

**Java Traits —  
Improving Opportunities for Reuse**

*Philip J. Quitslund*

Department of Computer Science and Engineering  
OGI School of Science & Engineering  
Oregon Health & Science University  
20000 NW Walker Road  
Beaverton, OR 97006-8921 USA

Technical Report Number CSE-04-005

September 2004

This material is based upon work supported in part by  
the National Science Foundation under Grant No. 0313401.  
Any opinions, findings and conclusions or recommendations expressed in  
this material are those of the author and do not necessarily reflect  
the views of the National Science Foundation.

# Java Traits — Improving Opportunities for Reuse

Philip J. Quitslund\*

September 8, 2004

## Abstract

One goal of Object-Oriented Programming is to enable programmers to craft elegant and reusable systems. In practice however, Java programmers have no choice but to copy and paste code that cannot be shared via inheritance. The resulting duplication makes systems difficult to understand and hard to maintain. Traits are a language feature, originally prototyped in Smalltalk, that directly address this barrier to reuse. Traits encapsulate collections of methods that can be reused anywhere in the inheritance hierarchy. We argue that Java would benefit from such a mechanism. To demonstrate, we present 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. In addition to a lack of multiple inheritance, our study revealed other barriers to reuse, including the use of private, final and synchronized qualifiers. Surprisingly, these are all easily overcome by a natural extension of traits.

## 1 Introduction

*When the system requires that you duplicate code, it is asking for refactoring.*  
— Kent Beck [1]

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 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 [2]. Although some languages provide multiple inheritance, such as C++ and Eiffel, multiple inheritance has turned out to be so sticky in practice that programmers are discouraged from using it except in the most restricted contexts [3]; more extremely, designers leave it out of their languages altogether. As Steve Cook quips, summarizing Alan Snyder's assessment at OOPSLA '87, "multiple inheritance is good, but there is no good way to do it" (quoted in Schärli *et al.* [2]).

---

\*philipq@cse.ogi.edu

Traits [2, 4, 5, 6] are a mechanism that complements inheritance as a means for concrete reuse. In short, traits are named collections of methods that can be used anywhere and multiple times over in the class hierarchy. Traits sidestep classic problems with multiple inheritance but provide 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 propose an extension of Java to support traits.

**Roadmap.** The paper is organized as follows. In Section 2 we give a quick introduction to traits as prototyped in Smalltalk. Next we present our proposed extension of Java with traits (Section 3) and describe a prototype traits compiler for a Java subset (Section 3.2). To motivate the value of traits for Java and to demonstrate a methodology for applying them, we present a refactoring of two classes from the standard `java.io` library in Section 4. This refactoring shows that traits have benefits beyond eliminating code duplication. In Section 5, we present a case study of duplication in Java’s Swing GUI libraries and introduce a technique for identifying candidates for refactoring to traits by a metric that we call *inheritance depth*. Using this measure, we identified duplication that cannot be factored out by single inheritance but can be eliminated with traits (Section 5.1). In addition to this duplication, our analysis exposed barriers to reuse that are imposed by Java features orthogonal to inheritance. In Section 5.2, we point out that the use of `final`, `private` and `synchronized` declaration qualifiers limits opportunities for reuse. Perhaps the greatest contribution of this work is the identification of these impediments to reuse and the insight that they can be overcome with an extension to traits. We present a preliminary design for this extension in Section 5.3. In Section 6 we describe related work and in Section 7 we propose directions for future research. In Section 8 we summarize our conclusions.

## 2 Smalltalk Traits in a Nutshell

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`. Since 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 excluding one of conflicting implementations 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).

Trait methods may also refer to a superclass using the `super` keyword. However, `super` is left unbound until the trait is composed with a class. If `ColorPoint` inherits from `DisplayObject` and uses `TColor`, and `TColor`’s `=` calls “`super hash`”, then `super` binds to `DisplayObject`’s `hash`. In this way, traits can place requirements on classes and their superclasses.

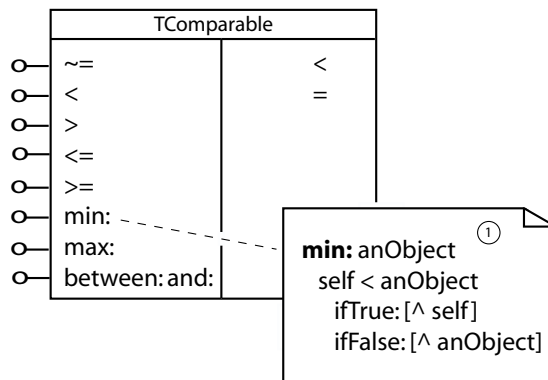


Figure 1: Trait `TComparable` *provides* `=`, `~=`, `<`, `>`, and so on (the methods on the left) and *requires* `<` and `=` (on the right). Provided methods can be implemented in terms of required ones as in the definition of `min:` (①).

**Example.** In implementing a compiler, one encounters many abstractions that are linked into lists: variable and method environments, variable and method declarations, formal parameters, and so on. Standard list operations may be required for many of these linkable objects. For example, it is common to implement a `reverse` function for use in a parser, where left-recursive productions would otherwise cause lists to be built up in reverse order. The `reverse` routine is in fact so ubiquitous that we would like to share it between all of these classes. Unfortunately this is infeasible with single inheritance as these classes should not be grouped under a common superclass. With traits, we can neatly encapsulate this behavior in a `TLinkable` trait. Now all of these sundry linkables can enjoy `reverse` functionality, and the functionality need only be implemented in one place. Further, once defined, our linkable trait can be applied in other contexts. Indeed our `TLinkable` trait only defines a small piece of behavior; If we mix it with other traits like `TAppendable`, which supports adding to the tail, and `TTailOp`, which uses `TAppendable` and adds support for removing from the tail, then we can derive behavior that could be composed into richer variations (like LIFO and FIFO queues).

### 3 Traits for Java

Traits in Smalltalk are a lightweight and flexible mechanism for the reuse of small and composable units of behavior. The methodology emerging around programming with traits is decidedly agile<sup>1</sup>, promoting iterative development, refactoring, and flexibility (avoiding premature commitments in the design of abstractions that might constrain a component’s reusability) [6]. How can we bring this experience to Java? There are reasons to expect that this will be challenging. Fundamentally, there is a synergy between the rhythm of programming with traits and Smalltalk’s dynamic types—dynamic types facilitate the development of units which are under-specified and incomplete. Can we design traits in the context of static typing and preserve agility?

<sup>1</sup>In the sense used to describe extreme programming [1, 7].

Our proposal brings the Java and Smalltalk cultures together, we think, in an amenable way. The dynamic typing of Smalltalk is simulated by a type inference mechanism that infers trait requirements, enabling programmers to avoid prematurely committing to the signatures and types of required methods. At the same time, static safety is guaranteed: our extension, like GJ’s extension for parametric types [8], ensures type-preservation. In the next two sections we briefly describe our design (3.1) and a prototype implementation for a subset of Java (3.2). A grammar for a full extension to Java is presented in Appendix A.

### 3.1 Language Design

Language designs are often best motivated by example. To illustrate our design and rationale we will step through a simple trait written in TMJ, our prototyped implementation for the language mini-Java (MJ). The discussion is informal and ‘broad brush’; implications of the design are treated as they come up. A discussion of our implementation of the traits-mini-Java compiler (TMJC) follows in the next Section (3.2).

**Example.** Our linkable trait from Section 2, implemented in TMJ, is shown in Listing 1.

---

```

trait TLinkable {
  Link reverse() {
    Link result; result = null;
    Link list; list = this;
    while (list != null) {
      Link temp; temp = list.getNext();
      list.setNext(result);
      result = list;
      list = temp;
    }
    return result;
  }
  boolean hasNext() { return getNext() != null; }
}

```

---

Listing 1: TLinkable in TMJ (first attempt).

A few things bear mentioning. First, notice that there are no explicit bounds on TLinkable (line 1). That is, although TLinkable requires *getNext()* and *setNext(\*)* methods (lines 6 and 7) these requirements are nowhere manifest in the declaration. This exposes a key design point:

**Design Point 1** *Keep Java traits lightweight.*

Instead of insisting that programmers make requirements explicit, our implementation infers unspecified requirements from the context of their use. Optionally we could declare requirements:

```

trait TLinkable requires { Link getNext(); } { ... }

```

This surfaces design point 2.

**Design Point 2** *Freedom of choice — allow users to choose between explicit or inferred requirements.*

This freedom supports a lightweight, flexible programming style with the option to declare requirements explicitly. We think this provides the ‘best of both worlds.’ While inferred requirements are sufficient and indeed preferable in many cases, there are some situations where we might want to be more explicit. (For instance for documentation purposes.)

The form of our requirements declaration illustrates another point.

**Design Point 3** *Support structural constraints (in addition to subtype constraints).*

Structural constraints promote lightweight traits. Moreover, the ability to match on signatures rather than types supports retroactive abstraction. To use a trait, a class should not need to implement an explicitly cooked-up interface. (Also notice that, beyond being cumbersome, a scheme based on types would not scale, as aliasing and exclusion produce a potentially exponential explosion of interfaces.)

By now, the astute reader might have noticed something funny about our trait definition as it stands in Listing 1. Why are we declaring a linkable to be of type `Link`? Are we not constraining the use of this trait to the concrete type `Link` (or one of its subtypes)? Setting the body of `TLinkable` aside for now we can make the same point with our declared requirement:

```
trait TLinkable requires { Link ① getNext (); } { ... }
```

By naming the type returned by `getNext()` (①) we constrain the applicability of `TLinkable` to the `Link` class (or one of its subtypes). Though legal, this is probably not what we want! To fix this we might think to define `TLinkable` recursively, like so:

```
trait TLinkable requires { TLinkable ② getNext (); } { ... }
```

but in TMJ this is an error, which highlights another design point:

**Design Point 4** *Traits are not Types.*

While perhaps initially surprising, this clearly has to be so. Because traits allow for exclusion and aliasing, trait types could make no guarantee of signature conformance. That is, although `TLinkable` provides a `reverse` method there is no guarantee that clients will not exclude or rename the method.

Although traits *are not* types, there are types in some sense *linked to* traits — that is, the concrete type of the class that instantiates them. Having reflective access to this type is extremely handy in cases like `TLinkable` and exposes our next point:

**Design Point 5** *Surface instantiating types with a `ThisType` construct.*

With `ThisType`, a number of things become possible. For one, traits can call the constructors of the instantiating type — useful for methods that need to make copies of `this`. Moreover, `ThisType` facilitates the implementation of *binary methods* [9], or methods that take a parameter of the same type as the implementing class. In `TLinkable`, this is exactly what we want: `getNext()`, `setNext(*)` and `reverse()` should be operating on things of `ThisType`. Recast, we arrive at our trait in its final form:

---

```

trait TLinkable {
  ThisType reverse() {
    ThisType result; result = null;
    ThisType list; list = this;
    while (list != null) {
      ThisType temp; temp = list.getNext();
      list.setNext(result);
      result = list;
      list = temp;
    }
    return result;
  }
  boolean hasNext() { return getNext() != null; }
}

```

---

Listing 2: TLinkable in TMJ using ThisType (final version).

**Summary.** Our design supports inferred requirements to keep traits lightweight, balanced with the option to be explicit; It provides structural constraints to support flexible expression of requirements; and, finally, it offers a `ThisType` construct to surface reflective access to the instantiating type.

### 3.2 Language Implementation

The TMJ compiler (TMJC) is *conservative*: it is fully backwards-compatible with pure Java and does not require a custom-tailored Java Virtual Machine or class loader. TMJC works by translation, taking source extended with traits, and translating it downward to pure mini-Java (a pure subset of Java). Static checks are performed on the source before translation, enabling us to ensure that TMJC produces type-safe mini-Java. Moreover, in the event of a programmer error, the compiler can generate good local error messages since the checks happen before translation. The TMJC build pipeline, translating TMJ source to MJ source and passing it to the Java compiler, is automated in an Eclipse plug-in [10] that also provides a simple TMJ editor with syntax highlighting and localized visual error messaging (see Figure 2).

## 4 Refactoring to Traits: Streams and Writers in java.io

With the basic picture outlined we are ready to explore a significant application of traits to eliminate duplication in Java’s IO libraries (`java.io`). This real-world refactoring demonstrates both the value of traits for Java and our methodology in using them.

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 [11] that add a printing protocol to the objects they wrap; but they cannot share code because their common super class is too far away (see Figure 3). With traits, we can greatly improve code-sharing, allowing for the sharing of twenty-nine methods between the two classes. There are other benefits to the refactoring as well: it produces code that is more uniform, easier to understand, and quite possibly contains less defects!

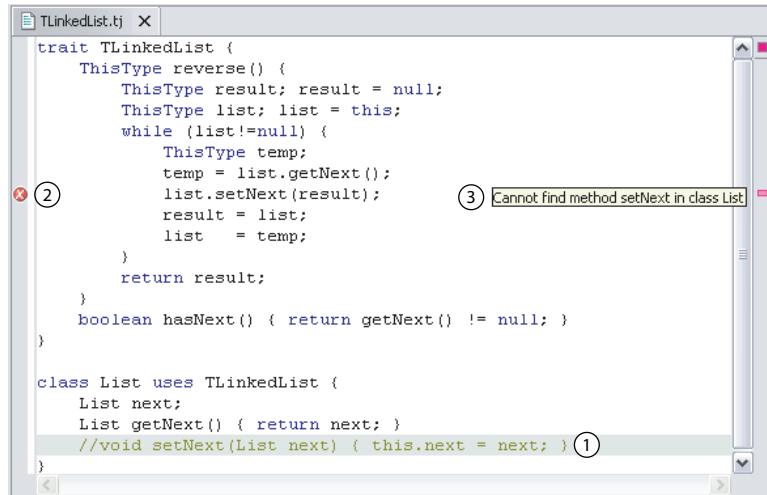


Figure 2: The TMJ editor Eclipse plug-in features localized error reporting. Here, `List` is missing a required method (①) which is annotated in the gutter at the place where it is required (②) and accompanied with a descriptive message (③).

## 4.1 Methodology

**Preliminaries.** Before refactoring we needed to do a little massaging of the code to prepare for traits.

1. **Encapsulate fields.** Since traits cannot have state, we needed first to encapsulate all fields, replacing all direct accesses to fields with calls to accessor methods. This is an established refactoring [12, 13] and is supported directly in many IDEs (e.g., Eclipse [10], IntelliJ [14], and the Smalltalk Refactoring Browser [15]).
2. **Normalize equivalences.** In a number of places we needed to identify and normalize equivalent pieces of code. Notably, the two classes use two interchangeable synchronization idioms: `PrintWriter` synchronizes explicitly on `this` and `PrintStream` synchronizes on a field, `lock`, which is initialized to `this` at object creation time. Since these are semantically equivalent, we could have picked arbitrarily; as it was, we favored the explicit ‘this’ idiom and refactored `PrintStream` to use it.

After priming, we proceeded by identifying four protocols that we separated into distinct traits: `TIOOperation` encapsulates basic I/O and error handling, `TPrintable` encapsulates the eighteen print routines, `TFormatable` encapsulates formatted printing, and `TAppendable` encapsulates appending. These traits are combined into the `TPrintProtocol` composite trait. A bird’s eye view of the refactoring is shown in Figure 4; an exploded view of the `TPrintProtocol` composite trait is shown in Figure 5.

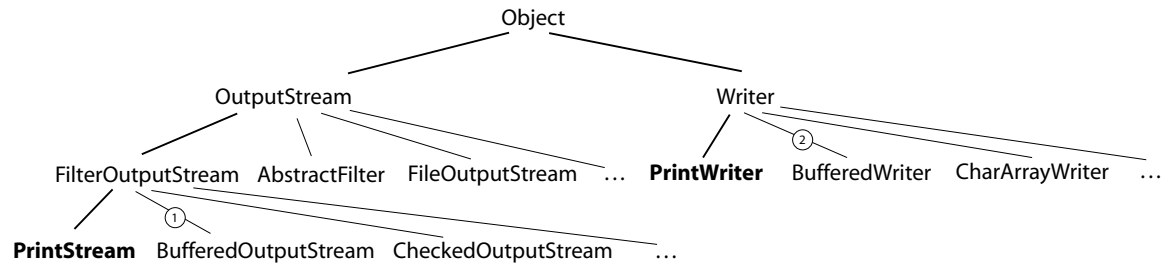


Figure 3: Pushing up the print protocol shared by `PrintWriter` and `PrintStream` classes is unsatisfactory because the behavior would be implemented “too high up” and would need to be cancelled in classes in which it is not appropriate — such as `BufferedOutputStream`(①) and `BufferedWriter`(②).

## 4.2 Discussion

With traits, we were able to share all twenty-nine duplicated methods. Of these, six required the `ThisType` construct. In the process we added seven new methods, all of them accessors (field getters and setters). We think these added methods are more than justified by the duplication they help eliminate. Moreover, the flexibility afforded by indirect access to fields is arguably a good design practice as it facilitates lazy initialization (promoting efficiency) and data validation (supporting defensive programming) [12]. By factoring out small sub-traits, we produced separate units of functionality that might be used in other combinations. For instance, `TIOOperation` might be used by other stream-oriented I/O classes or `TFormatable` might be mixed into other printable classes to add formatted printing.

**Uniformity of Protocols.** A nice benefit of using traits is that we neatly (and at no extra cost) produced implementations with more uniform protocols. Non-uniform protocols offend our sensibilities because they are not consistent with our expectations. In the original `java.io` versions, the write protocol is inconsistent between the classes—`PrintStream` and `PrintWriter` present different interfaces despite the uniformity of their underlying representations (see Figure 6). The rub is that users may be surprised if they expect `PrintStreams`, as printable things, to act like other printable things (`PrintWriters`). Copy and paste is opportunistic and so uniformity requires diligence; traits provide uniformity by design.

**Rogue Tiles.** Copy and paste programming is perilous: when code gets duplicated it is not a matter of *whether* the two versions will drift out of sync but *when*. Bugs that arise from out of sync code fragments have been dubbed “rogue tiles,” [16] likening the fragments to *tiles* and the fragments that go out of sync to *rogues*. Our refactoring exposed an interesting asymmetry in implementation that could well be a rogue tile.

`PrintWriter` and `PrintStream` share eight basic print routines that simply delegate to the `write` method. For instance, `int` printing is implemented like so:

```
public void print(int i) {
    write(String.valueOf(i));
}
```

All sixteen methods fulfil the same contract, but there is a twist: one method, `print(char)` in `PrintWriter` is implemented differently. Rather than calling `String.valueOf(char)`, this implementation

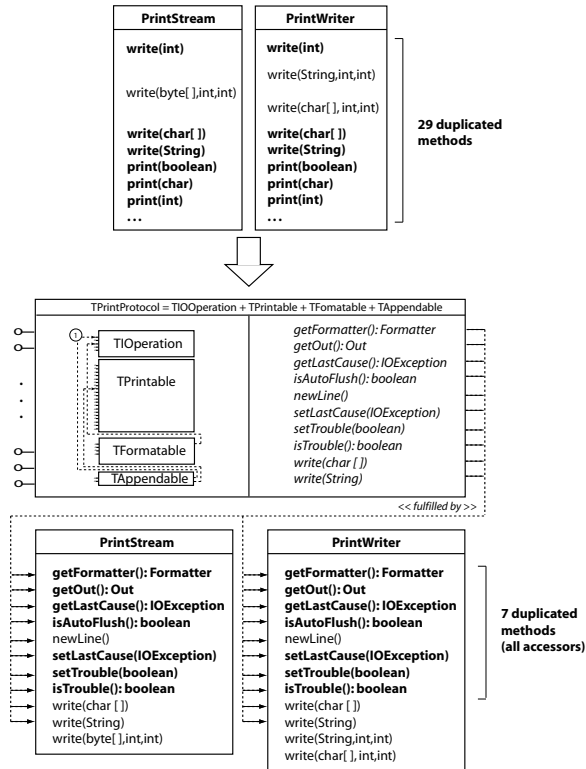


Figure 4: Refactoring `PrintStream` and `PrintWriter` to use traits. Methods with duplicated bodies are in bold. Details for the `TPrintProtocol` composite trait (①) can be found in Figure 5.

writes the `char` without conversion. This has interesting consequences. First, this implementation is *more efficient* than the rest: The `valueOf(*)` method is expensive because it creates a new `String` value—if unnecessarily called, this is clearly undesirable! On the other hand, we have to ask if this call is *correct*. In other words, is it possible that the other calls all depend on some side-effect of the `valueOf(*)` or, possibly, for some sophisticated character set conversion to function properly? In this case, the unconverted call is a bug! (See Figure 7).

**Programming by Difference.** A benefit of OO is the idea of *programming by difference* [17], which is to define new abstractions simply by defining specializations of other abstractions. Programs written this way are easy to understand because the differences between variations of abstractions are explicit and *manifest*. Copy and paste programming has exactly the opposite effect: similarities and differences are *latent* and need to be actively teased out to be understood. This relates to the asymmetry of `print(char)` described previously. While the difference in implementations of `print(char)` might be a bug, it is also possible that it reflects a sound design decision. Unfortunately, this intention is lost in the noise of gratuitous duplication. Traits reclaim the promise of programming by difference by eliminating duplication. Then, if the asymmetry is by design, then the intention is clear in the code (`print(char)` will be overridden in `PrintWriter`).

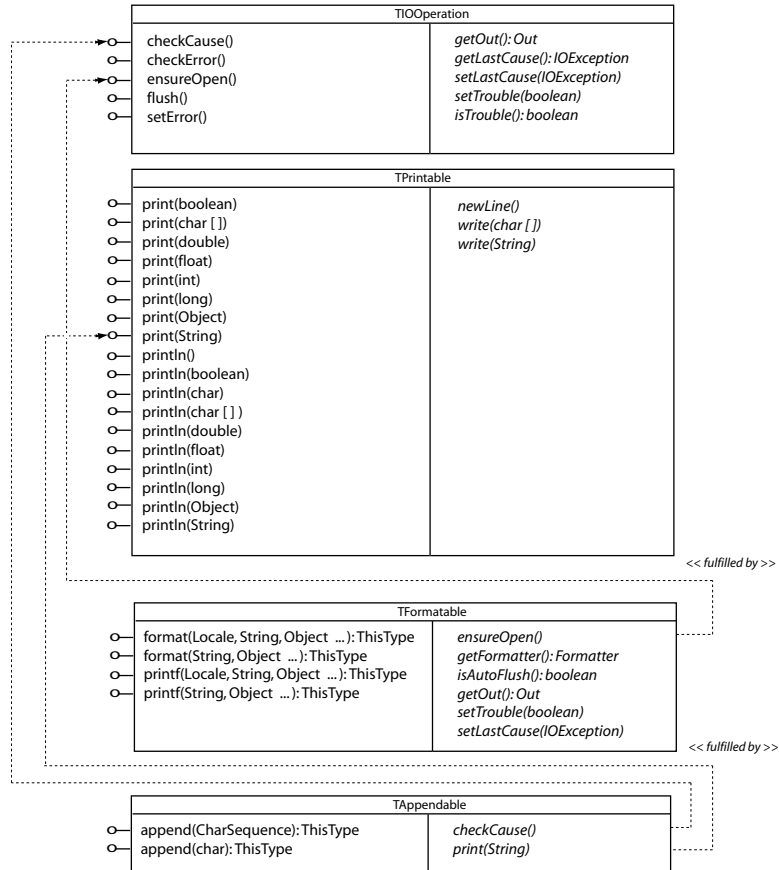


Figure 5: The TPrintProtocol composite trait.

## 5 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, and focussed on duplication that seems to result from lack of multiple inheritance. We chose Swing because it is production quality and quite large—in the JDK 1.5.0 [18], 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 [19, 20] which employs the fast (but naïve) Karp-Rabin string-matching algorithm [21]. CPD detected over 15 thousand duplicated lines across 231 classes, accounting for 5 percent of the source and 38 percent of Swing’s classes.

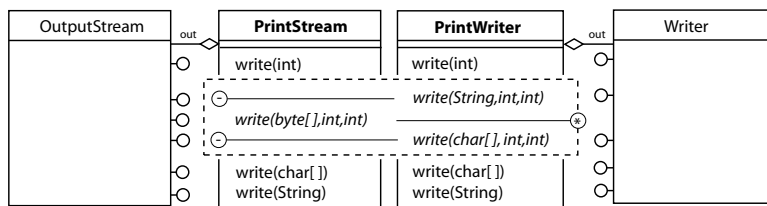


Figure 6: The write protocol is inconsistent between implementations. Despite the fact that `PrintStream`'s underlying `OutputStream` object supports writing portions of `Strings` and arrays of chars it does not export this functionality. The result is an API that surprises the user who might expect to use `PrintStreams` and `OutputStreams` in the same way.

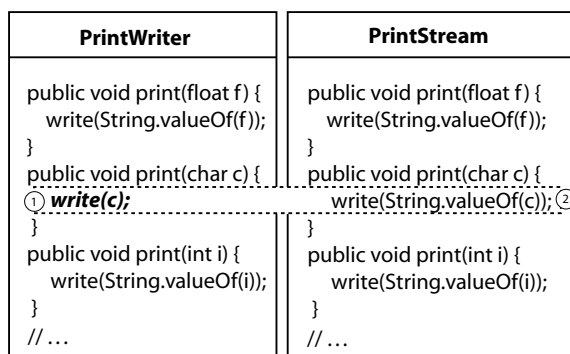


Figure 7: Copy and Paste programming is prone to implementation drift—as bugs are fixed, some fragments may not get updated. Here, is calling `write` on a `char` sufficient (①) or is the more expensive conversion to a `String` required (②) to get the appropriate behavior? As it is undocumented in the source, only the designers (or equally likely the maintainers) know for sure ... and even they might not!

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, 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 is a measure of how far from the shared superclass the classes are—a metric we will call *inheritance depth*. We define the depth of inheritance to be the sum of the distances between two compared classes and their shared superclass. Figure 8 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 more shallow 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<sub>3</sub> is that it may not be appropriate there or in C<sub>5</sub> which inherits it — here code is shared at the expense of understandability. This phenomenon has been described as putting behavior “too high” [4] and is a prime candidate for refactoring to traits.

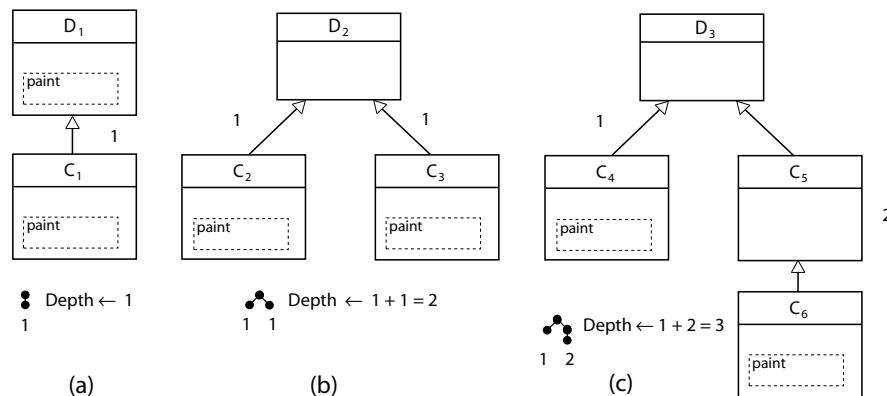


Figure 8: Duplicated methods and relative inheritance depths.

## 5.1 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 (or 29 percent) are prime candidates for traits. Figure 9 summarizes the distribution of inheritance depths for these candidates.

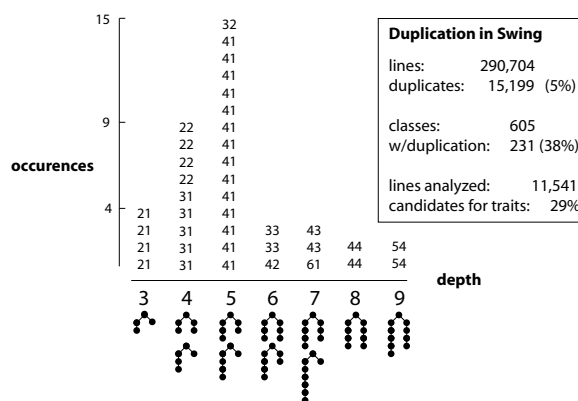


Figure 9: Distribution of inheritance depths in candidates for refactoring in Swing.

In summary, our naïve analysis of Swing detected 5 percent code duplication, of which at least 29 percent is eliminable 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.

## 5.2 Other Barriers to Reuse in Java

Our study concentrated on code duplication necessitated by the lack of multiple inheritance. We were surprised to find other features of Java impeding reuse. Beyond multiple inheritance, three features of Java are responsible for significant code duplication in the JFC.

1. **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 callbacks 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<sup>2</sup>. An example of this phenomenon can be seen in the `java.util.concurrent` package where the `FutureTask` and `ScheduledExecutor` classes both define identical, but non-reusable, private inner `ListIterator` classes.
2. **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).
3. **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.

What is striking about these limitations is that they can all be mitigated neatly and naturally with a simple extension of traits.

---

<sup>2</sup>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. This effectively 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.

## 5.3 Extending Traits for Java

The cause for Java’s limitations on reuse can be identified with its equating of subclassing with subtyping. To limit subtyping, Java also limits subclassing. Because the traits mechanism decouples these roles, it is well suited to support parametric reuse of methods and classes. The key is to generalize aliasing to allow clients to toggle declaration modifiers when using methods defined in traits. Although we do not have an implementation, we present a few reflections on a preliminary design below.

### 5.3.1 Generalizing Trait Aliasing — a Sketch

Aliasing can be generalized, trivially, if we expand our definition of selectors in aliased trait inclusions (see Appendix A.2) to include qualifiers. Having done this, we could use `equals` from a trait `TString` in a final context like this <sup>3</sup>

```
class MyString uses
  TString@{final public boolean equals(Object) aliases
    public boolean equals(Object);}–{public boolean equals(Object);}
    { // ... }
```

But this is cumbersome and we can do a bit better. With a little syntactic sugar we could make this more expressive (and terse). Using a pattern-matching scheme like that in AspectJ’s pointcut language [22], we might declare our synchronized `Vector` variation like so:

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

specifying with the wildcard that all of `TVector`’s methods should be synchronized.

**The Rub.** This simple extension heralds new opportunities for reuse. If clients could toggle declaration modifiers when using methods defined in traits the trouble with final classes and synchronized variations could be sidestepped. This way the same methods could 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.

## 6 Related Work

The decoupling of implementation inheritance and subtyping is not a new idea (see, for instance, the languages Emerald [23] and POOL-I [24]). Nor is the pitch to support a notion of structural conformance in Java novel [25]. The details of our design fit into a historical context as well. Our syntax for explicit `requires` is reminiscent of Beta’s where-clauses [26] as used in proposals to extend Java with bounded parametric types such as that of Liskov *et al.* [27, 28]. (It is noteworthy that, despite the value of structural constraints, Java generics based on the design of GJ [8], as slated for the next JDK release, will support only nominal type-based constraints.) Our strategy of inferred requirements is directly inspired by the work of Ancona *et al.* on separate compilation for Java [29] which, in turn, derives from Tichy and Baker’s pioneering paper on smart, selective

---

<sup>3</sup>Notice that we also need to *exclude* the aliased `equals` in order to avoid a conflict. Cases like this, where a method is aliased *and* excluded, recommend a `renames` operator which would allow the following rewrite:

```
class MyString uses TString@final public boolean equals(Object) renames public boolean equals(Object);.
```

recompilation [30] and its subsequent refinements [31, 32] as well as Cardelli’s work on program modularization [33]. Finally, our `ThisType` construct owes inspiration to Bruce’s proposal to extend Java with `ThisType` (along with a means to express exact types) to better support binary methods [34].

In addition to traits, there are a number of contrasting approaches to solving the code-sharing problem. For a thorough treatment of the relationship between traits and multiple inheritance, mixins and delegation, the reader is referred to the work of Schärli *et al.* [2].

## 7 Future Work

We see a number of directions for future work.

1. **Functionality.** Our current implementation of the TMJC compiler supports only a subset of Java. In the future we would like to make it more fully-featured:
  - (a) Our design does not address visibility (Smalltalk does not have visibility modifiers) but the judicious use of `private`, `protected` and `package` is common in Java programming. What should these modifiers mean in the context of traits?
  - (b) Moreover, Java 1.5 introduces new features that have potentially interesting interactions with a design for traits. How to best integrate traits with Java’s slated parametric types, improved import mechanism, and metadata code-generation facilities are an interesting area for future work.
  - (c) Finally, in Section 5.3, we explored an extension to traits aliasing that we believe will greatly improve opportunities for reuse. While a naïve implementation would be trivial, the design deserves careful consideration.
2. **Applications.** Armed with a fully featured traits compiler, we would like to apply traits to Swing and measure whether we can achieve the predicted improvements to code-sharing. We expect we can eliminate far more duplication than we detected with CPD and would like to qualify it with an empirical study.
3. **Formalization.** Our design and implementation grew organically from first principles and lack formal foundations. With a formal model we could make our tentative assertion of type preservation in translation more *iron clad*. Along these lines, Fisher and Reppy’s preliminary work on a statically typed calculus for traits [35] is a promising starting point; alternatively, Featherweight Java’s core calculus [36] provides another good point of departure.
4. **Research Synergies.** As we developed our mechanism for inferring requirements, we became aware of an affinity with Hindley Milner type inference schemes [37, 38, 39] as employed in functional languages like Haskell [40]. Type inference is well studied in the context of functional programming and in the future we would like to see how we might leverage this relationship.
5. **Implementation Strategies.** Our implementation of TMJC is conservative by design — TMJC requires no changes to the virtual machine or class loader mechanism. We think this is the best approach but other options should be considered. Along these lines, Odersky *et al.* did a comparative study of translation schemes in the context of implementing parametric polymorphism that should serve as a good point of reference [41].

## 8 Conclusion

In this paper, we have presented a design and prototype implementation for an extension of Java with traits. Our work is intentionally conservative: the language extension is fully backwards-compatible with standard Java and our implementation does not require any changes to the standard Java virtual machine or class loader. By simulating Smalltalk’s dynamic typing via type inference, our design strives to be at once flexible and type-safe.

To demonstrate the value of traits for Java in practice we presented a refactoring of code duplication found in the `java.io` libraries. In addition to eliminating duplication, we were able to produce classes that are more uniform and easier to understand. To provide further evidence of the value of traits for Java, we conducted a case study of duplication in Java Swing, a large production-quality library of classes. Using a naïve string-matching algorithm we detected that five percent of Swing is duplicated by copy and paste. By measuring inheritance depths, we ascertained that twenty-nine percent of this duplication could be eliminated by traits and not by single inheritance. But this, we think, is only the beginning.

Because our method for finding duplication is so naïve, we consider our results to be the *lower bound* on the actual duplication. As we saw in our refactoring of `PrintStream` and `PrintWriter`, not all duplicated logic can be detected by string-matching. That said, the most startling outcome of our analysis is not the duplication detected but is rather the other barriers to reuse it exposed (the use of `private`, `final` and `synchronized` declaration modifiers) — and the insight that they can be overcome by an extension to traits. Traits, with aliasing generalized, usher in new opportunities for code sharing. These opportunities were unexpected and perhaps mark this paper’s most significant contribution.

## Acknowledgements

This material is based upon work supported in part by the National Science Foundation of the United States under Grant No. 0313401, by Object Technology International, and by the State of Oregon’s Engineering and Technology Industry Council. Many thanks to Loren Barr, Andrew Black, Mark Jones and Emerson Murphy-Hill for their valuable feedback.

## A Java Traits Grammar

Here we follow the convention used by Bank *et al.* [28] and only present our extensions to Java, referring you to the Java Language Specification [42] for the rest of the details (definitions for the capitalized non-terminals can be found there).

### A.1 Type Declarations

Type declarations need to embrace classes, interfaces and traits<sup>4</sup>:

$$\langle typeDeclaration \rangle ::= \langle ClassOrInterfaceDeclaration \rangle \mid \langle traitDeclaration \rangle$$

### A.2 Class Declarations

Class declarations need to be extended to express composition from traits.

$$\begin{aligned} \langle classDecl \rangle & ::= [ \langle ClassModifiers \rangle ] \mathbf{class} \langle name \rangle [ \langle uses \rangle ] \langle ClassBody \rangle \\ \langle uses \rangle & ::= \mathbf{uses} \langle traitInclusion \rangle \{ + \langle traitInclusion \rangle \} \\ \langle traitInclusion \rangle & ::= \langle traitIdentifier \rangle [ \langle inclusionModifiers \rangle ] \\ \langle traitIdentifier \rangle & ::= \langle name \rangle \\ \langle inclusionModifiers \rangle & ::= \langle inclusionModifier \rangle \{ \langle inclusionModifier \rangle \} \\ \langle inclusionModifier \rangle & ::= \langle exclusion \rangle \mid \langle alias \rangle \\ \langle exclusion \rangle & ::= - \langle methodSig \rangle \\ \langle methodSig \rangle & ::= [ \mathbf{static} ] \langle ResultType \rangle \langle MethodDeclarator \rangle [ \langle Throws \rangle ] \\ \langle alias \rangle & ::= @ \{ ' \langle renames \rangle ' \} \\ \langle renames \rangle & ::= \langle rename \rangle \{ , \langle rename \rangle \} \\ \langle rename \rangle & ::= \langle methodSig \rangle \mathbf{aliases} \langle methodSig \rangle \end{aligned}$$

### A.3 Trait Declarations

$$\begin{aligned} \langle traitDecl \rangle & ::= \mathbf{trait} \langle name \rangle [ \langle uses \rangle ] [ \langle requires \rangle ] \langle traitBody \rangle \\ \langle requires \rangle & ::= \mathbf{requires} \langle methodSig \rangle \{ \langle methodSig \rangle \} \\ \langle typeConstraint \rangle & ::= \langle name \rangle \mathbf{extends} \langle name \rangle [ \langle param \rangle ] \\ \langle interfaceConstraint \rangle & ::= \langle name \rangle \mathbf{implements} \langle name \rangle \\ \langle structuralConstraint \rangle & ::= \langle name \rangle \{ \langle sig \rangle \{ , \langle sig \rangle \} \} \\ \langle sig \rangle & ::= \langle methodSig \rangle \mid \langle constructorSig \rangle \\ \langle constructorSig \rangle & ::= \langle ConstructorDeclarator \rangle [ \langle Throws \rangle ] \end{aligned}$$

---

<sup>4</sup>Notice, the non-terminal name should really be changed here — traits are *not* types!

## A.4 ThisType

Adding ThisType is a bit more invasive, requiring a new keyword:

```
<thisType> ::= ThisType
```

an extension to Java's Types:

```
<type> ::= <Identifier> { .<Identifier> } <BracketsOpt> | <BasicType> | <thisType>
```

and an extension to the Creator non-terminal to allow for the creation of ThisTypes:

```
<creator> ::= <QualifiedIdentifier>( <ArrayCreatorRest> | <ClassCreatorRest> )  
           | <thisType>( <Arguments> )
```

## References

- [1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [2] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew 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.
- [3] Scott Meyers. *Effective C++*. Addison Wesley, second edition, 1998.
- [4] Andrew P. Black, Nathanael Schärli, and Stéphane 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.
- [5] Stéphane Ducasse, Nathanael Schärli, Oscar Nierstrasz, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. In *Transactions on Programming Languages and Systems*, 2004. To appear.
- [6] Andrew P. Black and Nathanael Schärli. Programming with traits. In *Proceedings of the International Conference on Software Engineering 2004*, May 2004. To appear.
- [7] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.
- [8] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 183–200. ACM Press, 1998.
- [9] Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
- [10] Eclipse.org Main Page. <http://www.eclipse.org>. April 23, 2004.
- [11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.

- [12] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.
- [13] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.
- [14] IntelliJ IDEA Overview. <http://www.jetbrains.com/idea/>. April 23, 2004.
- [15] John Brant. Refactoring Browser. <http://st-www.cs.uiuc.edu/users/brant/Refactory/>. April 23, 2004.
- [16] Eric Allen. *Bug Patterns in Java*. APress, 2002.
- [17] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.
- [18] Sun Microsystems. Download Java 2 Platform, Standard Edition 1.5.0 Beta 1. <http://java.sun.com/j2se/1.5.0/download.jsp>. April 28, 2004.
- [19] Tom Copeland. Detecting duplicate code with PMD’s CPD. *On Java*, March 2003. [http://www.onjava.com/pub/a/onjava/2003/03/12/pmd\\_cpd.html](http://www.onjava.com/pub/a/onjava/2003/03/12/pmd_cpd.html). April 28, 2004.
- [20] PMD. <http://pmd.sourceforge.net/>. April 4, 2004.
- [21] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.
- [22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.
- [23] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM Press, 1986.
- [24] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.
- [25] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2000.
- [26] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 285–298. ACM Press, 1983.

- [27] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: constraining parametric polymorphism. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 156–168. ACM Press, 1995.
- [28] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145. ACM Press, 1997.
- [29] Davide Ancona, Giovanni Lagorio, and Elena Zucca. True separate compilation of Java classes. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 189–200. ACM Press, 2002.
- [30] Walter F. Tichy and Mark C. Baker. Smart recompilation. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 236–244. ACM Press, 1985.
- [31] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.
- [32] Zhong Shao and Andrew Appel. Smartest recompilation. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, January 1993.
- [33] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277. ACM Press, 1997.
- [34] Kim B. Bruce. Increasing Java’s expressiveness with ThisType and match-bounded polymorphism. Technical report, 1997.
- [35] Kathleen Fisher and John Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, December 2003.
- [36] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
- [37] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
- [38] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.
- [39] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
- [40] Simon Peyton Jones (editor). Haskell 98 language and libraries — the revised report. <http://www.haskell.org/onlinereport/>. May 2, 2004.
- [41] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your pizza - translating parameterised types into java. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 114–132. Springer, 2000.

- [42] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, June 2000.