Towards Improving Interface Modularity in Legacy Java Software through Automated Refactoring

Raffi Khatchadourian Olivia Moore
City University of New York, USA
rkhatchadourian@citytech.cuny.edu
olivia.moore@mail.citytech.cuny.edu

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Abstract

The skeletal implementation pattern is a software design pattern consisting of defining an abstract class that provides a partial interface implementation. However, since Java allows only single class inheritance, if implementers decide to extend a skeletal implementation, they will not be allowed to extend any other class. Also, discovering the skeletal implementation may require a global analysis. Java 8 enhanced interfaces alleviate these problems by allowing interfaces to contain (default) method implementations, which implementers inherit. Java classes are then free to extend a different class, and a separate abstract class is no longer needed; developers considering implementing an interface need only examine the interface itself. We argue that both these benefits improve software modularity, and discuss our ongoing work in developing an automated refactoring tool that would assist developers in taking advantage of the enhanced interface feature for their legacy Java software.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

Keywords refactoring; java; interfaces; default methods

1. Introduction

Java 8 is one of the largest upgrades to the popular language and framework in over a decade. There are several new, key features that can help make programs easier to read, write, and maintain, especially in regards to collections. The advantages of migrating legacy code to Java 8 include a reduction of code and files via enhanced interfaces [8].

An *interface* in Java is a type whose primary purpose is to lists method declarations (method headers only, no bodies) that classes can *implement*. Implementing an interface guarantees that the class provides implementations for *all* of the interface methods. If an implementing class does not provide an implementation for all methods, a compile-time error would result.

Listing 1 portrays a snippet of the `java.util.List` interface found in the Java Development Kit (JDK) [17] as an example. One of the method declarations, namely, `clear()`, along with its documentation, is listed. A concrete implementer of `List` must provide an implementation for the `clear()` method.

```
1  interface List { //...
2    /** Removes all of the elements from this list
   ↪   (optional operation) ...
3      @throws UnsupportedOperationException if the clear
   ↪   operation is not supported by this list. */
4    void clear(); /* ... */}
```

Listing 1: A snippet of the Java List interface.

Often times, interface implementers share common functionality, and some interface methods may be considered optional, i.e., implementers may decide to provide an implementation for them or simply state the operation is not supported by throwing an appropriate exception. As an example of the former, consider Listing 1, where the `clear()` method documentation states that an `UnsupportedOperationException` should be thrown by the implementation if the operation is not supported (line 3).

The *skeletal implementation* pattern has emerged to make interfaces easier to implement by providing a corresponding `abstract` class (i.e., a class that is solely used to capture common data and functionality and thus cannot be instantiated itself) as a partial interface implementation [2]. Implementers then extend the skeletal implementation, thereby inheriting the partial implementations. For each interface method, a default implementation may be provided that will execute if the implementer does not provide one. Or, functionality common to all method implementations may be supplied. Implementers can chose to inherit the complete method or furnish a customization via a super method call. Prominent examples of this pattern include the `java.util.AbstractCollection` and `java.util.AbstractList` classes, which are part of the JDK Collections library. Figure 1 depicts a UML class diagram of these classes, including their relationships and implemented interfaces. Note that `java.util.ArrayList` and `java.util.Vector` are two example concrete implementations of `java.util.List`.

Though useful, the skeletal implementation pattern has several significant drawbacks. Firstly, since Java allows only single class inheritance, if implementers decide to extend the skeletal implementation, they will not be allowed to extend any other class. Furthermore, if an implementer already extends another class, they would not be able to easily use the skeletal implementation.[1] Secondly, an additional type is created, which may further complicate the software and may not be easily locatable by developers whom may only be examining the interface they wish to implement.

[1] Implementers already extending a class can use the skeletal implementation pattern via delegation to an internal class [2].

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892681

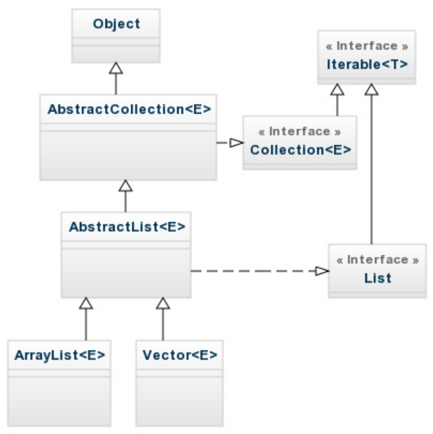

Figure 1. JDK collections UML class diagram snippet.

Java 8 default methods alleviate these problems by allowing interfaces to contain (default) method implementations along with their declarations [18].[2] Implementers inherit default methods, much like they inherit methods from an extended class. Implementers are not, however, constrained by single class inheritance, as is the case with the skeletal implementation pattern (Java allows multiple *interface* inheritance), and are thus free to extend other classes. Also, a separate class is not needed; developers considering implementing an interface need only examine the interface itself. This facilitates local reasoning as discovering skeletal implementations may require a global analysis (there is no syntactical path from the interface to the skeletal implementation).

For example, using `default` methods, the default implementation specification in the documentation of the `clear()` method in Listing 1, line 3 can be replaced with an actual code body:

```
default void clear() {throw new
 ↪  UnsupportedOperationException();}
```

In this case, implementers of `List` would not be required to provide their own implementation if the `clear()` method was not supported. Moreover, such a code body could appear in a skeletal implementation of the interface, e.g., `AbstractList`. If enough of these can be migrated to interfaces as `default` methods, it may be possible to eliminate the skeletal implementation. `List` implementers would then not need to extend a skeletal implementation and can implement `List` directly.

Our refactoring algorithm and corresponding automated tool will be of great use to developers looking to take advantage of the enhanced interface feature in Java 8. Refactoring [16] is a process in which code is restructured to improve software design and safety features while preserving the original program semantics. Refactoring tools typically automatically assist developers with a range of software evolution and maintenance tasks, including migrating source code to a new platform version [6,10,11,13,20,23] or to make use of more desirable paradigms for performance improvements [4], translating existing code to a new platform [3,12,24], and restructuring code to reflect a superior design philosophy [5,9,14].

While it is desirable, e.g., to improve modularity, to refactor code to take advantage of new features such as enhanced interfaces in Java 8, doing so may require significant manual effort by the developer that is tedious, time-consuming, and both error- and omission-prone [4]. Our tool must perform a semantics-preserving source-to-source transformation of existing Java programs in an efficient, error-free, and omission-free way. Moreover, the tool must be able to work on large-scale projects with minimal (if any) developer intervention. These requirements are, in fact, common to most refactoring approaches. In the following sections, we will detail how such requirements apply to the problem-at-hand.

[2] Java 8 interfaces cannot contain default fields nor constructors.

2. Approach

Our approach is being formulated via a careful study of the Java 8 language specification an will produce a sufficient set of refactoring preconditions. These are the conditions a program must meet prior to the refactoring to guarantee that the transformation is semantics-preserving. The algorithm will analyze the abstract syntax tree (AST) of the input program.[3] Transformation will occur via the use of existing tooling in the Eclipse Integrated Development Environment (IDE; `http://eclipse.org`). Eclipse has been chosen due to its ample refactoring tool plug-in creation documentation [1] and that it is completely open source for all Java development,[4] thus possibly impacting more Java developers. Moreover, we are adapting existing algorithms used for refactoring class hierarchies [21].

2.1 Research Questions

1. Is it possible to eliminate skeletal implementations from legacy Java code by migrating their methods to interfaces?
2. What are the refactoring preconditions? How applicable are they to real-world software?
3. What is the percentage of candidate skeletal implementations that can be completely migrated?
4. If all methods cannot be migrated to an interface (e.g., if a method accesses fields), either due to semantics-preservation (i.e., failed refactoring preconditions) or language constraints, is it worthwhile to migrate a subset of the methods in both skeletal implementations to interfaces?
5. Can the process be automated?
   (a) If so, can it be done accurately and efficiently? How does the automated result compare with a manual refactoring?
   (b) Can we minimize or completely eliminate developer intervention? That is, how much human input would be needed?
   (c) Is it possible automatically identify occurrences of the skeletal implementation pattern in legacy Java code?

2.2 Methodology

Considering Question 1 in Section 2.1, naturally, several issues arise, e.g., how do we determine if an interface implementer is "skeletal?" The pattern occurrence may not be completely obvious as the classic pattern can be slightly modified by the implementer. The perhaps obvious occurrence is an `abstract` class whose name begins with "Abstract," extends no classes, and implements a single interface. Clearly, implementers can still abide to the essence of the pattern without exhibiting *any* of these criteria. For example, an implementer may not be abstract but, nevertheless, used like one (i.e., never instantiated and/or mainly used for inheritance purposes). Moreover, a skeletal implementer may implement several interfaces (e.g., marker interfaces like `java.io.Serializable`). To complicate matters further, there may exist a *hierarchy* of skeletal implementers, e.g., `java.util.AbstractList` is a skeletal implementation of the `java.util.List` interface that extends `java.util.AbstractCollection`, which is a skeletal implementation of the `java.util.Collection` interface (cf. Figure 1).

[3] While many refactoring approaches analyze ASTs, it is possible to use other approaches, e.g., intermediate representation analysis [22].

[4] As of this writing, the IntelliJ IDE is closed source for Java EE development; see `http://jetbrains.com/idea/#chooseYourEdition`.

We are analyzing several large open source projects[5] for common occurrences and variations of the skeletal implementation pattern. We propose that our approach can later be expanded to deal with pattern variations. This preliminary study will help identify necessary refactoring preconditions (question 2 in Section 2.1).

There are several interesting preconditions that arise when considering type hierarchies and multiple interface implementation. For example, suppose that a skeletal implementation `A` defines a method `m()` and implements two interfaces `I` and `J`, each of which declare the same method `m()`. As such, `A.m()` is an implementation of both `I.m()` and `J.m()`. Further suppose that `I.m()` is a `default` method. In this case, `A.m()` overrides `I.m()`. If we choose to migrate `A.m()` to `J` as a `default` method, then any subclass of `A` inheriting `A.m()` will break because it must now choose which implementation, either `I.m()` or `J.m()`, it will inherit. We plan to further explore such complex relationships further.

Partial migration of skeletal implementations interfaces (Question 3 in Section 2.1) will be answered by adjusting and safely expanding refactoring preconditions while simultaneously analyzing large code bases. In our preliminary implementation (available at `http://git.io/v2nX0`), the preconditions are as strict as possible. We plan to carefully relax these to include more candidate classes. Upon each relaxation phase, we will employ techniques from the literature [7,15,19] to help test our refactoring algorithm.

For Question 4 in Section 2.1, we speculate that, at least, partial skeletal implementation migration will be worthwhile because *all* concrete implementations (i.e., classes extending the skeletal interface) may not need to extend the skeletal implementation as a result of the migration. This will make implementing the interface easier in some cases and possibly reduce the need for simultaneous evolution of interfaces and skeletal implementations.

We have uncovered several instances in the JDK 8 where existing methods were *manually* migrated to Java 8 enhanced interfaces. We plan to expand this investigation to more projects. Ideally, these projects will have existed for several years, are extended by a wide variety of client applications, and have a large developer-base. This last criterion helps prevent the collection of biased information as individual developers may use very specific styles and idioms. We plan to manually compare the differences between the versions and attempt to identify general transformations. Furthermore, any required client changes will be analyzed.

3. Conclusion & Future Work

We have presented our ongoing work in increasing Java interface modularity by automatically migrating method definitions in classes to Java 8 interfaces as default methods. Doing so makes Java interfaces easier to implement because software design patterns, such as the *Skeletal Implementation Pattern*, where a separate partial implementation class is provided, would not be necessary. This alleviates developers from a possible whole program analysis.

One point of future interest would be to find ways to compensate for limitations of Java 8 enhanced interfaces. For example, there is no direct support for `default` fields and constructors. As such, methods that access fields, for example, will not be able to be migrated to interfaces (failed precondition). This result may be mitigated by encapsulating fields and generating corresponding method declarations in the target interface.

There is also a possibility to migrate `static` methods to interfaces as part of the refactoring as such methods are now allowed in interfaces as of Java 8. We foresee exploring this avenue in the future. We also plan to evaluate our approach via a full empirical study on popular Java projects.

[5] See `http://git.io/v2ZDL` for a complete project listing.

Refactoring interfaces may be risky as third-party clients may rely of the interfaces. Like many refactorings, our preliminary algorithm works with a "closed-world" assumption. However, in our future work, we will explore ways to relax this assumption, possibly through more stringent preconditions in certain situations.

References

[1] D. Bäumer, E. Gamma, and A. Kiezun. Integrating refactoring support into a Java development tool. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

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

[3] A. De Lucia, G. Di Lucca, A. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *Int. Conf. Software Maintenance*, 1997.

[4] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Int. Conf. Software Engineering*, 2009.

[5] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[6] 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*, 2005.

[7] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Int. Conf. Software Engineering*, 2014.

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

[9] H. Kegel and F. Steimann. Systematically refactoring inheritance to delegation in Java. In *Int. Conf. Software Engineering*, 2008.

[10] R. Khatchadourian, J. Sawin, and A. Rountev. Automated refactoring of legacy Java software to enumerated types. In *Int. Conf. Software Maintenance*, 2007.

[11] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *Int. Conf. Software Engineering*, 2007.

[12] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: an experience report. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1998.

[13] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ programs through demacrofication. In *Int. Conf. Software Maintenance*, 2012.

[14] Y.-W. Kwon and E. Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, 2014.

[15] M. Mongiovi. Safira: A tool for evaluating behavior preservation. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[16] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] Oracle Corporation. Java^{TM} platform, standard edition 8 api, 2016. URL `http://docs.oracle.com/javase/8/docs/api`.

[18] Oracle Corporation. Default methods, 2016. URL `http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`.

[19] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 2013.

[20] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

[21] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[22] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[23] D. von Dincklage and A. Diwan. Converting Java classes to use generics. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[24] Y. Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. *Asia-Pacific Software Engineering Conference*, 2001.