Java with Traits — Improving Opportunities for Reuse

Philip J. Quitslund and Andrew P. Black

OGI School of Science & Engineering
Oregon Health and Science University
Portland, Oregon USA

ABSTRACT

The Java language includes features that present significant barriers to reuse; in practice, programmers have no choice but to copy and paste code that is not accessible via inheritance. Traits improve code-sharing in Smalltalk by providing a means to reuse such behavior, and we claim that a similar mechanism for Java would overcome not just the lack of multiple inheritance but Java's other barriers to reuse as well, including the use of private, final and synchronized qualifiers. In support of our claim we present the initial findings of a case study of Java Swing, a large production-quality library, showing how we isolated pieces of duplicated code that could not be eliminated by conventional means and how traits could be used to eliminate them.

Keywords: Reuse, Traits, Java, Code Duplication, Java Swing

1. JAVA'S BARRIERS TO REUSE

Four features of Java are responsible for significant code duplication in the Java Foundation Classes (JFC).

1. **Lack of Multiple Inheritance of Implementation.** With Java interfaces we can group classes in different hierarchies by the protocols they support, effectively enabling *multiple inheritance of type*. This allows clients to treat objects with a shared protocol uniformly, irrespective of their representation. While this promotes flexibility and a generic style of programming, it does nothing to address reuse: in many cases, classes that share protocols would also like to share aspects of their implementations. As a concrete example, take the PrintStream and Printwriter classes from the java.io package. Both support a uniform printing protocol (print(boolean), print(int), print(long), and so on) and *identical* implementations for twelve methods. While the shared protocol can be reflected by an interface, Java has no feature to abstract out and share the implementation because PrintStream and Printwriter both subclass classes in which printing is not appropriate (FilterOutputStream and Writer, respectively).

2. **Inaccessible Private Inner Classes.** Inner classes are a useful mechanism for grouping related classes and for codifying "friend" relationships between classes. Their use is especially common in GUI applications where they provide a convenient means for implementing call-backs and adapters. Unfortunately, inner classes are also very difficult to reuse because the conventional wisdom is to make them private to negate the security risk that they introduce*. An example of this phenomenon can be seen in the java.util.concurrent package where the FutureTask and ScheduledExecutor classes define identical, but non-reusable, private inner ListIterator classes.

Send correspondence to: philipq@cse.ogi.edu (Philip Quitslund) or black@cse.ogi.edu (Andrew Black).

*The JVM does not support inner classes directly. Instead they are compiled into separate classes that gain access to their containing class's fields and methods via compiler-generated accessor methods. Effectively this promotes methods and fields to public that might otherwise be private. Although the accessors are "hidden" behind mangled names, a malicious programmer could craft bytecode that violates their intended encapsulation policies.

3. **Non-Extensible Final Classes.** Making a class final ensures that it cannot be subtyped (it might also improve performance). Because Java equates subtyping and subclassing, final restrictions also block opportunities for reuse. A canonical example is Java's representation of strings in java.lang. To ensure the proper functioning of the interpreter and compiler, which depend on its concrete implementation, the class String is declared final to prevent programmers from substituting subtypes that break its semantic contract. To reuse String's implementation one is forced to copy and paste (as in java.lang.AbstractStringBuilder, where parts of String's indexing behavior are duplicated).

4. **Synchronized Variations.** In some cases, a basic concurrent version of a class can be obtained by adding the synchronized modifier to the critical methods. Such is the case with Vector (synchronized) and ArrayList (unsynchronized) in java.util which could share, with a little refactoring, at least fourteen method bodies if we could selectively introduce synchronization.

2. WHAT ARE TRAITS?

Traits[1] are a mechanism for code reuse that complements single inheritance. Traits, like classes, are containers for methods. But, unlike classes, traits have no fields. Traits, like abstract classes, cannot be instantiated directly; instead, they are composed into classes (which are instantiable). A class ColorPoint might be composed of traits TColor and TPoint and other bits (like state, for example), which means that ColorPoint will have the methods defined in TColor and TPoint. Because method names might conflict, composition can be selective, allowing for the removal and renaming of composed methods. Thus, if TColor and TPoint both define an equal method, we can exclude either implementation or alias one or both with another name. If the conflict is left unresolved, the composition includes neither trait method but instead includes a special stub method indicating an unresolved conflict. To use this trait, the programmer must explicitly disambiguate the conflict by exclusion or by defining an overriding method in the client class.

Commonly, traits refer to methods that they do not themselves define. Traits that do not define all the methods they call are said to *require* these methods. A *comparable* trait, for example, might define comparison methods (<=, >=, ~=, min, max, and so on) in terms of the < and = operations that it *requires* (see Figure 1). In order for a class to use this trait it must provide < and = methods. If it does not, it is considered incomplete (ostensibly abstract).

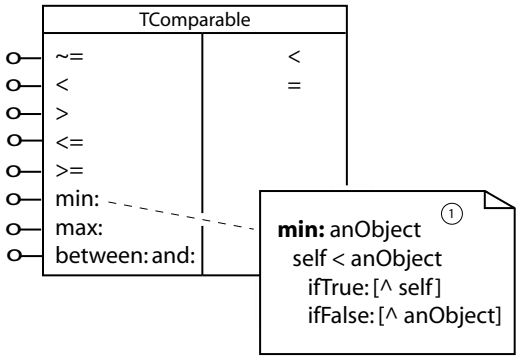

Figure 1. Trait TComparable *provides* ~=, <, >, and so on (the methods on the left) and *requires* < and = (on the right). Provided methods can be implemented in terms of required ones as in the definition of min: (①).

2.1. Extending Traits for Java

In seeking to adapt the ideas behind traits to the Java language, we must consider several issues that do not arise with Smalltalk. Most obviously, Java methods and constructors frequently refer to their enclosing class by name, so it is necessary to provide a mechanism (e.g., a thisclass keyword) that trait methods can use to abstract away from any specific named class. Beyond this, the need to declare types for local variables and parameters

in methods may make methods from traits less reusable than they would be if they could be typed generically. In addition, we must consider features like nested classes, visibility modifiers (public, private, protected) and other modifiers such as final, synchronized, and native. Which of these features merit first-class treatment?

We believe that a simple extension of the trait composition clause, not unlike the the aliasing mechanism of Smalltalk traits, will be sufficient to resolve the code duplication problems found in the JFC. In Smalltalk traits, an alias expression such as TEnumerable @(map→collect) denotes a trait like TEnumerable except that it also contains an additional method callen map whose implementation is the same as that defined for collect. A similar mechanism can be used to adjust the modifiers on trait methods when they are incorporated into another trait or class. Unlike the alias mechanism, which simply adds a new name for an existing method, a modification mechanism would need to hide the old version of the method, and introduce a new one with the modified property (see Figure 2 for a possible syntax). If clients could add declaration modifiers when using methods defined in traits, the difficulties introduced by final classes and synchronization modifiers could be sidestepped. Such a mechanism would allow the same methods to be reused in final and non-final and synchronized and un-synchronized settings. Inner classes, though a bit more complex, could be made shareable in a similar way.

```
class MySynchronizedVector uses TVector@{* as synchronized;} { // ... }
```

Figure 2. A synchronized Vector variation declared using a pattern-matching scheme like that employed in AspectJ's pointcut language.[2] The @ operator indicates an alias operation and the wildcard matches all of TVector's methods.

3. A CASE STUDY: CODE DUPLICATION IN JAVA SWING

The Java Foundation Class (JFC) libraries are flush with examples of code duplication that cannot be eliminated by single inheritance. To provide more than anecdotal support for the value of traits we sought to quantify just how much duplication there can be in production systems. To evaluate how traits might improve code-sharing in the wild, we looked at Java Swing, a library of cross-platform GUI components provided with the Java distribution. We focused on duplication that seems to result from the restricted power of single inheritance. We chose Swing because it is production quality and quite large—in the JDK 1.5.0,[3] Swing consists of 605 top-level classes/interfaces and over 290 thousand lines of (commented) code. We obtained a conservative estimate of code duplication in Swing by using the freely available CPD ("Copy Paste Detector") tool,[4,5] which employs the fast (but naïve) Karp-Rabin string-matching algorithm.[6] CPD detected over 15 thousand duplicated lines across 231 classes, accounting for 5 percent of the source and 38 percent of Swing's classes.

Surprisingly, some of this duplication might be eliminated using standard features of Java, without the need for traits. That is, if classes C_1 and C_2 contain duplicated methods and also share a direct superclass, then the duplicated methods could possibly be raised to the shared superclass or put in an intermediate shared abstract superclass. Similarly, if code is multiply defined in a class and its superclass, then the copy in the subclass can be eliminated.

Candidates for traits are those cases where the classes sharing the behavior do not share an immediate superclass. Here the feasibility of removing the duplication with inheritance depends on how far the classes containing the duplication are below their lowest shared superclass—a metric we will call *inheritance depth*. We define the inheritance depth to be the sum of the distances between two compared classes and their shared superclass. Figure 3 shows three scenarios: if code is duplicated in a class and its superclass, then we say the depth is one (case a), if it is in classes that share an immediate superclass, then we say the depth is two (case b), and if one of two sibling classes is separated from the shared superclass by another class, then we say the depth is three (case c). The smaller the inheritance depth, the easier it is to remove the duplicated method. The duplicated paint method is trivial to remove in (a), straightforward in (b) but tricky in (c). The danger in putting paint in D_3 is that it may not be appropriate there or in C_5 which inherits it — here code is shared at the expense of understandability. This phenomenon has been described as putting behavior "too high" [7] and is a prime candidate for refactoring to use traits.

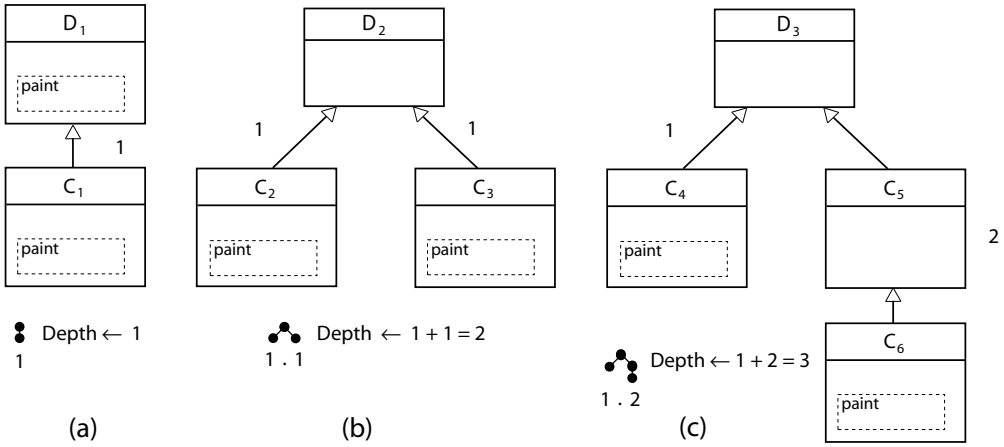

Figure 3. Duplicated methods and relative inheritance depths. Notice that depths can be represented as pairs that describe the shape of of the hierarchy. This helps differentiate between different hierarchies that share the same depth. For instance, the two distinct hierarchies that have depth 3 can represented by the pairs $\langle 3 \cdot 0 \rangle$ and $\langle 2 \cdot 1 \rangle$ (or its equivalent, $\langle 1 \cdot 2 \rangle$).

3.1. Results

To get a sense for how much of Swing's duplication is too deep to eliminate by single inheritance, we measured inheritance depths for 127 shared fragments accounting for over two thirds of the duplicated code. Of these cases we were surprised to find 58 where the code was duplicated within the same class or in an immediate superclass (case a) and 32 in sibling classes with a shared superclass (case b). Clearly duplication in Swing could be much reduced by traditional refactoring! The remaining 37 instances (or 29 percent) are prime candidates for traits. Figure 4 summarizes the distribution of inheritance depths for these candidates.

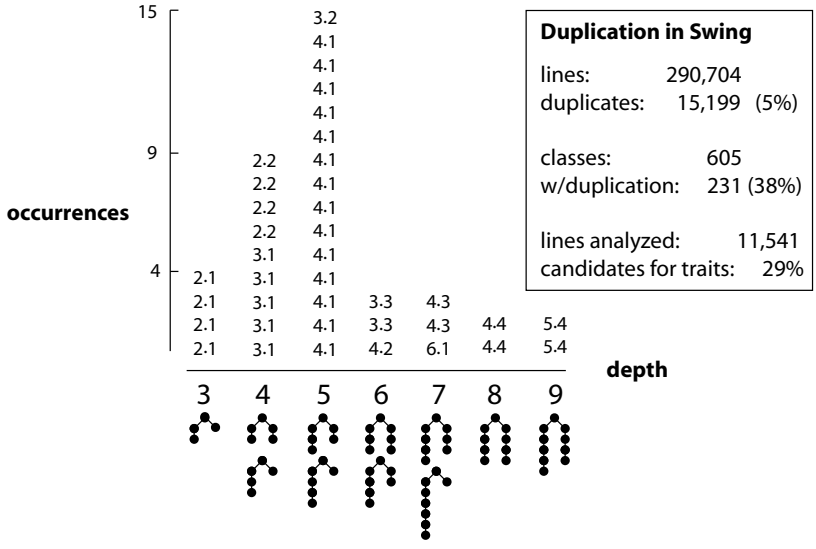

Figure 4. Distribution of inheritance depths in candidates for refactoring in Swing. For brevity, enclosing brackets are excluded from the depth pairs in the histogram. Thus, $2 \cdot 1$ in the first column should be interpreted as the pair $\langle 2 \cdot 1 \rangle$.

4. CONCLUSION

Smalltalk traits greatly reduce the need to copy and paste by providing a means to reuse behavior that is entirely separate from inheritance. In addition to lacking multiple inheritance, Java has other features that limit code-sharing. We believe that a well-designed mechanism for traits in Java could help us to overcome several of these obstacles. A naïve analysis of Swing detected 5 percent code duplication, of which at least 29 percent can be eliminated with traits but not by single inheritance. However, this is just the beginning. The CPD string-matching approach to finding duplication is extremely conservative and a more sophisticated algorithm would doubtless find more duplication. Moreover, code duplication says nothing of *logic duplication*. Our informal study of the JFC indicates that there is a great deal of logic duplication that is not detectable by such methods. Finally, it is worth noting that we only sought the most obvious opportunities to refactor to traits in looking for code that cannot be shared because single inheritance is insufficiently expressive. We believe that Java's other barriers to reuse (listed in Section 1) are responsible for a good deal more duplication, which traits could eliminate.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation of the United States (awards CCR-0098323 and CCR-031340), by Object Technology International, and by the State of Oregon's Engineering and Technology Industry Council. Many thanks to Loren Barr and Mark Jones for their comments and to the anonymous reviewers for their valuable feedback.

REFERENCES

1. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behavior," in *Proceedings of ECOOP 2003 - European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* **2743**, Springer, 2003.
2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science* **2072**, pp. 327–353, Springer-Verlag, (Berlin, Heidelberg, and New York), 2001.
3. Sun Microsystems, "Download Java 2 Platform, Standard Edition 1.5.0 Beta 1." http://java.sun.com/j2se/1.5.0/download.jsp. (April, 2004).
4. T. Copeland, "Detecting duplicate code with PMD's CPD," *On Java* , Mar. 2003. http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.html.
5. PMD, "PMD Project." http://pmd.sourceforge.net/. (April, 2004).
6. R. M. Karp and M. O. Rabin, "Efficient randomized pattern–matching algorithms," *IBM Journal of Research and Development* **32**, pp. 249–260, 1987.
7. A. P. Black, N. Schärli, and S. Ducasse, "Applying traits to the smalltalk collection classes," in *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 47–64, ACM Press, 2003.