2017 IEEE/ACM 39th International Conference on Software Engineering

Automated Refactoring of Legacy Java Software to Default Methods

Raffi Khatchadourian
City University of New York
raffi.khatchadourian@hunter.cuny.edu

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

***Abstract*—Java 8 default methods, which allow interfaces to contain (instance) method implementations, are useful for the skeletal implementation software design pattern. However, it is not easy to transform existing software to exploit default methods as it requires analyzing complex type hierarchies, resolving multiple implementation inheritance issues, reconciling differences between class and interface methods, and analyzing tie-breakers (dispatch precedence) with overriding class methods to preserve type-correctness and confirm semantics preservation. In this paper, we present an efficient, fully-automated, type constraint-based refactoring approach that assists developers in taking advantage of enhanced interfaces for their legacy Java software. The approach features an extensive rule set that covers various corner-cases where default methods cannot be used. To demonstrate applicability, we implemented our approach as an Eclipse plug-in and applied it to 19 real-world Java projects, as well as submitted pull requests to popular GitHub repositories. The indication is that it is useful in migrating skeletal implementation methods to interfaces as default methods, sheds light onto the pattern's usage, and provides insight to language designers on how this new construct applies to existing software.**

***Keywords*-refactoring; java; interfaces; default methods**

I. Introduction

With the introduction of *enhanced* interfaces in Java 8, developers can now write *default* (instance) methods, which include an implementation that implementers will inherit if they do not provide their own [1]. Although the original motivation was to enable developers to add new functionality to existing interfaces without breaking clients [2], default methods can be used [3] as a replacement of the *skeletal implementation* pattern [4, Item 18]. This pattern centers around creating an abstract skeletal implementation class, which implementers extend, that provides a partial interface implementation, making the interface easier to implement.

While there are many advantages in migrating legacy code using the skeletal implementation pattern to instead use default methods, e.g., foregoing the need for subclassing, having classes inherit behavior (but not state) from *multiple* interfaces [3], facilitating local reasoning [5], doing so may require significant manual effort, especially in large projects. Particularly, there are subtle language and semantic restrictions, e.g., interfaces cannot declare instance fields. It requires preserving type-correctness by analyzing complex type hierarchies, resolving issues arising from multiple (implementation) inheritance, reconciling differences between class and interface methods, and ensuring tie-breakers with overriding class methods, i.e., rules governing dispatch precedence between class and default methods with the same signature, do not alter semantics, all of which will be elaborated on later.

We propose an efficient, fully-automated, semantics-preserving refactoring approach that assists developers in taking advantage of enhanced interfaces. The approach, based on type constraints [6,7], works on large-scale projects with minimal intervention and features an extensive rule set, covering various corner-cases where default methods cannot be used. It identifies instances of the pattern and safely migrates class method implementations to interfaces as default methods.

The related PULL UP METHOD refactoring [7,8] safely moves methods from a subclass into a super class. Its goal is to solely reduce redundant code, whereas ours includes opening classes to inheritance, allowing classes to inherit multiple interface definitions, etc. Moreover, our approach deals with multiple inheritance, a more complicated type hierarchy involving interfaces, semantic differences due to class tie-breaking, and differences between class method headers and corresponding interface method declarations.

The refactoring approach is implemented as an open source Eclipse (http://eclipse.org) plug-in. The experimental evaluation used a set of 19 Java projects of varying size and domain with a total of $\sim$2.7 million lines of code. Additionally, we submitted pull requests (patches) of the refactoring results to popular GitHub (http://github.com) repositories. Our study indicates that (i) the analysis cost is practical, with average running time of 0.115 seconds per input method and 0.144 seconds per thousand lines of code, (ii) the skeletal implementation pattern is commonly used in legacy Java software, and (iii) the proposed approach is useful in refactoring method implementations into default methods despite language restrictions. It also provides insight to language designers on how this new language constructs applies to existing software.

This work makes the following specific contributions:

Approach design. We present a novel automated refactoring approach for migration to Java 8 enhanced interface default methods. The approach infers which methods can be safely migrated to default methods via an exhaustive formulation of refactoring preconditions. We present new type constraints involving default methods and other modern Java constructs, as well as a scheme for semantics preservation in the context of tie breaking rules with classes. Furthermore, we identify all code changes required to perform the migration, including

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.16

removal and replacement of skeletal implementation classes.

Implementation and experimental evaluation. The approach is implemented as an open source Eclipse plug-in to ensure real-world applicability. A study on 19 Java programs indicates that the proposed techniques are effective and practical, and a pull request study demonstrates that the results are well-grounded. These results advance the state of the art in automated tool support for the evolution of legacy Java code to modern Java technologies.

II. Motivating Example

In this section, we present an example that will be used throughout the paper to highlight the motivation of using default methods in legacy Java code, the refactoring challenges involved, and how our approach applies in particular situations.

Fig. 1 portrays a hypothetical collection type hierarchy snippet. Fig. 2 contains the corresponding UML class diagram. The hierarchy has been simplified for presentation, with only portions relevant to our refactoring shown, and illustrates specific issues that can arise with the refactoring. The original system (white space added for alignment) is pictured in Fig. 1(a), while Fig. 1(b) depicts the same system with several methods migrated to interfaces as default methods. Removed code is struck through, added code is underlined, and replaced code is both underlined and emphasized. Both systems are type-correct and semantically equivalent.

Several interfaces, including `Collection` and `List`, with the latter extending the former, meaning that any concrete class extending `List` must provide or inherit implementations of methods declared in both interfaces, are shown in Fig. 1(a). Note that, unlike classes, Java interfaces can extend multiple interfaces. Methods exist for determining a `Collection`'s `size()`, adding an element, whether it `isEmpty()`, its `capacity()`, and whether it is `atCapacity()`. Note that the **`abstract`** keyword is optional for interface methods.

Several methods also exist for `List`s, including setting a `List`'s size, removing its last element, adding an element (line 12), whose declaration overrides that of the super interface `Collection` (commonly done for documentation [10]), replacing an element at a specified position, printing it to a specified stream, and copying it. Several methods are denoted as so-called *optional* operations as, e.g., not all list types may support modification. In such cases, implementers may throw an exception when these methods are invoked.

`AbsList`, an abstract class providing a skeletal implementation of a sequential, variable length `List`, is declared on line 18. Its purpose is to assist (concrete) classes in implementing the interface by declaring appropriate instance fields (line 20) and basic method implementations for the more primitive operations. Since it is abstract, it is not required to implement all interface methods. For the optional `removeList()`, the provided implementation (line 28) simply throws an `UnsupportedOperationException`. This way, concrete implementers extending `AbsList` that support element removal can override it with a working implementation, while others need not override it. The provided implementation of `print()` sends the standard string representation of the `List` to the stream. `AbsList`s are also `Copiable` (line 17), the provided implementation of which returns **`null`** instead of throwing an exception in the case that copying is unsupported.

A `Queue` interface snippet (line 36) contains two methods (one default) for `offer`ing and `add`ing elements, respectively, with the former returning **`true`** if the operation was successful and **`false`** otherwise (e.g., if the queue is full) and the latter, written in terms of the former, throwing an exception.

`AbsQueue` (line 40) provides a skeletal implementation of `Queue`[1] by extending `AbsList`, similar to an adapter pattern [11]. Unlike `AbsList`, it supplies functioning implementations for `removeLast()` (useful for a pop operation, not shown) and `add()`, while also customizing the `print()` method. Another extension of `AbsList`, `AbsUnmodList`, which does not support the `add` operation, is declared on line 51. `AbsStack` (line 55) specializes `AbsContainer`, and both it and `AbsSet` (line 58) implement `Collection`, providing `isEmpty()` implementations. Types implementing the `Comparator` interface (line 61) can order objects; `DefaultComparator` (line 63), of which `CComparator` (line 66) extends, supplies a basic ordering using hashes. Lastly, a main method (line 68) declares several concrete subclasses of various skeletal implementations of classes via anonymous inner classes (line 70).

Fig. 1(a) illustrates the skeletal implementation pattern, several drawbacks for which include:

Inheritance. Due to single-class inheritance restrictions, there is no clean way for `AbsQueue` to simultaneously benefit from `AbsList` *and* subclass another class. Moreover, `AbsQueue` could not easily take advantage of skeletal implementations split over multiple abstract classes [12].[2]

Modularity. There is no syntactic path between `List` and `AbsList`, i.e., no syntax exists in `List` that refers to `AbsList`. In general, a whole program analysis may be required to find suitable skeletal implementers for interfaces as they may be split across different files and packages [5].

Bloated libraries. `AbsList` implements many of `List`'s methods. This extra class can further complicate large libraries and make maintenance difficult. Additionally, method declarations are needed in both interfaces and classes to represent a *single* method and its default implementation.

Java 8 Default Methods. Default methods enable skeletal implementations in interfaces, thereby foregoing separate classes. Moreover, interface implementers need not search for separate skeletal implementations classes. Lastly, implementers can extend other classes, as well as inherit behaviors (but not state) from *multiple* interfaces, which can reduce the need for code duplication and forwarding methods [3].

Fig. 1(b) shows a refactored version of the running example, in which several skeletal implementations in classes have been migrated to interfaces as default methods. In the ab-

[1] `offer()` uses a **`double`** to demonstrate issues related to **`strictfp`**.

[2] Implementers already extending a class can use the pattern via delegation to an internal class [4] at the expense of auxiliary forwarding code.

```
interface Collection<E> {
  int size();
  void add(E elem); // add to this collection (optional).
  boolean isEmpty();
  int capacity();
  abstract boolean atCapacity();}

interface List<E> extends Collection<E> {
  void setSize(int i) throws Exception;
  void removeLast(); // optional operation.

  void add(E elem); // append to this list (optional).

  void set(int i, @NamedArg(value="elementToSet") E e);
  void print(PrintStream stream);
  List<E> copy();}
interface Copiable<E> {E copy();}
abstract class AbsList<E> implements List<E>,
    Copiable<List<E>> {
  Object[] elems; int size; // instance fields.
  @Override public int size() {return this.size;}
  @Override public void setSize(int i) {this.size = i;}
  @Override public boolean isEmpty(){return this.size()==0;}
  @Override public int capacity(){return this.elems.length;}
  @Override public boolean atCapacity()
    {return this.size() == this.capacity();}
  @Override public void removeLast()
    {throw new UnsupportedOperationException();}
  @Override public void set(int i, @NamedArg(value="el")E e)
    {this.elems[i] = e;}
  @Override public void print(PrintStream out)
    {out.println(this);}
  @Override public AbsList<E> copy() {
    try {return (AbsList<E>) this.clone();}
    catch (Exception e) {return null;}}}
@FunctionalInterface strictfp interface Queue<E> {
  boolean offer(E elem);
  default void add(E elem)
    {if (!offer(elem)) throw new RuntimeException("full");}}
abstract class AbsQueue<E> extends AbsList<E> implements
    Queue<E> {
  @Override public void removeLast()
    {if (!isEmpty()) this.setSize(this.size()-1);}
  @Override public void add(E elem) { // resize if necessary.
    this.set(this.size(),elem);this.setSize(this.size()+1);}
  @Override public void print(PrintStream out)
    {super.print(out); out.println("Printing queue ...");}
  @Override @ManagedOperation public boolean offer(E elem) {
    if (size() + 1.0 < capacity()) {add(elem); return true;}
    else return false;}}
abstract class AbsUnmodList<E> extends AbsList<E> {
  @Override public void add(E elem)
    {throw new UnsupportedOperationException();}}
abstract class AbsContainer {/* ... */}
abstract class AbsStack<E> extends AbsContainer 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;}}
interface Comparator<T> {int compare(T o1, T o2);}

abstract class DefaultComparator<T> implements Comparator<T>{
  @Override public int compare(T o1, T o2) {
    return Objects.hashCode(o1)-Objects.hashCode(o2);}}
class CComparator<T> extends DefaultComparator<T>{/*...*/}
class Main {
  public static void main(String[] args) {
    AbsQueue<Integer> queue1 = new AbsQueue<Integer>() {};
    queue1.removeLast();
    AbsList<Integer> queue2 = queue1.copy();
    Queue<String> queue3 = (s) -> true;
    assert(new AbsUnmodList<String>() {}.isEmpty());
    AbsStack stack = //...
    Collection col = stack;
    AbsContainer container = stack;}}
```

(a) Using abstract skeletal implementation classes to ease interface implementation.

```
interface Collection<E> {
  int size();
  void add(E elem); // add to this collection (optional).
  default boolean isEmpty() {return this.size()==0;}
  int capacity();
  abstract default boolean atCapacity()
    {return this.size() == this.capacity();}}
interface List<E> extends Collection<E> {
  void setSize(int i) throws Exception;
  default void removeLast() // optional operation.
    {throw new UnsupportedOperationException();}
  default void add(E elem) // append to this list (optional).
    {throw new UnsupportedOperationException();}
  void set(int i, @NamedArg(value="elementToSet") E e);
  default void print(PrintStream out) {out.println(this);}
  List<E> copy();}
interface Copiable<E> {E copy();}
abstract class AbsList<E> implements List<E>,
    Copiable<List<E>> {
  Object[] elems; int size; // instance fields.
  @Override public int size() {return this.size;}
  @Override public void setSize(int i) {this.size = i;}
  @Override public boolean isEmpty(){return this.size()==0;}
  @Override public int capacity(){return this.elems.length;}
  @Override public boolean atCapacity()
    {return this.size() == this.capacity();}
  @Override public void removeLast()
    {throw new UnsupportedOperationException();}
  @Override public void set(int i, @NamedArg(value="el")E e)
    {this.elems[i] = e;}
  @Override public void print(PrintStream out)
    {out.println(this);}
  @Override public AbsList<E> copy() {
    try {return (AbsList<E>) this.clone();}
    catch (Exception e) {return null;}}}
@FunctionalInterface strictfp interface Queue<E> {
  boolean offer(E elem);
  default void add(E elem)
    {if (!offer(elem)) throw new RuntimeException("full");}}
abstract class AbsQueue<E> extends AbsList<E> implements
    Queue<E> {
  @Override public void removeLast()
    {if (!isEmpty()) this.setSize(this.size()-1);}
  @Override public void add(E elem) { // resize if necessary.
    this.set(this.size(),elem);this.setSize(this.size()+1);}
  @Override public void print(PrintStream out)
    {super.print(out); out.println("Printing queue ...");}
  @Override @ManagedOperation public boolean offer(E elem) {
    if (size() + 1.0 < capacity()) {add(elem); return true;}
    else return false;}}
abstract class AbsUnmodList<E> extends AbsList<E> {
  @Override public void add(E elem)
    {throw new UnsupportedOperationException();}}
abstract class AbsContainer {/* ... */}
abstract class AbsStack<E> extends AbsContainer 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;}}
interface Comparator<T> {default int compare(T o1, T o2)
  {return Objects.hashCode(o1)-Objects.hashCode(o2);}}
abstract class DefaultComparator<T> implements Comparator<T>{
  @Override public int compare(T o1, T o2) {
    return Objects.hashCode(o1)-Objects.hashCode(o2);}}
class CComparator<T> implements DefaultComparator<T>{/*...*/}
class Main {
  public static void main(String[] args) {
    AbsQueue<Integer> queue1 = new AbsQueue<Integer>() {};
    queue1.removeLast();
    AbsList<Integer> queue2 = queue1.copy();
    Queue<String> queue3 = (s) -> true;
    assert(new AbsUnmodList<String>() {}.isEmpty());
    AbsStack stack = //...
    Collection col = stack;
    AbsContainer container = stack;}}
```

(b) Improvements after our refactoring is applied.

Fig. 1. A running example of a collection type hierarchy (inspired by [7,9]).

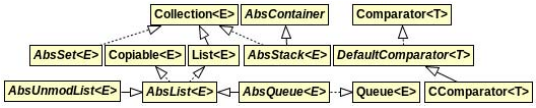

Fig. 2. Collection type hierarchy UML class diagram.

stract classes, the migrated methods were completely removed, including the method header and `@Override` annotations.[3] For each migrated method, the **`default`** keyword was inserted before the method return type and the delimiter was replaced with the body from the abstract class. In the case of `Collection.atCapacity()`, the **`abstract`** keyword on the method was removed (line 6) as the method is now concrete. For `List.print()`, the parameter name was altered to match that of the migrated method implementation.[4]

Now, developers considering implementing the interfaces can clearly recognize `List.removeLast()` as optional and what should happen when it is not implemented. Classes can inherit the default method `Collection.isEmpty()` without the need for finding and inheriting from a separate class or duplicating existing or writing new forwarding code.

`AbsUnmodList` and `DefaultComparator` were completely removed in Fig. 1(b). `CComprator` now *implements* `Comparator` rather than extending `DefaultComparator` (line 66) and inherits the default implementation of `compare()` from the interface, freeing it to extend other classes. Similar transformations are also required for classes extending `AbsUnmodList` but are slightly different depending on context. For example, the anonymous inner class (AIC) declared on line 73 now uses the interface rather than the class in its constructor call, making developer intent more explicit.

III. Problem Analysis

To highlight refactoring challenges, we examine Fig. 1(a) more carefully, revealing the following:

- In `AbsList`, methods `isEmpty()`, `atCapacity()`, `removeLast()`, `print()`, in `AbsUnmodList`, method `add()`, in `AbsStack`, method `isEmpty()`, and in `DefaultComparator`, method `compare()` can be migrated without affecting type-correctness and program semantics as classes now inherit them as default methods. The transformation also occurs unambiguously; each method has a well-defined, unique "destination" interface. Several methods are migrated to "indirect" interfaces, i.e., those not explicitly implemented by their declaring class but rather a super class up the type hierarchy.
- The *target* method `Collection.isEmpty()` has multiple *source* methods: `AbsList.isEmpty()`, `AbsStack.isEmpty()`, `AbsSet.isEmpty()`. Only the first two were migrated due to common implementations.[5]
- Methods `size()`, `setSize()`, `capacity()`, and `set()` in `AbsList` cannot be migrated because they access instance fields of the receiver, which cannot be migrated. Also, `offer()` in `AbsQueue` cannot be migrated because `size()` and `capacity()` are not declared in `Queue`. Nevertheless, exploring composite refactorings that may compensate in certain cases is an area for future work.
- Methods `add()` in `AbsUnmodList` and `copy()` in `AbsList` have multiple interface abstract target methods for which they provide implementations. Specifically, since `AbsUnmodList` extends `AbsList`, `AbsList` implements `List`, and `List` extends `Collection`, `AbsUnmodList` provides an `add()` implementation for methods in *both* interfaces. The migration must be to `List`, otherwise, migrating to `Collection` would result in a compilation error at line 73 because the (new) default implementation would be squelched by the corresponding abstract method in `List` further down the hierarchy. In the case of `copy()`, since `AbsList` implements *multiple* interfaces, the ambiguity occurs *across* the type hierarchy as both the `List` and `Copiable` interfaces declare a `copy()` method.
- Method `setSize()` in `AbsList` also cannot be migrated to `List` as `AbsList.setSize()` does not declare that an exception is thrown, while `List.setSize()` does (note that this is type-correct). Migrating it would result in a compile-time error at the call at line 43 because `removeLast()` does not deal with the declared thrown exception. Also, in the same class, `copy()` cannot be migrated to `List` as doing so would result in a compile-time error at line 71 because `queue1.copy()` would return `List`, which is not (a subtype of) `AbsList`. The same method could also not be migrated to `Copiable` because the returned value of `AbsList<E>` (line 34) is not type-compatible with the generic return type `E` from `Copiable`.
- `copy()` in `AbsList` also cannot be migrated because it calls `clone()`, which is a *protected* method of `Object`. While interfaces do not extend `Object`, they implicitly declare abstract methods for (only) each *public* `Object` method, which does not include `clone()` [13, Ch. 9.2]. Moreover, `print()` in `AbsQueue` cannot be migrated as it contains an unqualified reference to **`super`**, which is disallowed in interfaces to prevent the diamond problem [14,15].
- Note that `Queue` is functional (via the annotation at line 36), meaning that lambda expressions can be used to represent types implementing the interface. An interface may also be *effectively* functional, i.e., an annotation is not required to instantiate an interface using a lambda expression. It is required that functional interfaces have exactly one abstract method so that it is not ambiguous as to which method is invoked when the lambda expression is evaluated [13, Ch. 9.8]. As such, migrating `offer()` from `AbsQueue` to `Queue` would result in a type-incorrect program because `Queue` would no longer have any abstract methods, thus invalidating the lambda expression on line 72.
- Method `add()` in `AbsQueue` would be otherwise fine to migrate to `List`, however, doing so would produce a type-

[3]The `@Override` annotation is not carried over to the default method since the method body is no longer in an overriding relationship.

[4]Alternatively, each parameter reference could have been modified to match the interface parameter, however, changing only one location is less invasive.

[5]`AbsSet.isEmpty()` contains an unjustifiably different implementation to demonstrate issues that arise in complicated type hierarchies.

incorrect program as `AbsQueue` would "inherit" multiple interface methods, i.e., `add()` from `List` *and* `Queue`. Such a situation is disallowed by Java's typing rules.[6] Note that this situation would also arise even if `Queue.add()` were default. Furthermore, the same problem would emerge in `AbsQueue` had `add()` been defined in one of its subclasses.

- `Queue` is **`strictfp`** (line 36), meaning all calculations within its methods use strict floating-point math. While not related to type-correctness, migrating `offer()` from `AbsQueue` (line 48) to `Queue` could possibly alter semantics as it is not **`strictfp`**. The modifier can also be used at the method level, where a similar mismatch can occur. Additionally, there is an difference in annotation *types* between the two methods, possibly affecting processing frameworks like dependency injection. `set()` in `AbsList` (line 29) also has an annotation difference with the method in `List` (line 14), except here the difference is in the annotation *value*.
- Migrating method `removeLast()` in `AbsQueue` (line 42) to `List` does not result in any type-incorrect code, however, semantics would be altered as the run time target of the call at line 70 would change to the method in `AbsList`. This is because `AbsQueue` would inherit two different different versions of `removeLast()`, one from class `AbsList` and the other from interface `List`. In this situation, classes take precedence over interfaces. Moreover, we would have this problem had the conflicting method come from a directly implemented interface rather than from the class hierarchy.
- As a result of the refactoring, `AbsUnmodList` (line 51) is now empty, i.e., no methods, fields, or inner types remain. As such, it can be removed and the reference at line 73 can be replaced with its super class (see Fig. 1(b)).
- Although `AbsStack` (line 55) is now also empty, it cannot be removed because replacing the reference at line 74 with its super class would produce an error at line 75 as it does not implement `Collection`. Replacing it with the implemented interface would also cause an error at line 76 as `AbsContainer` is not (a super type of) `Collection`.
- `DefaultComparator` (line 63) is now empty as well and can be removed with the reference at line 66 being replaced with the implemented interface and changing **`extends`** to **`implements`**. However, we must ensure that subclasses like `CComparator` do not contain references to **`super`** that are not type-compatible with their new super class `Object`.

IV. Migration Approach

Assumptions. Our approach operates on a *closed-world* assumption that assumes full accessibility to all source code that could possibly affect or be affected by the refactoring. We also assume that method calls and references to fields and types can be statically resolved and that the original program successfully compiles under a Java 8 compiler. §V discusses how we relax several of these in our implementation.

[6]In this situation, the interface method to inherit must be *explicitly* selected by overriding the method, e.g., in `AbsQueue`, by calling one of them, e.g., `List.`**`super`**`.add()`. However, doing so would be in conflict with our goals as we would be delegating the class method to the default one.

Top-level Processing. Our input is concrete instance methods declared in abstract classes that implement at least one interface. §V discusses how using only abstract classes increases the likelihood that these methods would make suitable default methods; §VII considers pattern variations. Each method must have a body and not be **`final`** nor **`synchronized`**, as such methods are disallowed in interfaces [13].

Type Constraints. To determine whether migrating a method to an interface as a default method results in a type-correct program, we build upon an existing framework of type constraints [6,7]. For each program element, type constraints denote the subtyping relationships that must hold between corresponding expressions for that portion of the system to be considered well-typed. Thus, a complete program is type-correct if all constraints implied by all program elements hold.

Notation and Terminology: In line with [7], we will use the term *declaration element* to represent declarations of local variables, method parameters, fields, and method return types. We will use v to denote variables, M for methods, F for fields, C for classes, I for interfaces, Ex for exceptions, and T types (either a class, interface, or enum). We should note that the symbol M represents a method along with its signature, return type, and exception information, as well as a reference to its declaring type. Likewise, F and C denote a field and a type, respectively, along with its name, type in which it is declared, and its declaring type in the case of fields. Furthermore, we will use E to represent an expression or declaration element in a specific point in the program's abstract syntax tree. Type information regarding expressions and declaration elements are assumed to be available from the compiler.

Next, we define notions of virtual methods and root definitions, taken directly from [7] for expressing the type constraint for method overriding. A method M is virtual if M is not a constructor, M is not private and M is not static. Def. 1 defines the concept of *overriding*.[7]

Definition 1 (overriding)**.** *A virtual method M in type C overrides a virtual method M' in type B if M and M' have identical signatures and C is equal to B or C is a subtype of B.*[8] *In this case, we also say that M' is overridden by M.*

Def. 2 pertains to the concept of root definitions. For a method M, $\mathit{RootDefs}(M)$ is the set containing the most general methods in the type hierarchy that are overridden by M. Using our motivating example in Fig. 1, $\mathit{RootDefs}(\texttt{AbsQueue.add()}) = \{\texttt{Queue.add()}, \texttt{Collection.add()}\}$.

Definition 2 (root definitions)**.** *Let M be a method. Define:*

$$\mathit{RootDefs}(M) = \{M' \mid M \text{ overrides } M', \text{ and there exists no } M'' \ (M'' \neq M') \text{ such that } M' \text{ overrides } M''\}$$

Def. 3 presents a new notion of exception handling specific to our paper, particularly helpful in dealing with ex-

[7]This definition expresses that a virtual method overrides itself.

[8]We ignore access rights as interface methods are always public, whose visibility cannot be reduced. Throws clauses are a topic for future work.

$[E]$	the type of expression or declaration element E.
$[M]$	the declared return type of method M.
$Decl(M)$	the type that contains method M.
$Param(M, i)$	the i-th formal parameter of method M.
$T' \leq T$	T' is equal to T, or T' is a subtype of T.
$T' < T$	T' is a proper subtype of T, i.e., $T' \leq T \wedge T \not\leq T'$.
$super(C)$	the super class of class C.
$Class(T)$	iff type T is a class.
$Interface(T)$	iff type T is an interface.
$Default(M)$	iff method M is a **`default`** method.
$Public(M)$	iff method M is a **`public`** method.
$Abstract(M)$	iff method M has no body.

Fig. 3. Type constraint notation (inspired by [7]).

ception throws clause differences between source and target methods. For a given method call expression $E.m(\ldots)$, $Handle(E.m(\ldots))$ is the set of checked exceptions either thrown (declared by the enclosing method) or caught (by a surrounding try-catch block) at the expression. It is computed by intersecting the checked exceptions declared as being thrown by the compile-time target of $E.m(\ldots)$ with those appearing in corresponding catch blocks surrounding the call.

Definition 3 (handled exceptions)**.** *Let $E.m(\ldots)$ be a method call expression. Define:*

$$Handle(E.m(\ldots)) = \{Ex_h \mid Ex_h \text{ is a checked exception either thrown or caught at } E.m(\ldots)\}$$

Another concept specific to our paper is that of a functional interface, captured by Def. 4. It is useful in preventing existing lambda expressions from being invalidated.

Definition 4 (functional)**.** *An interface I is functional iff I is annotated with `@FunctionalInterface` or if there exists a lambda expression implementing I, in which case I has exactly one abstract method (effectively functional).*

Fig. 3 portrays the notation, including several helper predicates, we will use to describe type constraints. Let α be a *constraint variable*, which can be a type constant T, $[E]$, i.e., the type of an expression or declaration element E, $Decl(M)$, i.e., the type declaring method M, or $Decl(F)$, the type declaring field F. Then, a *type constraint* can be one of: (i) $\alpha_i \triangleq \alpha_j$, i.e., α_i is defined to be the same as α_j, (ii) $\alpha_i \leq \alpha_j$, i.e., α_i must be equal to or a subtype of α_j, (iii) $\alpha_i = \alpha_j$, i.e., $\alpha_i \leq \alpha_j \wedge \alpha_j \leq \alpha_i$, and (iv) $\alpha_i < \alpha_j$, i.e., $\alpha_i \leq \alpha_j \wedge \alpha_j \not\leq \alpha_i$. Note that if T' is a class and T is an interface, $T' < T$ means that T' *implements* T. If both T' and T are interfaces, then $T' < T$ means that T' *extends* T.

Inferring Type Constraints: Fig. 4 shows several rules for generating type constraints for many Java constructs. Constraints (5), (12)–(14), (16)–(18), (25), and (26) are new. The remaining are from [7]. Of these, due to space limitations, we only show those that are relevant here. Constraints (21)–(24) define the types of declaration elements, while constraints (25) and (26) are for default method migration.

As an example, for a virtual method call $E.m(E_1, \ldots, E_n)$ that statically resolves to a method M, by rule (2), the type of the method call expression is that of M's return type. It is also required by rule (3) that each method argument E_i be a subtype or equal to each method parameter $Param(M, i)$. Furthermore, the type of the receiver expression E must either declare or inherit the called method, expressed by rule (4) using Def. 2. For each of the thrown exceptions M declares, there must exist an exception handled at the call such that the thrown exception is a subtype or equal to the handled exception, expressed by our new rule (5) and Def. 3.

For convenience, let us combine rules 9 and 10 into a single "nearly" overriding relation $NOverrides(M', M) \equiv [Param(M', i)] = [Param(M, i)] \wedge [M'] \leq [M]$, which is a looser version of the constraints for method overriding, not requiring any relationship between the declaring types of M and M'. Rule (13) expresses that interfaces, unlike classes and enums, do not *inherit from* `Object` (cf. rule (12)), although they are *subtypes of* `Object` [16]. They do, however, have public methods ($Public(M)$) from `Object` ($Decl(M) \triangleq$ `java.lang.Object`) available to them ($Decl(M') \triangleq I \wedge NOverrides(M', M)$), i.e., they can be called whether or not they are explicitly declared by the interface [13, Ch. 9.2].

Rule (14) ensures that functional interfaces (see Def. 4) declare exactly one abstract method. For types other than interfaces, the type of **`super`** in method M is defined to be that of the super class of M (rule (16)). Otherwise, an unqualified **`super`** reference is undefined. For interface qualified **`super`** references where the declaring type of M implements the referenced interface, the type is defined to be that interface (rule (17)). Rule (18) defines the type of anonymous inner class (AIC) creation expressions.

Rule (25) is enforced for each method declared in an interface. For each method M declared in interface I, the rule applies if there is an unrelated interface J, another method M' declared either in J or inherited by J such that both M' and M have similar method signatures ($NOverrides(M', M)$) and at least one of them is a default method. In these cases, for all classes C such that C implements both I and J, there must exist a public class method M'' disjoint from M' and M such that C either declares or inherits M'', and M'' either implements or overrides at least one of the corresponding methods in I and J. If rule (25) is not satisfied, the underlying program will not be type-correct as C and its subtypes will "inherit" at least two different versions of the same interface method. Note that this problem arises even if not all the inherited methods are default. An example application of this and the following rule is provided in the next subsection.

Rule (26) ensures that any *concrete* type T implementing an interface I that declares a method M defines or inherits an implementation for M. The clause $T \leq Decl(M') \wedge NOverrides(M', M)$ is equivalent to the constraints for method overriding. It states that there must exist such a method M' with a body (i.e., $\neg Abstract(M')$) and, moreover, that there are no methods M'' *between* the two (i.e., $T < Decl(M'') < Decl(M')$) in the type hierarchy that is abstract, thus squelching the inherited implementation.

Checking Migration Preconditions. We will apply the type constraints to determine if migrating a class method to its

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 $Ex_{t1}, \ldots, Ex_{tj}$)	$[E.m(E_1, \ldots, E_n)] \triangleq [M]$ $[E_i] \leq [Param(M, i)]$ $[E] \leq Decl(M_1) \vee \cdots \vee [E] \leq Decl(M_k)$ where $RootDefs(M) = \{M_1, \ldots, M_k\}$ $\forall Ex_t \in \{Ex_{t1}, \ldots, Ex_{tj}\}$ $\exists Ex_h \in Handle(E.m(E_1, \ldots, E_n))[[Ex_t] \leq [Ex_h]]$	(2) (3) (4) (5)
access $E.f$ to field F	$[E.f] \triangleq [F]$ $[E] \leq Decl(F)$	(6) (7)
return E in method M	$[E] \leq [M]$	(8)
M' overrides M, $M' \neq M$	$[Param(M', i)] = [Param(M, i)]$ $[M'] \leq [M]$ $Decl(M') < Decl(M)$	(9) (10) (11)
for every class (and enum) C	$C \leq$ `java.lang.Object`	(12)
for every interface I	$I \not\leq$ `java.lang.Object` $\wedge \forall M[Decl(M) \triangleq$ `java.lang.Object` $\wedge$ $Public(M) \implies \exists M'[Decl(M') \triangleq I \wedge NOverrides(M', M)]]$	(13)
for every functional interface I	$\exists M[Decl(M) \triangleq I \wedge Abstract(M)$ $\wedge \forall M'[Decl(M') \triangleq I \wedge M' \neq M \implies \neg Abstract(M')]]$	(14)
implicit declaration of **`this`** in method M	$[$**`this`**$] \triangleq Decl(M)$	(15)
implicit declaration of **`super`** in method M	$\neg Interface(Decl(M)) \implies [$**`super`**$] \triangleq super(Decl(M))$	(16)
implicit declaration of `I.`**`super`** in method M	$Decl(M) < I \implies [$`I.`**`super`**$] \triangleq I$	(17)
expression **`new`** $T(E_1, \ldots, E_n)$ `...`	[**`new`** $T(E_1, \ldots, E_n)$ `...`] $\leq [T]$	(18)
declaration of method M (declared in type T)	$Decl(M) \triangleq T$	(19)
declaration of field F (declared in type T)	$Decl(F) \triangleq T$	(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	$\exists J, M'[Interface(J) \wedge J \not\leq I \wedge I \not< J \wedge J \leq Decl(M')$ $\wedge NOverrides(M', M) \wedge (Default(M') \vee Default(M))]$ $\implies \forall C \mid Class(C) \wedge C < I \wedge C < J[\exists M''[M'' \neq M' \wedge M'' \neq M$ $\wedge Class(Decl(M'')) \wedge C \leq Decl(M'') \wedge Public(M'')$ $\wedge NOverrides(M'', M')]]$	(25)
declaration of concrete type T implementing interface I declaring method M	$\exists M'[T \leq Decl(M') \wedge NOverrides(M, M')$ $\wedge \neg Abstract(M') \wedge \forall M''[T < Decl(M'') < Decl(M') \wedge$ $NOverrides(M'', M') \implies \neg Abstract(M'')]]$	(26)

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

implementing method in an interface as a default method affects type-correctness using Fig. 1(a):

- Migrating `AbsList.size()` to the method it implements, i.e., `Collection.size()` as a default method implies that [**`this`**] = `Collection`, violating constraint (7) that [**`this`**] $\leq$ [`AbsList`]. Migrating `AbsQueue.offer()` to `Queue` implies that `Queue` $\leq$ `Collection`, violating constraint (4) that [**`this`**] $\leq Decl($`Collection.capacity()`$)$.
- Migrating `AbsUnmodList.add()` to `Collection` (`Col`) implies that `AbsUnmodList<..>(){}` $\leq Decl($`Col.add()`$) \wedge NOverrides($`AbsUnmodList<..>(){}`, `Col.add()`$) \wedge \neg Abstract($`Col.add()`$)$ violating constraint (26) because `AbsUnmodList<..>(){}` $< Decl($`List.add()`$) < Decl($`Col.add()`$) \wedge NOverrides($`List.add()`, `Col.add()`$) \wedge Abstract($`List.add()`$)$. Migrating to `List`, however, does not violate the type constraint. The ambiguity arising from identifying the destination interface for `AbsList.copy()` is not as easily resolvable. We discuss handling ambiguity *across* the type hierarchy further below.
- Migrating `AbsList.setSize()` to `List` implies that the method now throws an `Exception`, violating constraint (5) as $Handle($**`this`**`.setSize(`**`this`**`.size()-1)`$) = \varnothing$. Migrating `AbsList.copy()` to `Copiable<E>` implies that [`(AbsList<E>)`**`this`**`.clone()`] = `AbsList<E>`, which violates constraint (8) that [`AbsList<E>`] $\leq$ [`E`].
- Migrating `AbsList.copy()` to `List` implies that [**`this`**] = [`List`], violating constraint (4) that [**`this`**] $\leq Decl($`Object.clone()`$)$ as `List` $\not\leq$ `Object` (cf. rule (13)). By rule (13), `clone()` is not available to interfaces as it is protected. By migrating `AbsQueue.print()` to `List`, the type of the unqualified **`super`** reference would be undefined by rule (16).
- Migrating `AbsQueue.offer()` to `Queue` implies that `Queue` has no abstract methods, violating constraint (14).
- Migrating `AbsQueue.add()` to either `List` or `Collection` implies that `AbsQueue` now inherits multiple `add()` interface methods, one of which contains a body, and does not override the method, violating constraint (25).

Note that the constraint would also have been violated had `add()` been defined in an `AbsQueue` subclass.

- While not related to type-correctness, migrating `AbsQueue.offer()` to `Queue` may alter the semantics of floating-point calculations. To solve this, we implicitly propagate all type-level **`strictfp`** modifiers to all constituent methods and require that the source method modifier match that of the target. A similar technique is employed for annotations but if an annotation value is present, as is the case with migrating `AbsList.set()` to `List`, the values, which are always statically available, must also match. The `@Overrides` annotation is excluded as it is removed. Type constraints are not used for these cases because they are not related to type-correctness but rather semantics-preservation. Since type constraints express relationships over only a single program versions, they are not typically used to preserve semantics [7].
- Migrating `AbsQueue.removeLast()` to `List` also does not raise any type-correctness issues but does alter semantics. To prevent this, we disallow from migration methods that override methods (using Def. 1) in *both* classes and interfaces. In this way, the migrated method, that would now reside in an interface rather than a class, will never lose in a tie to a class method with a similar signature.
- Replacing `AbsStack` with its super class `AbsContainer` on line 74 implies $[\texttt{stack}] \triangleq \texttt{AbsContainer}$ by rule (21), which violates constraint (1) that $[\texttt{AbsContainer}] \leq [\texttt{Collection}]$, generated from line 75. Replacing `AbsStack` with its implemented interface `Collection` instead also violates the constraint that $\texttt{Collection} \leq \texttt{AbsContainer}$, generated from line 76. Cases where the class to remove implements multiple interfaces is discussed further below.
- By removing the explicit super class of `CComparator`, the type of **`super`** in any of its methods changes from `DefaultComparator` to `Object` per rule (16). Any uses of **`super`** in `CComparator` to reference elements of `DefaultComparator` would violate type constraints.

Ambiguous Destination Interfaces. When the destination interface is ambiguous, if the ambiguity lies up the type hierarchy, we select the "closest" method to the source method up the hierarchy, which is akin to selecting a method not violating constraint (26). When the ambiguity is across the hierarchy, more input is needed. Currently, such methods fail preconditions to facilitate full automation, however, in our experiments, this accounted for only 0.87% of failures. Nevertheless, §VII explores dealing with such ambiguity.

Ambiguous Substitutable Interfaces. When classes become empty, we determine the type, which can be specific to each reference, to use in replacing references. Where the class has no (explicit) super class, one of the implemented interfaces is used. If multiple valid interfaces, i.e., those that do not violate any type constraints when substituted, are available at a particular reference expression, choosing any of them suffices.

Target Methods with Multiple Source Methods. As seen with `Collection.isEmpty()`, there may not be a one-to-one correspondence between source and target methods. In these cases, choosing any of the source methods passing preconditions to migrate would be safe as the non-migrated methods would override the new default method, as such, we migrate the largest number of equivalent source methods, as seen with `AbstractList.isEmpty()` and `AbsStack.isEmpty()`. For this, we categorize viable source method bodies into equivalence sets, which are deemed equivalent by performing an AST differencing algorithm [17] after fully qualifying any constituent elements. Methods in the largest set are chosen for migration, while the others fail preconditions. Although exploring other techniques is a topic of future work, this case accounted for 0.31% of failures.

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

TABLE I
EXPERIMENTAL RESULTS. JAVA8 IS THE `JAVA.` PACKAGE OF THE JDK 8.

V. EVALUATION

Implementation. Our approach is implemented as an open source plug-in to the Eclipse IDE, chosen for its existing refactoring framework [18] and that it is completely open source for all Java development. Eclipse ASTs with source symbol bindings were used as an intermediate representation. Our implementation is completely separate from the type constraints generated by the Eclipse Java Developer Tools.

To increase real-world applicability, we relaxed the closed-world assumption (§IV). For example, if an input method's destination interface is outside of the considered source code, it is conservatively labeled as non-migratable. Several options for reducing client impact exist, e.g., empty skeletal implementation classes can be either removed or deprecated. Furthermore, if such classes extend a super class not implementing all of the implemented interfaces, regardless of client code, the class is not removed. We additionally require no mismatches involving exception throws clauses and return types between source and target methods. An option to not consider non-standard (outside `java.lang`) annotation differences is available.

Experimental Evaluation. 19 open-source Java applications and libraries of varying size and domain (Table I) were used to

#	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 II
PRECONDITION FAILURES.

evaluate our approach's effectiveness. Column **KL** is the number of non-blank, non-comment thousands of lines of code, ranging from ∼2K for `rdp4j` to ∼600K for `Azureus`. The analysis was executed 5 times on an Intel Xeon E5 machine with 8 cores and 15GB RAM and a 8GB maximum heap size. Running time is shown in column **tm (s)**, averaging ∼0.144 secs/KLOC, which is practical even for large applications.

Default Method Migration: Column **KM** is the number of *all* methods in thousands. These are separated into two categories; the first is (filtered) methods that *definitely* cannot be refactored or be participating in the targeted design pattern. This includes methods *not* meeting the following criteria: (i) ones whose source is available, writable, and not generated, (ii) non-**`native`**, concrete instance methods (excluding constructors) declared in abstract classes implementing at least one interface whose source is also available, writable, and not generated, (iii) neither **`synchronized`** nor **`final`**, and (iv) overriding at least one non-default interface method. The remaining are *candidates* (column **cnds**). That these methods are declared in interface-implementing abstract classes is a strong indication that they are participating in the targeted pattern, i.e., they are meant to be subclassed and override an interface method. This increases the likelihood that these methods are general enough to be suitable default methods.

Our approach was able to automatically migrate 19.63%, accounting for 652 methods across 19 projects, of the methods that could possibly be participating in the targeted skeletal implementation pattern (column **dflts** for *defaults*) in spite of its conservative nature. Since we migrate methods having only a single target, it is unlikely that a more suitable default implementation exists, with the only possible caveat being methods that are eliminated in smaller equivalence sets, but this accounted for only 0.31% of failure cases (explained next).

The tool was unable to refactor the remaining 80.37% of candidate methods. Column **fps** portrays the total number of failed preconditions, averaging ∼2.32 per unmigratable method. Table II categorizes failures by kind, many of which correspond to the situations illustrated in §III, others are more fine-grained. For instance, P3 and P6 occurs when there is either a method- or type-level mismatch in the number of type parameters or when they are not assignment compatible between source and target methods, respectively. P5 occurs for methods residing in non-selected equivalence sets, while P7 corresponds to the situation where there are ambiguous destination interfaces. P9 and P10 coincide with violations of type constraints (25) and (26), respectively, whereas P11 is a violated type constraint involving **`this`** in the source method body. P8, P18, and P19 arise when the source method body accesses (e.g., **`private`**) elements not visible from or inherited by the destination interface, and P15 preserves program semantics by circumventing tie breakers to classes. Failures of type P17 involve accessible fields violating constraint (7).

Skeletal Implementation Class Removal: Column δ depicts the number of skeletal implementation classes that contained methods that were migrated. Of those, column **-**δ shows the number that can be safely removed, constituting 18.07%. The remaining 81.93% were not removable largely because the class was not completely empty after the refactoring, accounting for 66.02%, 18.93% had a super class not implementing a destination interface, and 15.05% had no super class but subclasses had incompatible **`super`** references. The total number of references to removable classes (not shown) averaged ∼9 per removed class and was as high as 106.

Discussion: Precondition failures are not unequivocally due to the analysis capabilities or conservative nature of our approach; portions are solely due to language constraints and/or limitations, and reporting these may give significant insight to language designers. P2, e.g., could be fixed in Java by having interfaces extend `Object`. To alleviate errors arising from P10, designers could choose to not allow the squelching of default methods up the hierarchy by abstract methods down the hierarchy. In other words, the ability of subclasses to "hide" default methods could be removed; a similar consideration is discussed in [3, §3.6]. Insight is also given into how well this new language construct integrates into existing code, thus answering some of the questions raised by [3, §10].

Conversely, several failures may be alleviated by either further analysis or transformation. P8, P19, and P18, e.g., may sometimes be mitigated by prompting developers to increase element visibility. For P18, the inaccessible method could be declared in the destination interface if each implementer already provides an implementation. P17 may sometimes be reduced by a composite refactoring that first encapsulates the field and then copies the setter/getter declarations to the destination interface. Note, however, this would not have fixed the field access failures related to `size()` and `setSize()` in Fig. 1(a) as they already encapsulate `size`.

While each of these are interesting areas of future work, they may require manual input from the developer or result in more invasive source code changes. In favor of a conservative, automated solution, we chose not to implement them at this time so that developers can take advantage of default methods on a mass scale with minimal code changes. P10 could be fixed

by creating a delegate method in the source method's declaring type that calls the migrated default method. However, doing so does not reduce bloat, one of the motivations of this work.

As discussed previously, 18.07% of classes containing migrated methods could be completely removed. Thus, 81.93% of such classes were only partially migrated to interfaces. Despite the small percentage of completely migrated classes, there are many benefits to partial migration, including forgoing the need to extend a skeletal implementation class when the interface default methods suffice and less maintenance between the interface and skeletal class for the migrated methods (e.g., modifying duplicate method header annotations) [5].

Pull Request Study. To assess our approach's usability, we also submitted pull requests to popular open source Java projects on GitHub. We ensured that projects compiled correctly and had identical unit test results and compiler warnings before and after the refactoring. In some instances, minor manual intervention was required, e.g., to merge javadoc. Due to space limitations, study details can be found on our website http://cuny.is/interefact. Of the issued 19 pull requests, 4 have been successfully merged, 5 are open, and 10 were closed without merging. Ranging in size and domain, the merged requests are to projects and frameworks from organizations such as Eclipse, AOL, and NHL (National Hockey League), and include the popular Eclipse (formally Goldman Sachs) Collections framework. In all, the merged projects total 163 watches, 1071 stars, and 180 forks. Several other projects, although enthusiastic about the approach, rejected our pull requests, citing reasons such as that they had not yet moved to Java 8 or needed to support older Java clients at the time. We expect demand for Java 8 features to increase, especially since they are now (at least partially) supported by Android [19].

As the code's organization is changing, our refactoring may have engineering and design implications. While improving design is a motivating factor, enhancing modularity and reducing library bloat are others (see §II). The acceptance of our refactoring results, totaling the addition of 349 and the removal of 647 LOC, to large and popular frameworks during this study increases confidence that the resulting design is acceptable.

Verifiability: Our plug-in is available at http://git.io/v2nX0 and subject source is available on our aforementioned website.

VI. Related Work

Goetz [10] also formalizes (using [20], an early version of) default methods. Constraints (25) and (26) have similar purposes as the type checking rules there, however, [10] does not deal with exceptions, functional interface invalidation, method body compatibility, and class removal. Type constraints are also well-established in many refactoring approaches.

A constraint-based approach to refactoring is also utilized in [21]. There, constraints are generated to assist refactorings in avoiding access control violations. [22] also develops constraints for averting such violations but for use in a broader transformation framework to simplify refactorings. These problems are related yet orthogonal to ours, however, these approaches could be used to augment our approach. [23] tackles the *foresight* problem typically found in constraint-based refactorings where new constraints are needed as a result of applying the refactoring. However, since our approach moves (method) *bodies* rather than declarations, no new constraints should be necessary as a result of the move.

Similar to the Pull Up Method refactoring [7], the "move original method to superclass" law [24] uses rules expressing when such a transformation is semantically equivalent. The declaration constraints there, however, do not apply here since no method declarations are being moved but rather bodies. Our approach fundamentally builds upon the other constraints, and we could have expanded this law to account for multiple inheritance issues and tie breakers with classes.

Our preliminary work [5] solely explored the feasibility of improving modularity via default methods. [25] inquires about using default methods for trait-oriented programming or mixins. [26] also analyzes differences between arguments and parameters names. Many refactorings use type constraints for type checking [7], type inference [27–29], and semantics preservation [30]. [7,31,32] deal with reorganizing type hierarchies, [33] and [34] refactor Java programs to use lambda expressions and enumerated types, respectively. [35] demacrofies C++11 programs. [36] refactors to use cloud-based services, and [37] automatically migrates CSS to preprocessors.

VII. Conclusion & Future Work

We have presented an efficient, fully-automated, type constraint-based, semantics-preserving approach, featuring an exhaustive rule set, that migrates the skeletal implementation pattern in legacy Java code to 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. Moreover, 4 pull requests were merged into GitHub repositories, which include large, widely used frameworks from reputable organizations.

In the future, we will explore pattern variations, e.g., allowing input methods from concrete classes, which requires analyzing instantiations and determining suitable default methods from concrete classes. Compensating for source methods directly accessing fields or methods outside destination interfaces will also be investigated, as well as javadoc merging, transforming build scripts for migrations across modules, other heuristics for method equivalence set elimination, machine learning to disambiguate destination interfaces, improving efficiency [38], and alternate testing techniques [39–42].

Acknowledgment

Many thanks to Olivia Moore and Md Arefin for their assistance with experiments. This material is based upon work supported by PSC-CUNY under award #69165-00 47 and the Tokyo Institute of Technology Research Abroad program.

References

[1] Oracle Corporation, "Java Programming Language Enhancements." [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html

[2] ——, "Default methods," 2016. [Online]. Available: http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

[3] B. Goetz, "Interface evolution via virtual extensions methods," Oracle Corporation, Tech. Rep., Jun. 2011. [Online]. Available: http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf

[4] J. Bloch, *Effective Java*. Prentice Hall, 2008.

[5] R. Khatchadourian, O. Moore, and H. Masuhara, "Towards improving interface modularity in legacy java software through automated refactoring," in *Companion Proceedings of the 15th International Conference on Modularity*, ser. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 104–106.

[6] J. Palsberg and M. I. Schwartzbach, *Object-oriented type systems*. John Wiley and Sons Ltd., 1994.

[7] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 3, pp. 9:1–9:47, May 2011.

[8] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] Oracle Corporation, "Java™ platform, standard edition 8 api," 2016. [Online]. Available: http://docs.oracle.com/javase/8/docs/api

[10] B. Goetz and R. Field, "Featherweight defenders: A formal model for virtual extension methods in java," Oracle Corporation, Tech. Rep., 2012. [Online]. Available: http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley, 1995.

[12] C. S. Horstmann, *Java SE 8 for the Really Impatient*. Addison-Wesley Professional, 2014.

[13] J. Gosling, B. Joy, G. L. S. Jr., G. Bracha, and A. Buckley, *The Java Language Specification*, 8th ed. Addison-Wesley Professional, 2014.

[14] G. B. Singh, "Single versus multiple inheritance in object oriented programming," *SIGPLAN OOPS Mess.*, vol. 6, no. 1, pp. 30–39, Jan. 1995.

[15] M. Sakkinen, "Disciplined inheritance," in *European Conference on Object-Oriented Programming*, 1989.

[16] A. Lundblad, "Java: Do interfaces inherit from object?" 2016. [Online]. Available: http://programming.guide/java/do-interfaces-inherit-from-object.html

[17] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[18] D. Bäumer, E. Gamma, and A. Kiezun, "Integrating refactoring support into a Java development tool," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[19] Google Inc., "Use java 8 language features | android developers." [Online]. Available: http://developer.android.com/guide/platform/j8-jack.html

[20] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight java: A minimal core calculus for java and gj," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, May 2001.

[21] F. Steimann and A. Thies, "From public to private to absent: Refactoring java programs under constrained accessibility," in *European Conference on Object-Oriented Programming*, ser. ECOOP'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 419–443.

[22] M. Schafer, A. Thies, F. Steimann, and F. Tip, "A comprehensive approach to naming and accessibility in refactoring java programs," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1233–1257, Nov. 2012.

[23] F. Steimann and J. von Pilgrim, "Constraint-based refactoring with foresight," in *European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 535–559.

[24] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic reasoning for object-oriented programming," *Science of Computer Programming*, vol. 52, no. 1-3, pp. 53–100, Aug. 2004.

[25] V. Bono, E. Mensa, and M. Naddeo, "Trait-oriented programming in java 8," in *Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14. New York, NY, USA: ACM, 2014, pp. 181–186.

[26] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1063–1073.

[27] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for parameterizing java classes," in *International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 437–446.

[28] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring java applications to use generic libraries," in *European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 71–96.

[29] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 265–279.

[30] B. De Sutter, F. Tip, and J. Dolby, "Customization of

java library classes using type constraints and profile information," in *European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg, 2004, pp. 584–608.

[31] I. Moore, "Automatic inheritance hierarchy restructuring and method refactoring," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: ACM, 1996, pp. 235–250.

[32] Z. Alshara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi, "Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation," in *ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 55–64.

[33] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 543–553.

[34] R. Khatchadourian, "Automated refactoring of legacy java software to enumerated types," *Automated Software Engineering*, pp. 1–31, 2016.

[35] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating c++ programs through demacrofication," in *International Conference on Software Maintenance*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 98–107.

[36] Y.-W. Kwon and E. Tilevich, "Cloud refactoring: Automated transitioning to cloud-based services," *Automated Software Engineering*, vol. 21, no. 3, pp. 345–372, Sep. 2014.

[37] D. Mazinanian and N. Tsantalis, "Migrating cascading style sheets to preprocessors by introducing mixins," in *International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 672–683.

[38] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1145–1156.

[39] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1095–1105.

[40] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, Feb. 2013.

[41] M. Mongiovi, "Safira: A tool for evaluating behavior preservation," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 213–214.

[42] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194.