source methods.
TARGET METHODS WITH MULTIPLE SOURCE METHODS

interface Collection<E> {
    default boolean isEmpty() {return this.size() == 0;}}

abstract class AbsList<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;}
}

abstract class AbsStack<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        return this.size() == 0;}
}

abstract class AbsSet<E> implements Collection<E> {
    @Override public boolean isEmpty() {
        int size = this.size(); return size == 0;}
}

· May not have a one-to-one correspondence between source and target methods.
· Migrating any of the source methods passing preconditions would be safe.
· Choose the largest number of “equivalent” source methods.
Interfaces cannot declare instance fields.

```java
interface Collection<E> {
    int size();
}

abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() {return this.size;}
}
```

- Migrate `AbsList.size()` to `Collection` as a default method?

Question: In general, how can we guarantee that migration results in a type-correct transformation?

Answer: Use type constraints [Palsberg and Schwartzbach, 1994; Tip et al., 2011] to check refactoring preconditions.
Interfaces cannot declare instance fields.

```java
interface Collection<E> {
    default int size() {return this.size;}
}
```

Abstract class `AbsList<E>` implements `Collection<E>` {
```java
Object[] elems; int size;
@override public int size() {return this.size;}
```

- Migrate `AbsList.size()` to `Collection` as a default method?
Interfaces cannot declare instance fields.

```
interface Collection<E> {
    default int size() {return this.size;}  
}

abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() {return this.size;}  
}
```

- Migrate AbsList.size() to Collection as a default method?
- size() accesses instance fields; migrate them to Collection?
INTERFACES CANNOT DECLARE INSTANCE FIELDS

interface Collection<E> {
    Object[] elems; int size;
    default int size() {return this.size;}}

abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() {return this.size;}}

- Migrate AbsList.size() to Collection as a default method?
- size() accesses instance fields; migrate them to Collection?
Interfaces cannot declare instance fields.

```java
interface Collection<E> {
    Object[] elems; int size;
    default int size() { return this.size; }
}
```

```java
abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() { return this.size; }
}
```

- Migrate `AbsList.size()` to `Collection` as a default method?
- `size()` accesses instance fields; migrate them to `Collection`?
- Interfaces cannot declare instance fields.
interfaces cannot declare instance fields

```java
interface Collection<E> {
    Object[] elems; int size;
    default int size() {return this.size;}
}
```

```java
abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() {return this.size;}
}
```

- Migrate AbsList.size() to Collection as a default method?
- size() accesses instance fields; migrate them to Collection?
- Interfaces cannot declare instance fields.
INTERFACES CANNOT DECLARE INSTANCE FIELDS

interface Collection<E> {
  Object[] elems; int size;
  default int size() {return this.size;}
}

abstract class AbsList<E> implements Collection<E> {
  Object[] elems; int size;
  @Override public int size() {return this.size;}
}

- Migrate AbsList.size() to Collection as a default method?
- size() accesses instance fields; migrate them to Collection?
- Interfaces cannot declare instance fields.

Question

In general, how can we guarantee that migration results in a type-correct transformation?
INTERFACES CANNOT DECLARE INSTANCE FIELDS

interface Collection<E> {
    Object[] elems; int size;
    default int size() {return this.size;}
}

abstract class AbsList<E> implements Collection<E> {
    Object[] elems; int size;
    @Override public int size() {return this.size;}
}

· Migrate AbsList.size() to Collection as a default method?
· size() accesses instance fields; migrate them to Collection?
· Interfaces cannot declare instance fields.

Question

In general, how can we guarantee that migration results in a type-correct transformation?

Answer

Use type constraints [Palsberg and Schwartzbach, 1994; Tip et al., 2011] to check refactoring preconditions.
Type constraints denote the subtyping relationships for each program element that must hold between corresponding expressions for that portion to be considered well-typed.

Migrating `size()` to `Collection` would imply `this` = `Collection`. This violates constraint (2) that `this` `AbsList`.

```java
abstract class AbsList<E> implements Collection<E> {
    @Override public int size() {return this.size;}
}
```
· Type constraints denote the subtyping relationships for each program element that must hold between corresponding expressions for that portion to be considered well-typed.

· A complete program is type-correct if all constraints implied by all program elements hold.
· Type constraints denote the subtyping relationships for each program element that must hold between corresponding expressions for that portion to be considered well-typed.
· A complete program is type-correct if all constraints implied by all program elements hold.

<table>
<thead>
<tr>
<th>program construct</th>
<th>implied type constraint(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access E.f to field F</td>
<td>[E.f] $\triangleq$ [F] \hspace{1cm} (1)  \</td>
</tr>
</tbody>
</table>
Type constraints denote the subtyping relationships for each program element that must hold between corresponding expressions for that portion to be considered well-typed.

A complete program is type-correct if all constraints implied by all program elements hold.

<table>
<thead>
<tr>
<th>Program Construct</th>
<th>Implied Type Constraint(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access E.f to field F</td>
<td>[E.f] ⊑ [F]</td>
</tr>
<tr>
<td></td>
<td>[E] ⊑ Decl(F)</td>
</tr>
</tbody>
</table>

Migrating size() to Collection would imply [this] = Collection.

```java
interface Collection<E> {
    default int size() {return this.size;}{}
}
```
Type constraints denote the subtyping relationships for each program element that must hold between corresponding expressions for that portion to be considered well-typed. A complete program is type-correct if all constraints implied by all program elements hold.

<table>
<thead>
<tr>
<th>program construct</th>
<th>implied type constraint(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access E.f to field F</td>
<td>[E.f] \triangleq [F]</td>
</tr>
<tr>
<td></td>
<td>[E] \leq \text{Decl}(F)</td>
</tr>
</tbody>
</table>

Migrating `size()` to `Collection` would imply `[this] = Collection`.

```java
interface Collection<E> {
    default int size() {return this.size;}
}
```

This violates constraint (2) that `[this] \leq [AbsList]`.

```java
abstract class AbsList<E> implements Collection<E> {
    @Override public int size() {return this.size;}
}
```
Extend [Tip et al., 2011] with **new constraints, new definitions, and semantics preservation** for default methods.

- See paper for more details.
abstract class AbsList<E> implements Collection<E> {
    @Override public void removeLast() {
        throw new UnsupportedOperationException();}
}

interface Queue<E> extends Collection<E> {
    void removeLast();
    void setSize(int i);}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
    @Override public void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);}
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsQueue.

Can we migrate removeLast() from AbsQueue to Queue?

Now dispatches to AbsList as classes take precedence!

Queue loses “tie” with AbsList.

Disallow methods that override in both classes and interfaces.
abstract class AbsList<E> implements Collection<E> {
    @Override public void removeLast() {
        throw new UnsupportedOperationException();
    }
}

interface Queue<E> extends Collection<E> {
    void removeLast();
    void setSize(int i);
}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
    @Override public void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);
    }
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsQueue.

· Can we migrate removeLast() from AbsQueue to Queue?
abstract class AbsList<E> implements Collection<E> {
   @Override public void removeLast() {
      throw new UnsupportedOperationException();}
}

interface Queue<E> extends Collection<E> {
   default void removeLast(); {
      if (!isEmpty()) this.setSize(this.size() - 1);}
   void setSize(int i);}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
   @Override public void removeLast() {
      if (!isEmpty()) this.setSize(this.size() - 1);}
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsQueue.

- Can we migrate removeLast() from AbsQueue to Queue?
abstract class AbsList<E> implements Collection<E> {
    @Override public void removeLast() {
        throw new UnsupportedOperationException();
    }
}

interface Queue<E> extends Collection<E> {
    default void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);
    }
    void setSize(int i);
}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
    @Override public void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);
    }
}

new AbsQueue<Integer>() {}.removeLast(); // to Queue?

- Can we migrate removeLast() from AbsQueue to Queue?
abstract class AbsList<E> implements Collection<E> {
    @Override public void removeLast() {
        throw new UnsupportedOperationException();
    }
}

interface Queue<E> extends Collection<E> {
    default void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);
    }
    void setSize(int i);
}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
    @Override public void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);
    }
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsList.

- Can we migrate removeLast() from AbsQueue to Queue?
- Now dispatches to AbsList as classes take precedence!
abstract class AbsList<E> implements Collection<E> {
    @Override public void removeLast() {
        throw new UnsupportedOperationException();
    }
}

interface Queue<E> extends Collection<E> {
    default void removeLast(); { 
        if (!isEmpty()) this.setSize(this.size() - 1);}
    void setSize(int i);}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
    @Override public void removeLast() {
        if (!isEmpty()) this.setSize(this.size() - 1);}
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsList.

- Can we migrate removeLast() from AbsQueue to Queue?
- Now dispatches to AbsList as classes take precedence!
- Queue loses “tie” with AbsList.
abstract class AbsList<E> implements Collection<E> {
  @Override public void removeLast() {
    throw new UnsupportedOperationException();}
}

interface Queue<E> extends Collection<E> {
  default void removeLast(); {
    if (!isEmpty()) this.setSize(this.size() - 1);
  }
  void setSize(int i);
}

abstract class AbsQueue<E> extends AbsList<E> implements Queue<E> {
  @Override public void removeLast() {
    if (!isEmpty()) this.setSize(this.size() - 1);
  }
}

new AbsQueue<Integer>() {}.removeLast(); // to AbsQueue.

- Can we migrate removeLast() from AbsQueue to Queue?
- Now dispatches to AbsList as classes take precedence!
- Queue loses “tie” with AbsList.
- Disallow methods that override in both classes and interfaces.
## ECLIPSE PLUG-IN AND CASE STUDY

<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>747</td>
<td>116</td>
<td>1366</td>
<td>31</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>30.99</td>
<td>299</td>
<td>93</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>7.75</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>osgi</td>
<td>18</td>
<td>1.81</td>
<td>13</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0.76</td>
</tr>
<tr>
<td>rdp4j</td>
<td>2</td>
<td>0.26</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1.10</td>
</tr>
<tr>
<td>spring</td>
<td>506</td>
<td>53.51</td>
<td>776</td>
<td>150</td>
<td>1459</td>
<td>50</td>
<td>13</td>
<td>91.68</td>
</tr>
<tr>
<td>Tomcat</td>
<td>176</td>
<td>16.15</td>
<td>233</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>VietPad</td>
<td>11</td>
<td>0.58</td>
<td>15</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</td>
<td>166</td>
<td>30</td>
<td>383.17</td>
</tr>
</tbody>
</table>

- Implemented as an open source Eclipse plug-in.
- Evaluated on 19 Java programs of varying size and domain.
- Automatically migrated 19.63% (column `dflts`) of candidate despite conservatism.
- Running time (column `tm (s)`) averaged ~0.144 secs/KLOC.
· Field and method **inaccessibility** from the destination interface accounted for largest number of errors.
· Next largest failure due to **instance field accesses** (failures of constraint (2)).
Submitted 19 pull requests to Java projects on GitHub.
- 4 were successfully merged, 5 are still open, and 10 were closed without merging.
- Merged projects totaled 163 watches, 1071 stars, and 180 forks.
- Projects rejecting requests citing reasons such as:
  - They had not yet moved or were in the process of moving to Java 8.
  - Needed to support older Java clients (Android).
Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.
SUMMARY

• Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.

• Implemented as an Eclipse IDE plug-in (available at http://cuny.is/interefact) and evaluated on 19 open source projects.
Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.

Implemented as an Eclipse IDE plug-in (available at http://cuny.is/interefact) and evaluated on 19 open source projects.

Tool scales and refactored 19.63% of methods possibly participating in the pattern.
Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.

Implemented as an Eclipse IDE plug-in (available at http://cuny.is/interefact) and evaluated on 19 open source projects.

Tool scales and refactored 19.63% of methods possibly participating in the pattern.

4 pull requests merged into GitHub repositories, including large, widely used frameworks from reputable organizations.
SUMMARY

- Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.
- Implemented as an Eclipse IDE plug-in (available at http://cuny.is/interefact) and evaluated on 19 open source projects.
- Tool scales and refactored 19.63% of methods possibly participating in the pattern.
- 4 pull requests merged into GitHub repositories, including large, widely used frameworks from reputable organizations.
- Studies highlight pattern usage and gives possible insight to language designers on construct applicability to existing software.
SUMMARY

- Efficient, fully-automated, semantics-preserving refactoring approach based on type constraints that migrates the skeletal implementation pattern in legacy Java code to instead use default methods.
- Implemented as an Eclipse IDE plug-in (available at http://cuny.is/interefact) and evaluated on 19 open source projects.
- Tool scales and refactored 19.63% of methods possibly participating in the pattern.
- 4 pull requests merged into GitHub repositories, including large, widely used frameworks from reputable organizations.
- Studies highlight pattern usage and gives possible insight to language designers on construct applicability to existing software.
## Precondition Failure Breakdown

<table>
<thead>
<tr>
<th>#</th>
<th>Precondition</th>
<th>Fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>MethodContainsInconsistentParameterAnnotations</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>MethodContainsCallToProtectedObjectMethod</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>TypeVariableNotAvailable</td>
<td>10</td>
</tr>
<tr>
<td>P4</td>
<td>DestinationInterfaceIsFunctional</td>
<td>17</td>
</tr>
<tr>
<td>P5</td>
<td>TargetMethodHasMultipleSourceMethods</td>
<td>19</td>
</tr>
<tr>
<td>P6</td>
<td>MethodContainsIncompatibleParameterTypeParameters</td>
<td>42</td>
</tr>
<tr>
<td>P7</td>
<td>NoMethodsWithMultipleCandidateDestinations</td>
<td>53</td>
</tr>
<tr>
<td>P8</td>
<td>TypeNotAccessible</td>
<td>64</td>
</tr>
<tr>
<td>P9</td>
<td>SourceMethodImplementsMultipleMethods</td>
<td>72</td>
</tr>
<tr>
<td>P10</td>
<td>SourceMethodProvidesImplementationsForMultipleMethods</td>
<td>79</td>
</tr>
<tr>
<td>P11</td>
<td>MethodContainsTypeIncompatibleThisReference</td>
<td>79</td>
</tr>
<tr>
<td>P12</td>
<td>IncompatibleMethodReturnTypes</td>
<td>104</td>
</tr>
<tr>
<td>P13</td>
<td>ExceptionTypeMismatch</td>
<td>105</td>
</tr>
<tr>
<td>P14</td>
<td>MethodContainsSuperReference</td>
<td>147</td>
</tr>
<tr>
<td>P15</td>
<td>SourceMethodOverridesClassMethod</td>
<td>258</td>
</tr>
<tr>
<td>P16</td>
<td>AnnotationMismatch</td>
<td>305</td>
</tr>
<tr>
<td>P17</td>
<td>SourceMethodAccessesInstanceField</td>
<td>463</td>
</tr>
<tr>
<td>P18</td>
<td>MethodNotAccessible</td>
<td>1,679</td>
</tr>
<tr>
<td>P19</td>
<td>FieldNotAccessible</td>
<td>2,565</td>
</tr>
</tbody>
</table>

*Table: Precondition failures.*
FOR FURTHER READING


