Fall 10-30-2017

Defaultification Refactoring: A Tool for Automatically Converting Java Methods to Default

Raffi T. Khatchadourian
CUNY Hunter College

Hidehiko Masuhara
Tokyo Institute of Technology

Defaultification Refactoring: A Tool for Automatically Converting Java Methods to Default

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

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

***Abstract*—Enabling interfaces to declare (instance) method implementations, Java 8 default methods can be used as a substitute for the ubiquitous skeletal implementation software design pattern. Performing this transformation on legacy software manually, though, may be non-trivial. The refactoring 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. All of this is necessary to preserve type-correctness and confirm semantics preservation. We demonstrate an automated refactoring tool called MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE for transforming legacy Java code to use the new default construct. The tool, implemented as an Eclipse plug-in, is driven by an efficient, fully-automated, type constraint-based refactoring approach. It features an extensive rule set covering various corner-cases where default methods cannot be used. The resulting code is semantically equivalent to the original, more succinct, easier to comprehend, less complex, and exhibits increased modularity. A demonstration can be found at http://youtu.be/YZHIy0yePh8.**

***Index Terms*—refactoring; java; interfaces; default methods; type constraints, eclipse**

I. INTRODUCTION

Java 8 *enhanced* interfaces enable developers to write *default* (instance) methods that include an implementation that implementers will inherit if one is not provided [1]. Although originally intended to facilitate the addition of new functionality to existing interfaces without breaking clients [2], default methods can also be used [3] to substitute the *skeletal implementation* pattern [4, Item 18], which is ubiquitous in many software projects [5]. The pattern involves creating an abstract skeletal implementation class that implementers can extend. This class provides a partial interface implementation and thus results in an interface that is easier to implement.

Advantages in migrating legacy code from using the skeletal implementation pattern to default methods include foregoing the need for subclassing, having classes inherit behavior (but not state) from *multiple* interfaces [3], and facilitating local reasoning [6]. Although advantageous, such a migration requires significant manual effort, particularly in large projects, as there are subtle language and semantic restrictions that must be considered. One such restriction is that interfaces cannot declare instance fields. The migration 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, preserve semantics.

We demonstrate an automated refactoring tool named MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE for transforming legacy Java code to use default methods. The tool assists developers in taking advantage of enhanced interfaces in an efficient, fully-automated, and semantics-preserving fashion. The approach is based on type constraints [7,8] and works on large-scale projects with minimal intervention. Featuring an extensive rule set that covers diverse corner-cases where default methods are prohibited, the approach identifies instances of the skeletal implementation pattern and safely migrates methods to corresponding interfaces as default methods.

The related PULL UP METHOD refactoring tool [8,9] manipulates a type hierarchy by safely moving methods from a subclass up into a super class so that all subclasses may inherit from it. This refactoring is fundamentally different from migrating method definitions from skeletal implementations to interfaces as default methods in terms of its goals and the targeted design pattern. Namely, its sole goal is to reduce redundant code, whereas ours includes opening classes to inheritance, allowing classes to inherit multiple interface definitions, etc. Moreover, while the two refactorings share some preconditions, i.e., conditions that must be met to guarantee refactoring correctness, in terms of type constraints violations, our approach deals with multiple inheritance, a more complicated type hierarchy involving interfaces since classes may implement multiple interfaces while extending a class, semantic differences due to class tie-breaking, further constraints on interfaces as they cannot declare fields, and differences between class method headers and corresponding interface method declarations. Lastly, while methods to be pulled up typically are declared in a common class, in our case, default methods may be migrated from multiple classes into a single interface, pressing the need for a more widespread, batch processing approach across classes and packages.

Our refactoring tool (available at http://git.io/v2nX0) is implemented as an open source Eclipse (http://eclipse.org) plug-in built atop of the Java Development Tools (JDT) (http://eclipse.org/jdt) refactoring infrastructure [10]. Our tool can process projects in batch, mining for occurrences of the skeletal implementation pattern than can be converted to default methods. A refactoring preview pane is provided, along

with detailed information of code that fails preconditions.

For the tool evaluation, an extensive refactoring test suite was created, featuring 259 refactoring regression tests, triggered via continuous integration. Each tests verifies that (i) both the input and output code versions compile successfully and (ii) the actual refactored version matches that of the expected refactored version given the initial version of the input source code. The usefulness of the tool was assess via the analysis of 19 Java projects of varying size and domain with a total of ∼2.7 million lines of code. Additionally, pull requests (patches) of the refactoring results were submitted to popular GitHub (http://github.com) repositories as a preliminary study.

The details of the underlying approach, as well as thorough experimental results, can be found in our previous work [5]. Beyond [5], we make the following specific contributions:

Implementation details A thorough treatment of the novel aspects of the tool implementation is presented in detail. This includes the tool's architecture, API usage, data representations, algorithms, and implementation issues and limitations. Furthermore, the tool's relationship to the PULL UP MEMBER refactoring implementation is thoroughly explored.

User perspective A broad overview of how our tool is used to perform large-scale refactorings is given. This includes screenshots of the tool's usage and a video demonstration. Our hope is to receive valuable feedback on the improvement of the user interface, as well as promote its usage.

II. ENVISIONED USERS

The users we envision our tool attracting are especially those who are tasked with maintaining and/or improving legacy (currently) Java systems. Our tool is most advantageous in situations where legacy systems are using Java 8 and are actively maintained. In this way, using our tool on these systems would result in code that is more succinct and easier to maintain, e.g., skeletal implementation classes may be eliminated, as will be discussed in the following sections.

Since our tool takes advantage of the built-in, user-friendly Eclipse refactoring infrastructure, developers with even little refactoring experience may use our tool. Users may be those that are tasked with refactoring an entire project or writing new code in only portions of a large system. Since our tool will identify possible instances of the skeletal implementation pattern and that the resulting program will be semantically equivalent to the original, users do not necessarily have to possess a thorough knowledge of the pattern.

III. SOFTWARE ENGINEERING CHALLENGES

In this section, we discuss the Software Engineering challenges our tool is made to address. Fig. 1 portrays a screenshot of the refactoring preview pane that a user is presented with prior to executing the refactoring on a simplified example. In order for such a pane to be displayed, the user (developer) selects Java elements of the Eclipse IDE, e.g., the package explorer. MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE supports a wide range of element granularity, from (multiple) methods up to (multiple) projects within an Eclipse workspace. For example, if the user context-clicks (right-clicks) a Java class in the Eclipse UI and selects our refactoring option, the tool will traverse the entire class for instance method definitions (implementations) that implement an interface method in an interface explicitly specified as being implemented by either the enclosing class or one of its parents.

Background and Motivation. Fig. 1 consists of two panes, namely, the "Original Source" and the "Refactored Source," with the former being in input to the tool and the latter the proposed output. Users may select "Finish" if they agree with the refactoring, and may revise the input parameters (e.g., files) by unchecking them from the top section. In the each pane, there are two types, namely, interface `I` and an abstract class `C`. On the left, `I` declares a simple, single abstract method `m()` (line 4). Class `C` implements `I` and thus provides a basic implementation of `m()`. Since it is not meant to be directly instantiated but rather to be used as a *skeletal* implementation of `I`, `C`i is declared as abstract. Instead of implementing `I` directly, prospective implementers can subclass `C` and thereby inherit "default" implementations of some or all of the interface methods. Often times, such skeletal implementations provide "default" method implementations that clients can inherit from rather than providing their own if the implementation is applicable to them. Other times, such abstract skeletal implementers provide complete implementations comprised of more primitive interface methods that subclasses override.

Although useful, there are several drawbacks to the skeletal implementation pattern, especially w.r.t. inheritance, modularity, and bloat [5]. Classes extending `C`, for example, to benefit from the provided skeletal implementations will not be able to extend other classes. This could be problematic in situations where classes implement *multiple* interfaces with each one have its own corresponding skeletal implementer. Moreover, there is no syntactic path between an interface and a skeletal implementer; clients looking to take advantage of a skeletal implementation must rely on either a global project analysis and/or documentation. Lastly, skeletal implementers require an additional, separate type, which could make already highly complicated libraries more complicated.

Many of the aforementioned problems can be solved with **`default`** methods that are part of the *enhanced interface* feature of Java 8 [11, Ch. 9]. The right pane of Fig. 1 portrays the refactored version of the left with `m()` removed from class `C` and its body appended in the formerly abstract method `m()` in interface `I`. Furthermore, in `I`, `m()` is now prefixed with the **`default`** keyword. After the refactoring, class `C` is now empty; whether it can be completely removed is explored below.

Now, implementers of `I` can simultaneously benefit from the default implementation of `m()` and extend a different class. Implementers also do not need to discover skeletal implementers of `I` as default implementations are coupled with the interface method declaration. Lastly, a new type is not needed to represent the default method.

Although Fig. 1 portrays a scenario where the refactoring succeeds, cases exist where executing the refactoring would produce either type-incorrect or semantically-inequivalent re-

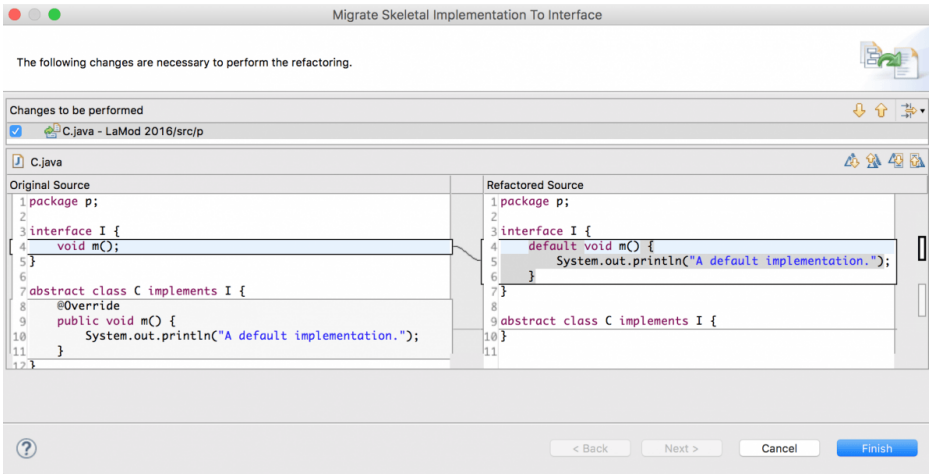

Fig. 1: Screenshot of the refactoring preview pane for the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring.

sults. For example, consider the following snippet:

```
interface I {void m();}
interface J {default void m() {/* ... */}}
abstract class C implements I,J {void m() {/*...*/}}
```

Here, migrating method `C.m()` to interface `I` as a default method would cause a compilation error due to class `C` now inheriting ambiguous method definitions of method `m()`.

Default methods have many advantageous over the skeletal implementation pattern, however, there are some potential trade-offs. For example, placing implementations directly in interfaces can violate some of the fundamental benefits of interfaces acting as abstract data types (ADTs), where implementation details should not be included. Particular to default methods in Java, there has been some reported performance degradation in certain cases when using default methods [12]. However, this has been seen as a temporary JDK/JRE problem that affects only a small number of cases [13].

Analysis Challenges. Although Fig. 1 is a simple example, there are many other situations where determining whether it is safe to convert a method to default may not be obvious:

- A particular skeletal implementer may provide a single skeletal implementation for *multiple* interfaces, complicating the processes of determining the *target* interface of where the *source* class method shall be migrated to as a default method. Our current implementation rejects input methods with ambiguous target interfaces.
- A given interface may have *multiple* skeletal implementers; which of these should be migrated to the interface as a default method? Our current implementation performs equivalence set merging to find the largest set of equivalent source methods for migration and fails the others.
- A skeletal implementer may declare instance fields that are used in the source method. Since instance fields cannot be declared interfaces, is it possible to convert such methods to default? Our current implementation does not allow methods directly accessing fields to be refactored, but if the developer is willing to make accessors and mutators part of the interface, they are free to perform the ENCAPSULATE FIELD and EXTRACT INTERFACE refactorings prior to ours.
- Lambda expressions require interfaces with a *single* abstract method. Converting a method to default may invalidate this requirement. Our approaches rejects methods that are part of functional interfaces used in lambda expressions.
- In cases where a class inherits the same method from *both* a class and an interface where the interface method is default, the interface method will "lose a tie" to the class. As such, we must ensure that the dispatch semantics remain intact after the refactoring. Otherwise, calls to the source method will not dispatch to the target in the refactored version but rather a method in a different class. Our current implementation rejects methods in this situation.

Other issues include those related to empty skeletal implementers, i.e., cases where all source methods have been migrated to their targets. After the refactoring, further analysis is required to safely remove them. For example, the class may not be able to be removed if it is instantiated somewhere in the code base. Other references of the class could arise w.r.t. inheritance. Due to the pattern structure, it is likely that the skeletal implementer is being extended. Naturally, we would replace the *extension* of subclasses with the *implementation* of the destination interface, i.e., the interface for which the target method was migrated. However, the subclass classes may have its own subclasses with references to `super` that used to refer to the super class of the (now empty) skeletal implementer. If the replacement is performed, those references would now refer to the *interface* rather than the super class.

Implementation Challenges. While [5] handles many of the above issues, implementation-specific challenges include:

Architecture What is the best way to organize such a refactoring tool? As a research prototype, how can the tool simultaneously help real developers while being easily evaluated?

Reuse Given that the refactoring problem shares similarity with the PULL UP METHOD refactoring, how can we best leverage existing code from that plug-in here?

Applicability Do the various assumptions made in the approach scale to real-world software refactoring?

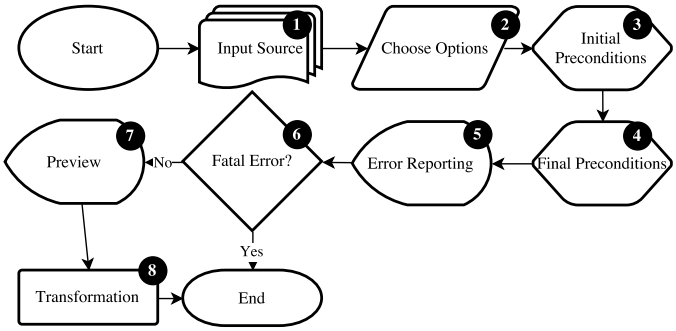

Fig. 2: High-level system workflow.

Validation Given the significant number of "corner-cases" involved in this refactoring, how can we ensure to developers that the resulting refactoring code will be type-correct and semantically equivalent to the original?

Usability Challenges. There are also challenges specific to developer adoption. A refactoring that makes such large scale and broad changes may not be immediately appealing to developers of mature and highly utilized projects due to risk. As such, our tool needs to address this by incorporating UI features to reduce risk in certain cases. These features were essential in the tool's evaluation via a pull request study [5]. Such challenges are addressed in the next section.

IV. Implementation

The MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring is implemented as an open source plug-in for the Eclipse IDE (available at http://git.io/v2nX0). Eclipse has been chosen for its existing, well-documented, and well-integrated refactoring framework [10]. Our tool has been built atop of this framework and utilizes many of its features, including source code analysis and transformation APIs (e.g., `ASTRewrite`), refactoring preview pane (as shown in Fig. 1), precondition checking `Refactoring.checkFinalPreconditions()`), and refactoring testing. `ITypeHierarchy`, which facilitates efficient traversals of a type hierarchy in Eclipse, is used extensively in checking refactoring preconditions. Furthermore, Eclipse is completely open source for all Java development thus possibly impacting more Java developers. Eclipse ASTs with source symbol bindings are used as an intermediate representation.

Workflow. Fig. 2 depicts the high-level workflow for our plug-in. The input to our tool is source code at various levels of granularity (step 1). At the smallest level, the developer may select a single method, or set thereof, for migration. At the highest level, the plug-in works on (multiple) projects. In this case, the tool will search through each project for method implementations to be used as input to the refactoring.

Next, the developer is presented with options regarding the invasiveness of the refactoring (step 2, discussed later). The tool proceeds to perform simple checks on the input methods, e.g., whether the input method is contained in a writable file, in step 3. Traditionally, these basic checks can be repeated, with the developer possibly selecting different elements to be used as input to the refactoring. However, our tool slightly breaks from this traditional behavior by simply filtering out method definitions that fail the checks using a non-fatal error. In this way, developers are free to execute the refactoring again with any modifications they have made to their project(s).

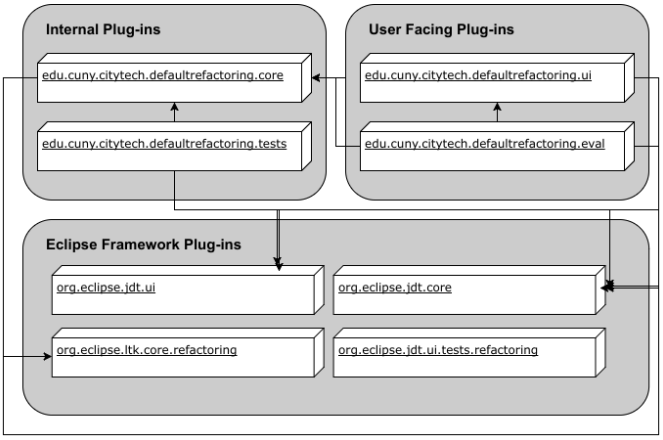

Fig. 3: Architecture and plug-in dependency diagram.

The bulk of the computationally intensive precondition checking occurs in step 4. Both fatal and non-fatal errors are reported to the user in step 5. An example of a fatal error is that no input methods have passed preconditions, as such, no refactoring can take place. Conversely, an example of a non-fatal error is where migrating a method would alter program semantics and at least other one method has passed. If no fatal errors are present, the tool provides the developer with a preview of the proposed changes (step 7) and performs the transformation (step 8) upon the developer's confirmation. Our tool supports undo capabilities in the case the developer changes their mind after executing the transformation.

Architecture and Dependencies. Fig. 3 portrays the overall plug-in architecture and dependency overview of our refactoring tool. It consists of four plugins that are organized into two categories, i.e., internal plug-ins (those not directly interacting with the developer) and user-facing plug-ins (those directly interacting with the developer and utilizing a UI). As shown in Fig. 3, the *core* and *test* plug-ins are not accessible outside of the development environment, while *UI* and *eval* are invoked via the Eclipse interface. Splitting the UI into two plug-ins, i.e., the UI plug-in for normal usage and eval plug-in for evaluating the research aspect of the tool, is, we believe, a novel concept in source code analysis and transformation research prototype design. In this way, the deployment artifact of the plug-in does not include the evaluation portion (i.e., buttons to generate experimental data files) of the tool, nevertheless, both the evaluation and user plug-ins are available to the researchers. Doing so makes the plug-in particularly useful for *both* tasks. On the other hand, the *test* plug-in is not part of the deployable artifact. The foundational Eclipse plug-ins and their dependencies are also depicted in Fig. 3.

Relationship to Other Refactoring Plug-ins. The plug-in menu options are coupled with other reorganization refac-

toring support in Eclipse. Although our approach is type constraint-based (see [5]), similar to the current Eclipse PULL UP METHOD refactoring, our implementation is completely separate from the type constraints generated by the JDT. In other words, type constraints serve as a conceptual basis.

Some of our implementation leverages and adapts code from the current Eclipse PULL UP METHOD refactoring, especially those related to determining if a code entity is accessible from another type. However, several important changes were necessary to achieve the integration. For example, the current PULL UP METHOD refactoring is not well-suited for batch processing as its pulled up members must originate from the same declaring type. Since Java 8 default methods are a new feature that developers may want to adopt to entire projects, we felt the need to modify the existing code to process on the project-level for this particular refactoring. As such, our plug-in accepts both fine-grained (e.g., individual methods) and course-grain (e.g., multiple projects) inputs. Other modifications included considerations related to modern Java features such as generics and lambda expressions.

Technical Details. Although a thorough treatment of technical details can be found in [5], here, we discuss several aspects of how our implementation realizes the concepts set-forth by type constraints. As mentioned earlier, we make extensive use of `ITypeHierarchy` to traverse the input methods' relationships to other types in the project. The goal is to check of particular type constraints would be violated as a result of the refactoring.

To demonstrate the imperative-style code that realizes the more declarative-style type constraints, consider the constraint for method invocation. This constraint applies to any program construct in the form of $E.m(E_1, \ldots, E_n)$ to a virtual method M where E is an expression. Such a construct may appear within an input method body being refactored to a default method. The constraint's purpose is to preserve type-correctness when the method invocation is moved to the interface. In short, we must ensure that there is a corresponding method in the type hierarchy of the destination interface when $E =$ **`this`** as the type of **`this`** will change after the refactoring.

One of the corresponding type constraints for method calls is $[E] \leq \mathit{Decl}(M_1) \vee \cdots \vee [E] \leq \mathit{Decl}(M_k)$ where $\mathit{RootDefs}(M) = \{M_1, \ldots, M_k\}$, meaning that the type of E must be a (sub)type of the type declaring one of the called method's root definitions. Informally, root definitions are the top-most types in a type hierarchy declaring a method. Preserving the validity of this constraint guarantees that there is a corresponding method in the destination interface.

In our implementation, we use `ITypeHierarchy` to check that the called method exists in the destination interface's "super type" hierarchy via a call to `IType.newSupertypeHierarchy()`. A super type hierarchy is one containing the type in question and all of its super types. The checking is achieved via the following code snippet; full source can be found in our open source repository:

```
mInHier = isInHier(calledMeth, destInterSuperHier);
boolean isMethInHier(IMethod m, ITypeHierarchy h) {
 return Stream.of(h.getAllTypes()).anyMatch(t -> {
  IMethod[] meths = t.findMethods(m);
  return meth != null && meths.length > 0;});}
```

Real-world Applicability. To increase real-world applicability, we relaxed a closed-world assumption utilized by our approach as detailed in [5]. A closed-world assumption is useful for the conceptual basis of automated refactoring approaches as it allows the algorithm designer to assume that all code that could ever be affected by the refactoring be present at the time of its execution. While useful in algorithm formulation, it is typical for (i) source code of libraries, frameworks, and remote (e.g., web-based) services not to be present as input to the tool, and (ii) clients depending (or will be depending in the future) on the refactored software to not even be associated with the input project let alone present.

Thus, to make our approach applicable to real-world scenarios, we relaxed the above described assumption on several fronts. For example, if an input method's destination interface is outside of the considered source code, it is conservatively labeled as non-migratable (i.e., failed precondition).

Usability and Managing Risk. We also offer the developer several options to the refactoring for reducing client impact exist. For instance, we allow the developer to choose whether empty skeletal implementation classes should be removed. For those not removed, we offer the option to deprecated so that clients can plan for their future removal. Furthermore, if such classes extend a super class not implementing all of the implemented interfaces, regardless of client code, the class is not removed and references not replaced with the super type. We additionally require no mismatches involving exception throws clauses and return types between source and target methods. Lastly, an option to not consider non-standard (outside `java.lang`) annotation differences is available, which may be useful in projects not using such processing frameworks.

Developers have several other options, including whether to include only abstract classes as input, which increases the likelihood classes that truly are skeletal implementers.

V. EVALUATION

Our tool successfully converted ∼20% of methods possibly participating in the skeletal implementation pattern to interfaces as default methods [5] in 19 real-world, open source projects of varying size and domain with a total of ∼2.7 million lines of code. Our approach is extremely conservative, and thus 20% is respectable considering that the approach is fully automated. Moreover, many of the changes made by the tool are widespread, and developers do not need to carefully analyze large and complex code bases to take advantage of default methods for their legacy code. Many of the precondition failures were related to inaccessibility of members between skeletal implementers and destination interfaces and access to instance fields.

The correctness of the refactoring approach was validated in several ways. First, we ensured that no compilation errors existed before and after the refactoring. Furthermore, we verified that all unit tests results were identical before and after the refactoring. A preliminary pull request study was

also performed to ensure that the musically produced results matched what experienced developers may have written. Four projects accepted our pull requests, and the tool's results were integrated into the projects. This indicates that it is useful.

To validate our implementation, our plug-in features an extensive refactoring test suite with over 200 refactoring tests. Such tests consist of "before" and "after" files. The "before" file is used as input to the tool and the output is matched against the "after" file. The significant number of test cases exercises many corner-cases that appear in the refactoring.

VI. Related Work

Current Eclipse refactorings do not have the capability to deal with default method conversions. In fact, when attempting to "pull up" a method from a class to an interface, Eclipse states that the method already exists in the destination interface. Moreover, as previously discussed, the PULL UP METHOD refactoring is not typically widely applicable; it normally applies to a single class. This is because issuing this refactoring on a broad scale has potentially disruptive and widespread results. In contrast, our refactoring has more of a subtle effect and can be issued throughout the project. Nevertheless, it may be possible to combine our refactoring with PULL UP METHOD depending on the target.

Another important difference between our tool and PULL UP METHOD is related to a "stubbing" behavior. For example, in PULL UP METHOD, if an instance method call in the source method exists, that method may be available in the target type. If it is not, an abstract method (stub) can be created in the target type to compensate. However, in our case, there may be no relationship between the called instance method and the destination interface. For example, if the called instance method is in the skeletal implementer, then, there is a relationship. However, the called method may be inherited from another class that does not implement the interface.

Other refactorings [8,14,15] reorganize type hierarchies, though not for default methods. [16] and [17] transform Java programs to use lambda expressions and enumerated types, respectively, while [18] demacrofies C++11 programs.

VII. Conclusion & Future Work

A refactoring tool that incorporates an efficient, fully-automated, type constraint-based, semantics-preserving approach that migrates the skeletal implementation pattern in legacy Java code to instead use default methods has been demonstrated. The tool is implemented as an Eclipse IDE plug-in and was evaluated using several techniques.

In the future, we plan to compensate for situations where source methods directly accessing fields or methods outside destination interfaces. Since interfaces cannot declare instance fields, fields may be encapsulated in the skeletal implementers and corresponding methods declared in the destination interface. A similar methodology could be employed for missing accessed methods, and missing static fields could be directly moved. However, all peer implementers must supply implementations. We also plan to improve refactoring speed [19].

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*, 2nd ed. Addison Wesley, 2008.

[5] R. Khatchadourian and H. Masuhara, "Automated refactoring of legacy Java software to default methods," in *ICSE*, 2017.

[6] R. Khatchadourian, O. Moore, and H. Masuhara, "Towards improving interface modularity in legacy java software through automated refactoring," in *International Conference on Modularity Companion*, 2016.

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

[8] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM TOPLAS*, 2011.

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

[10] D. Bäumer, E. Gamma, and A. Kiezun, "Integrating refactoring support into a Java development tool," in *OOPSLA*, 2001.

[11] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*. Addison-Wesley Professional, 2014.

[12] L. Rytz, "Performance of using default methods to compile scala trait methods," 2016. [Online]. Available: http://scala-lang.org/blog/2016/07/08/trait-method-performance.html

[13] G. Wilkins, "eclipse/jetty.project pull request #773," Webtide, 2016. [Online]. Available: https://git.io/v56qs

[14] I. Moore, "Automatic inheritance hierarchy restructuring and method refactoring," in *OOPSLA*, 1996.

[15] 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 *GPCE*, 2015.

[16] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *FSE*, 2013.

[17] R. Khatchadourian, "Automated refactoring of legacy java software to enumerated types," *ASE*, 2016.

[18] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating C++ programs through demacrofication," in *ICSM*, 2012.

[19] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *ICSE*, 2016.