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

Recommended Citation

Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy Java software to default methods. In International Conference on Software Engineering, ICSE '17. ACM/IEEE, May 2017.

AUTOMATED REFACTORING OF LEGACY JAVA SOFTWARE TO DEFAULT METHODS

Raffi Khatchadourian[1] Hidehiko Masuhara[2]

International Conference on Software Engineering, 2017

[1]Computer Science, Hunter College & the Graduate Center, City University of New York, USA

[2]Mathematical and Computing Science, Tokyo Institute of Technology, Japan

MOTIVATION

INTERFACES ARE TRADITIONALLY LISTS OF METHOD DECLARATIONS

- Traditionally, an **interface** is a Java type that lists method **declarations**.

```java
interface Collection<E> {
    int size();
    void add(E elem);
    boolean isEmpty();
    int capacity();
    abstract boolean atCapacity();}
```

INTERFACES ARE TRADITIONALLY LISTS OF METHOD DECLARATIONS

- Traditionally, an **interface** is a Java type that lists method **declarations**.
- Clients are guaranteed that concrete interface **implementers** provide **implementations** for all listed methods.

```
interface Collection<E> {
    int size();
    void add(E elem);
    boolean isEmpty();
    int capacity();
    abstract boolean atCapacity();}
```

SOME INTERFACE METHODS ARE OPTIONAL

- Interface methods can be listed as **optional** operations.

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

SOME INTERFACE METHODS ARE OPTIONAL

- Interface methods can be listed as **optional** operations.
- Implementers may choose to support them or not.

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

```
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`.

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

```
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.

SKELETAL IMPLEMENTATION CLASSES HELP IMPLEMENT INTERFACES

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

```
abstract class AbstractImmutableList<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.
- 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();}}
```

```java
class ImmutableList<E> extends AbstractImmutableList<E>{
  // ...
  @Override public void add(E elem) {
    throw new UnsupportedOperationException();}}}
```

THE SKELETAL IMPLEMENTATION PATTERN HAS SEVERAL DRAWBACKS

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**.

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

JAVA 8 DEFAULT METHODS CAN REPLACE SKELETAL IMPLEMENTATIONS

- Java 8 **enhanced** interfaces allow both method declarations and **definitions**.
- Implementers inherit the (`default`) implementation if none provided.

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

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

JAVA 8 DEFAULT METHODS CAN REPLACE SKELETAL IMPLEMENTATIONS

- Java 8 **enhanced** interfaces allow both method declarations and **definitions**.
- Implementers inherit the (`default`) implementation if none provided.
- Original motivation to facilitate interface **evolution**.

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

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

JAVA 8 DEFAULT METHODS CAN REPLACE SKELETAL IMPLEMENTATIONS

- 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].

```
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();}}~~

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

DEFAULT METHODS CAN COMPENSATE FOR DRAWBACKS

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.

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.

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.

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.

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.

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.

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.

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.

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.

PULL UP METHOD REFACTORING NOT DIRECTLY APPLICABLE

- 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 DOES NOT DEAL WITH BODIES

- "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

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() {
    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() {
    int size = this.size(); return size == 0;}}
```

- May not have a one-to-one correspondence between source and target methods.

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() {
    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.

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() {
    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

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

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

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

```java
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

```
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?

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?

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

```
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**.

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**.

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**.

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.

USING TYPE CONSTRAINTS AS 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.

USING TYPE CONSTRAINTS AS 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.
- A complete program is type-correct if all constraints implied by all program elements hold.

USING TYPE CONSTRAINTS AS 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.
- A complete program is type-correct if all constraints implied by all program elements hold.

program construct	implied type constraint(s)
access E.f to field F	$[E.f] \triangleq [F]$ (1) $[E] \leq Decl(F)$ (2)

USING TYPE CONSTRAINTS AS 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.
- A complete program is type-correct if all constraints implied by all program elements hold.

program construct	implied type constraint(s)
access E.f to field F	$[\text{E.f}] \triangleq [\text{F}]$ (1) $[\text{E}] \leq \text{Decl}(\text{F})$ (2)

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

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

USING TYPE CONSTRAINTS AS 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.
- A complete program is type-correct if all constraints implied by all program elements hold.

program construct	implied type constraint(s)	
access E.f to field F	$[E.f] \triangleq [F]$	(1)
	$[E] \leq Decl(F)$	(2)

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

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

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

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

program construct	implied type constraint(s)	
assignment $E_i = E_j$	$[E_j] \leq [E_i]$	(1)
method call $E.m(E_1, \ldots, E_n)$ to a virtual method M (throwing exceptions $Ext_1, \ldots, Ext_j$)	$[E.m(E_1, \ldots, E_n)] \triangleq [M]$ $[E_i] \leq [\mathit{Param}(M, i)]$ $[E] \leq \mathit{Decl}(M_1) \vee \cdots \vee [E] \leq \mathit{Decl}(M_k)$ where $\mathit{RootDefs}(M) = \{M_1, \ldots, M_k\}$ $\forall Ext \in \{Ext_1, \ldots, Ext_j\}$ $\exists Ex_h \in \mathit{Handle}(E.m(E_1, \ldots, E_n))[[Ex_t] \leq [Ex_h]]$	(2) (3) (4) (5)
access $E.f$ to field F	$[E.f] \triangleq [F]$ $[E] \leq \mathit{Decl}(F)$	(6) (7)
return E in method M	$[E] \leq [M]$	(8)
M' overrides M, $M' \neq M$	$[\mathit{Param}(M', i)] = [\mathit{Param}(M, i)]$ $[M'] \leq [M]$ $\mathit{Decl}(M') < \mathit{Decl}(M)$	(9) (10) (11)
for every class (and enum) C	$C \leq \texttt{java.lang.Object}$	(12)
for every interface I	$I \not\leq \texttt{java.lang.Object} \wedge \forall M[\mathit{Decl}(M) \triangleq \texttt{java.lang.Object} \wedge$ $\mathit{Public}(M) \implies \exists M'[\mathit{Decl}(M') \triangleq I \wedge \mathit{NOverrides}(M', M)]]$	(13)
for every functional interface I	$\exists M[\mathit{Decl}(M) \triangleq I \wedge \mathit{Abstract}(M)$ $\wedge \forall M'[\mathit{Decl}(M') \triangleq I \wedge M' \neq M \implies \neg \mathit{Abstract}(M')]]$	(14)
implicit declaration of **this** in method M	$[\textbf{this}] \triangleq \mathit{Decl}(M)$	(15)
implicit declaration of **super** in method M	$\neg \mathit{Interface}(\mathit{Decl}(M)) \implies [\textbf{super}] \triangleq \mathit{super}(\mathit{Decl}(M))$	(16)
implicit declaration of `I.super` in method M	$\mathit{Decl}(M) < I \implies [\texttt{I.}\textbf{super}] \triangleq I$	(17)
expression **new** $T(E_1, \ldots, E_n)$ ...	$[\textbf{new}\ T(E_1, \ldots, E_n)\ \ldots] \leq [T]$	(18)
declaration of method M (declared in type T)	$\mathit{Decl}(M) \triangleq T$	(19)
declaration of field F (declared in type T)	$\mathit{Decl}(F) \triangleq T$	(20)
explicit declaration of variable or method parameter $T\ v$	$[v] \triangleq T$	(21)
declaration of method M with return type T	$[M] \triangleq T$	(22)
declaration of field F with type T	$[F] \triangleq T$	(23)
cast $(T)E$	$[(T)E] \triangleq T$	(24)
declaration of method M declared in interface I	$\exists J, M'[\mathit{Interface}(J) \wedge J \not\leq I \wedge I \nless J \wedge J \leq \mathit{Decl}(M')$ $\wedge \mathit{NOverrides}(M', M) \wedge (\mathit{Default}(M') \vee \mathit{Default}(M))]$ $\implies \forall C \mid \mathit{Class}(C) \wedge C < I \wedge C < J[\exists M''[M'' \neq M' \wedge M'' \neq M$ $\wedge \mathit{Class}(\mathit{Decl}(M'')) \wedge C \leq \mathit{Decl}(M'') \wedge \mathit{Public}(M'')$ $\wedge \mathit{NOverrides}(M'', M')]]$	(25)
declaration of concrete type T implementing interface I declaring method M	$\exists M'[T \leq \mathit{Decl}(M') \wedge \mathit{NOverrides}(M, M')$ $\wedge \neg \mathit{Abstract}(M') \wedge \forall M''[T < \mathit{Decl}(M'') < \mathit{Decl}(M') \wedge$ $\mathit{NOverrides}(M'', M') \implies \neg \mathit{Abstract}(M'')]]$	(26)

Fig. 4. Type constraints for a subset of core Java features.

- Extend [Tip et al., 2011] with **new** constraints, **new** definitions, and **semantics preservation** for **default methods**.
- See paper for more details.

PRESERVING SEMANTICS IN LIGHT OF MULTIPLE INHERITANCE

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

```java
interface Queue<E> extends Collection<E> {
  void removeLast();

  void setSize(int i);}
```

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

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

```
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`?

PRESERVING SEMANTICS IN LIGHT OF MULTIPLE INHERITANCE

```
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`?

PRESERVING SEMANTICS IN LIGHT OF MULTIPLE INHERITANCE

```
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!

PRESERVING SEMANTICS IN LIGHT OF MULTIPLE INHERITANCE

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

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

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

```java
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`.

PRESERVING SEMANTICS IN LIGHT OF MULTIPLE INHERITANCE

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

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

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

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

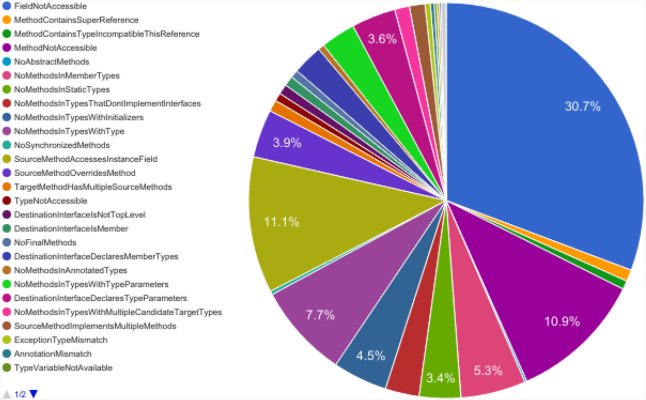
- Can we migrate `removeLast()` from `AbsQueue` to `Queue`?
- Now dispatches to `AbsList` as classes take precedence!
- `Queue` **loses** “tie” with `AbsList`.
- Disallow methods that override in **both** classes and interfaces.

ECLIPSE PLUG-IN AND CASE STUDY

subject	KL	KM	cnds	dflts	fps	δ	-δ	tm (s)
ArtOfIllusion	118	6.94	16	1	34	1	0	3.65
Azureus	599	3.98	747	116	1366	31	2	61.83
Colt	36	3.77	69	4	140	3	0	6.76
elasticsearch	585	47.87	339	69	644	21	4	83.30
Java8	291	30.99	299	93	775	25	10	64.66
JavaPush	6	0.77	1	0	4	0	0	1.02
JGraph	13	1.47	16	2	21	1	0	3.12
JHotDraw	32	3.60	181	46	282	8	0	7.75
JUnit	26	3.58	9	0	25	0	0	0.79
MWDumper	5	0.40	11	0	24	0	0	0.29
osgi	18	1.81	13	3	11	2	0	0.76
rdp4j	2	0.26	10	8	2	1	0	1.10
spring	506	53.51	776	150	1459	50	13	91.68
Tomcat	176	16.15	233	31	399	13	0	13.81
verbose	4	0.55	1	0	1	0	0	0.55
VietPad	11	0.58	15	0	26	0	0	0.36
Violet	27	2.06	104	40	102	5	1	3.54
Wezzle2D	35	2.18	87	13	181	5	0	4.26
ZKoss	185	15.95	394	76	684	0	0	33.95
Totals:	2677	232.2	3321	652	6180	166	30	383.17

- 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 $\sim$0.144 secs/KLOC.

REFACTORING PRECONDITION FAILURE DISTRIBUTION



- 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)).

PRELIMINARY PULL REQUEST STUDY

- 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).

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.

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.

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.

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.

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.
- Graduate positions available! `http://bit.ly/cunygrad`

PRECONDITION FAILURES BREAKDOWN

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

Table: Precondition failures.

FOR FURTHER READING

Joshua Bloch. **Effective Java**. Prentice Hall, 2008.

Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. **Science of Computer Programming**, 52 (1-3):53–100, August 2004. ISSN 0167-6423. doi: 10.1016/j.scico.2004.03.003.

Martin Fowler. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Professional, 1999.

Brian Goetz. Interface evolution via virtual extensions methods. Technical report, Oracle Corporation, June 2011. URL `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf`.

Cay S. Horstmann. **Java SE 8 for the Really Impatient**. Addison-Wesley Professional, 2014.

Raffi Khatchadourian, Olivia Moore, and Hidehiko Masuhara. Towards improving interface modularity in legacy java software through automated refactoring. In **Companion Proceedings of the 15th International Conference on Modularity**, MODULARITY Companion 2016, pages 104–106, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4033-5. doi: 10.1145/2892664.2892681.

Jens Palsberg and Michael I. Schwartzbach. **Object-oriented type systems**. John Wiley and Sons Ltd., 1994. ISBN 0-471-94128-X.

Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. **ACM Transactions on Programming Languages and Systems**, 33(3):9:1–9:47, May 2011. ISSN 0164-0925. doi: 10.1145/1961204.1961205.