

Supporting Java Traits in Eclipse

Philip J. Quitslund* Emerson R. Murphy-Hill* Andrew P. Black*

Department of Computer Science
Portland State University, USA

{pq,emerson,black}@cs.pdx.edu

Abstract

Traits are a language technology that complements inheritance as a means for code reuse and class structuring. Traits encapsulate collections of methods so that they can be used and reused anywhere in the inheritance hierarchy. An important property of traits is that classes structured with traits have the same semantics as classes structured without them. With environment support, a programmer can move freely between views of the system with or without its component traits. In this paper we describe an environment for programming with traits for Java implemented in Eclipse.

1. Introduction

When the system requires that you duplicate code, it is asking for refactoring.

— Kent Beck [3]

Duplicated code is anathema to reliable software: it makes systems hard to understand, maintain, and evolve. Unfortunately, duplication sometimes seems to be necessitated by the limitations of our programming languages. Languages such as Java, where the only (language supported) mechanism for reuse is single inheritance, force us to duplicate logic that is common between classes that do not (and cannot be made to) share an immediate superclass. Multiple inheritance would mitigate this problem but it introduces unacceptable complexity [15]. Although some languages, such as C++ and Eiffel, provide multiple inheritance, multiple inheritance has turned out to be so sticky in practice that programmers are discouraged from using it except in the most restricted contexts [10]; more extremely, designers leave it out of their languages al-

*This work is partially supported by the National Science Foundation under Grant No. 0313401 and by an IBM Eclipse Innovation Grant.

OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop,
Oct. 24-28, 2004, Vancouver, British Columbia, Canada.
Copyright 2004 ACM

together. As Steve Cook quips, summarizing Alan Snyder's assessment, "multiple inheritance is good, but there is no good way to do it" [6].

Traits [5, 15] are a mechanism that complements inheritance as a means for concrete reuse. In short, traits are named collections of methods that can be used multiple times anywhere in the class hierarchy. Traits sidestep classic problems with multiple inheritance but provide most of its benefits, greatly improving opportunities for reuse. Traits were originally prototyped in Smalltalk but the model is broadly applicable to OO languages.

In this paper we describe an environment for programming with traits in Java, prototyped in Eclipse [2]. We start, in Section 2, with a brief description of traits and go on to describe our implementation of traits for Java (Section 3). In Section 4 we present a trait extraction tool devised to help in refactoring legacy code to use traits. In Section 5 we sketch our trait editing environment. Some directions for future work are detailed in Section 6. In Section 7 we present our conclusions.

2. What Are Traits?

Traits encapsulate collections of methods that can be reused anywhere in the inheritance hierarchy. Unlike classes, traits have no fields and cannot be instantiated directly; they are composed into classes which, if not abstract, can be instantiated.

For example, suppose a class `ColorPoint` is composed of the traits `TColor` and `TPoint`. `TColor` and `TPoint` each define a number of methods, including `brighter()` and `translate(int,int)`. When composed into `ColorPoint`, the methods defined in each trait make up the *protocol* of `ColorPoint`. (`ColorPoint` is likely to contain other things as well, such as instance variables and accessors to those variables.) Because two composite traits might define the same method, such as `equals(Object)`, composition can be selective, allowing the user to exclude a method from the composite class; if the programmer wants

both methods, one can also be aliased, that is, given another non-conflicting name. Conflicts must be resolved before the class that uses the trait can be instantiated.

Commonly, traits depend on functionality that they do not themselves define. Traits publish *requirements* much in the same way that abstract (or template) classes provide hooks for clients to implement. In Figure 1, trait TColor *provides* `getRed()`, `getGreen()`, `getBlue()`, `darker()` and `lighter()` (the methods on the left) and *requires* `getRGB()` (on the right). Provided methods can be implemented in terms of required ones as in the definition of `getGreen()` (①). Required methods must be defined in any concrete class that uses the trait.

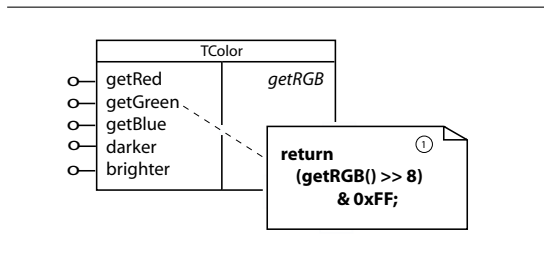


Figure 1: The TColor trait.

In addition to being used to build classes, traits can also be used to build other traits. Often, traits are quite small, encapsulating fine-grained units of functionality. Bundling functionality into small units improves understanding and opportunities for reuse.

3. Implementing Traits

In previous work [13] we implemented traits directly as an extension to mini-Java (MJ) [9], a subset of the Java language, by devising a compiler (TMJC) that works by translation, taking source extended with traits, and translating it to pure mini-Java. This approach allowed us to grow traits for Java in a user centered fashion, tailoring new syntax as we saw fit. Unfortunately, this flexibility in design came at a significant price in implementation. To ensure that TMJC only emitted type-correct mini-Java we had to perform our own type-checking. While this enabled us to do clever things (like inferring requirements) it also forced us to repeat much of the work done by the static analysis phases of javac.

Our current prototype takes a different approach: no modifications are made to the base language. Instead, traits are modelled as (possibly abstract) classes where requirements are stubbed out

as abstract methods. Trait composition is achieved by merging method dictionaries. If class TPoint is composed with TColor, for example, a new class is produced with all the non-conflicting methods of both. In the case of conflicts, concrete implementations override abstract methods and otherwise signal an error that must be resolved by the programmer via an explicit exclusion. Non-conflicting methods are directly copied into the target composition keeping all the modifiers (`private`, `final`, `synchronized`, and so on) as is. (Future work involves a parametric *uses* operation whereby modifiers could be changed [14].) In keeping with Smalltalk traits, fields are not composed but this is not essential. Field composition conflicts could, in principle, be resolved in the same way as method conflicts.

Leveraging the JDT. Our implementation makes extensive use of the JDT. Method dictionaries are essentially collections of `IMethods` extracted from `ITypes` extended with functionality to manage conflicts, exclusions and aliases. Aliasing is the most subtle feature but is neatly accomplished via the JDT’s support for AST re-writing.

4. Refactoring to Traits

When working with legacy code, traits can be used to remove duplication and reuse code in existing classes. In this context, working with traits involves a kind of refactoring: removing duplicated code, putting it into a trait, and using that trait in the classes from which the code was extracted. In practice this is cumbersome and much of the process can be automated. To support programmers in trait extraction, we designed the TraitExtractor plugin which allows programmers to select collections of methods from classes to be extracted into stateless (possibly abstract) classes that can be used in trait compositions.

Example. The `java.io` libraries contain a good deal of unavoidable duplication because they contain abstractions like streams and writers that share protocol but cannot share implementation due to the shape of the inheritance hierarchy. For example, `PrintStream` and `PrintWriter` are both adapter classes [7] that add a printing protocol to the objects they wrap; but they cannot share code because their common superclass is too far away (see Figure 2). Pushing up the print protocol shared by `PrintWriter` and `PrintStream` classes is unsatisfactory because the behavior would be implemented “too high up” and would need to be cancelled in classes in which it is not appropriate — such as `BufferedOutputStream` (①) and `BufferedWriter` (②). With traits, we can greatly improve code-sharing, removing the duplication of *twenty-nine methods* [13].

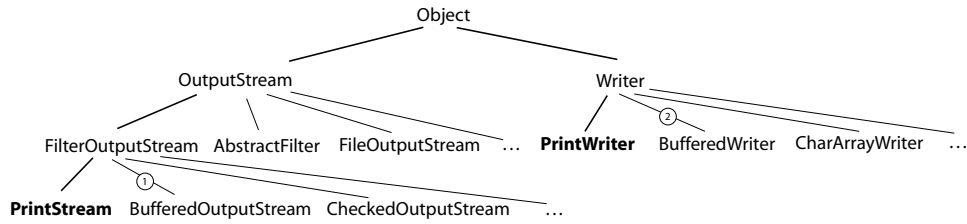


Figure 2: The PrintStream and PrintWriter class hierarchy.

To use the TraitExtractor to remove this duplication, we start by selecting a class from which to extract the duplicated methods (either will do). Choosing `PrintStream`, we select “*Extract Trait*” from the *Traits* submenu in the Package Explorer and are presented with a wizard which lets us select methods to extract (see Figure 3). Since the methods `ensureOpen()`, `flush()`, `close()`, `checkError()`, and `setError()` embody a single concern, we put them into a trait called `TIOOperation`. Before committing to the extraction, we are given a preview comparing `PrintStream` and the new trait `TIOOperation`. The trait is essentially a modified version of `PrintStream`, with the following changes: the class is now abstract, all instance variables have been removed and references to those variables have been replaced by accessor methods, and these methods have been defined as abstract. The class `TIOOperation` can now be used to remove the duplicated IO protocol in `PrintStream` and `PrintWriter`.

5. Editing Traits

Since our representation of a trait is just a class, much of Eclipse’s Java tooling can be reused whole-cloth (particularly, the robust Java editor). However, tool support for managing trait methods once they have been composed has proved invaluable for working with traits [11]. To help manage and understand classes structured with traits our environment implements a number of additional features on top of the JDT.

As mentioned in Section 3, methods provided by traits are copied into the classes that use them. Without an environment to monitor the status of these methods, a programmer might accidentally change one copy without changing the others, thus defeating the objective of traits. Instead, we provide a mechanism that monitors every copy of a trait method for changes. If a change occurs, the changed method is annotated with a marker that enables the user to take action on the modified method. For example, the user may wish to use

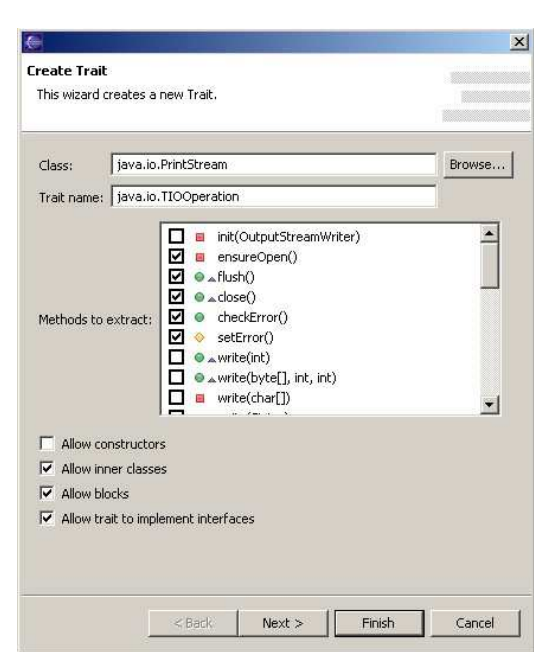


Figure 3: Extracting the TIOOperation trait.

the modified method instead of the trait method, and may exclude the trait method. Alternatively, the user might want to *push* the modified method up into the trait.

Classes composed of traits can be viewed in two complementary ways as the programmer sees fit. A class can be seen as a composition of traits, or *flattened*, with its trait substructure elided. Neither view is promoted at the expense of the other. The programmer can toggle between these two views through contributed actions as shown in Figure 4: trait-provided methods (①) can be elided by an action that folds them out of view (②). Similarly, trait methods in the outline view (④), can also be filtered out of view (③).

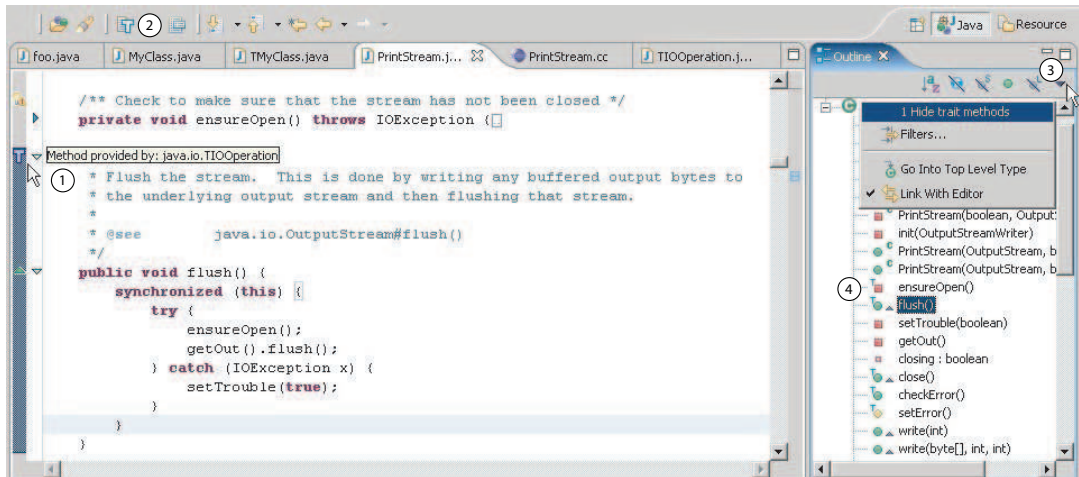


Figure 4: Editing with traits.

6. Future Work

We see a number of promising directions for future work.

- Programming with Multiple Views.** We have only just begun to understand (and our environment barely exploits) the implications of programming with complementary views. Multi-view programming [4] is widely applicable to the Aspect-Oriented paradigm and traits are an ideal place for study since their semantics are so much simpler than other new technologies like AspectJ.
- Trait-Mining.** Refactoring of legacy systems could be made much easier with automated support for identifying candidate protocols for extraction to traits. We envision strategies that go beyond simple duplication detection. A promising angle of attack is to investigate the implementation of interfaces. Since interfaces are effectively shared protocols, and are often implemented across inheritance hierarchies, there is potential for reuse of implementation via traits. Of course this approach is bounded by the degree that interfaces are used, which is often less than one might hope [8]. An interesting piece of related work is to identify opportunities to refactor to interfaces by identifying shared protocols in a target system. Such an analysis would make a good oracle for the utility of traits.
- Research Synergies.** Affinities with related research deserve exploration: Trait-mining is squarely in the range of work being done in the aspect-oriented community on concern and feature extraction. Our composition scheme bears an interesting resemblance to HyperJ [16] and CME [1]. Scala [12] shares our “unified” view of traits as special classes (as opposed to distinct programming units).
- Language Design.** A key goal of this work is to better understand how traits fare in the presence of a (strong) static type system. We have noted earlier [14, 13] that we think a `ThisType` construct is key to making Java traits expressive and so have provided such a mechanism in our system. As we gather more experience applying traits we will grow and reflect on a set of requirements for traits implemented as a language feature. On a related note, we are interested in exploring how traits should be best integrated with parametric types forthcoming in Java 5.
- Trait Model Extensions.** In a study of code duplication in the Java Swing libraries [14] we exposed a number of opportunities for reuse that involve extensions to traits. Making the uses operation parametric would allow us to use behavior defined once in different contexts (e.g., making it `final` or `synchronized` for example). This feature is easy to add to our compositional model but its implications need to be understood.

7. Conclusion

In this paper we previewed the beginnings of an implementation and supporting environment for Java traits prototyped in Eclipse. This work is just part of a greater project to build support for viewing and effecting programs from multiple equally primary perspectives [4]. We see this as an area full of promise, especially in the context of the growing interest in Aspect-Oriented Programming, which, if it is to enable us to program in a truly new way, needs to be more than just the exchange of one dominant decomposition for another. In light of this, traits are both an exciting language technology *and* platform for exploring the emerging multi-view story and Eclipse, with its rich and extensible support for Java-like languages, is an invaluable supporting player.

8. About the Authors

Philip Quitslund and Emerson Murphy-Hill are PhD students and Andrew Black is a professor in the Department of Computer Science at Portland State University, Oregon, USA.

9. References

- [1] Concern Manipulation Environment. <http://www.eclipse.org/cme/>. (August, 2004).
- [2] Eclipse Project. <http://www.eclipse.org>. (August, 2004).
- [3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [4] A. P. Black and M. P. Jones. The case for multiple views. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, 2004. http://www.cs.pdx.edu/~black/publications/Multiview_WoDiSEE.pdf.
- [5] 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*, pages 47–64. ACM Press, 2003.
- [6] S. Cook. OOPSLA '87 Panel P2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, pages 35–40. ACM Press, Oct. 1987.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.
- [8] J. Gößner, P. Mayer, and F. Steimann. Interface utilization in the Java development kit. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1310–1315. ACM Press, 2004.
- [9] M. P. Jones. Course materials for CSE511, Principles of compiler design. OGI School of Science & Engineering. (August, 2004).
- [10] S. Meyers. *Effective C++*. Addison Wesley, second edition, 1998.
- [11] E. R. Murphy-Hill and A. P. Black. Traits: experience with a language feature. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 275–282. ACM Press, 2004.
- [12] M. Odersky. The Scala Programming Language. <http://scala.epfl.ch/>. (August, 2004).
- [13] P. J. Quitslund. Java traits — improving opportunities for reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, 2004.
- [14] P. J. Quitslund and A. P. Black. Java with traits — improving opportunities for reuse. In *Proceedings of the 3rd International Workshop on Mechanisms for Specialization, Generalization and Inheritance (ECOOP 2004)*, pages 45–49. Laboratoire Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), 2004.
- [15] 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*, volume 2743 of *Lecture Notes in Computer Science*. Springer, 2003.
- [16] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.