Removing Duplication from java.io: a Case Study using Traits

Emerson R. Murphy-Hill, Philip J. Quitslund, and Andrew P. Black
Maseeh College of Engineering & Computer Science, Portland State University
Portland, Oregon USA
emerson@cs.pdx.edu, pq@cs.pdx.edu, black@cs.pdx.edu

ABSTRACT

Code duplication is a serious problem with no easy solution, even in industrial-strength code. Single inheritance cannot provide for effective code reuse in all situations, and sometimes programmers are driven to duplicate code using copy and paste. A language feature called traits enables code to be shared across the inheritance hierarchy, and claims to permit the removal of most duplication. We attempted to validate this claim in a case study of the java.io library. Detecting duplication was more complex than we had imagined, but traits were indeed able to remove all that we found.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.3 [**Language Constructs and Features**]: Classes, objects and inheritance.

General Terms

Design, Languages

Keywords

Refactoring, Traits, Java IO Class Libraries

1. INTRODUCTION

As a mechanism for code-reuse, single-inheritance has well-documented limitations. Thus, when we program in a single-inheritance language such as Java, we are often faced with a difficult choice. If we wish to reuse functionality in a new class, we must sometimes choose between (a) restructuring our inheritance hierarchy so that the new class can extend the existing class that provides the desired functionality, possibly destroying the conceptual integrity of the hierarchy or eliminating some future opportunity for reuse, and (b) reusing by copy and paste. At other times, as when the method that we wish to reuse has been defined in an indirect superclass but overridden in an immediate superclass, copy and paste is the only option.

Traits [13] are a language extension that provide a third choice. They address the need for fine-grained reuse by providing programmers with a unit of reuse smaller than a class, and independent of the inheritance hierarchy. A trait is essentially a set of pure methods (that is, methods that do not directly refer to instance variables) that can be *used* by name in a class (or in another trait). Traits provide many of the benefits of multiple inheritance while side-stepping its many pitfalls. It is claimed that traits will obviate the need to resort to copy and paste reuse, or to subvert the inheritance hierarchy to facilitate reuse.

To validate this claim in a practical setting, we studied the use of traits to refactor a large Java code base. We chose java.io as the subject of our study for several reasons.

- The java.io libraries are maintained by a mature software house and are in daily industrial use; we would expect them to be well factored.
- Since the libraries are in wide use, our results should interest a wide audience.
- The libraries are big enough to provide a significant subject for study but not so large as to be intractable, given our limited resources.
- In previous work [10, 11] we discovered some anomalies that we identified as ideal candidates for a judicious application of traits.

Our goals for the refactoring were threefold: to evaluate a trait mechanism for Java empirically; to stress-test our prototype implementation of traits; and to gain a better understanding of the consequences of the design decisions that we had made in designing the trait extension for Java. This paper describes the results of our refactoring and the insights that we gained by carrying it out.

2. WHAT ARE TRAITS?

Traits [13] are a modularity mechanism that complements inheritance as a way to share functionality. A trait is a named collection of methods (but not fields) that can be used to build up other traits and classes. If a class C is composed of traits T_1 and T_2 (we say C *uses* $T_1 + T_2$), it means that C contains all of the *non-conflicting* methods defined in T_1 and T_2. Two (concrete) methods with the same name and the same type signature, or with the same name and incompatible signatures, are said to conflict. This is because attempting to include them both in a class would generate an ill-formed class, that is, a context-sensitive syntax error.

Unlike various flavors of mixins and multiple inheritance, there are no default rules for resolving conflicts. Instead, it is the programmer's responsibility to resolve trait conflicts explicitly. To make this easier, it is possible to use a trait

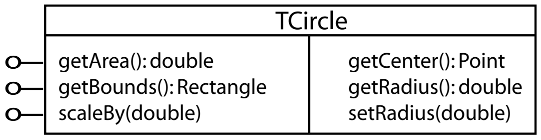

Figure 1: The TCircle trait.

selectively, that is, to use a trait but to *exclude* a subset of its methods. It is also possible to *alias* one or more methods obtained from a trait, that is, to give that method an additional name. Since methods defined directly in a class (or a trait) override those obtained from a component trait, the combination of exclusion and aliasing enable the programmer of the composite entity to reuse exactly the methods that are needed.

In contrast to Java's classes and subclassing, traits do not imply types and trait use does not imply subtyping. This is partly for philosophical reasons, and partly because the presence of exclusion and aliasing would dictate the use of a structural type system, which would not mesh well with the existing nominal type system of Java.

Traits themselves are usually *incomplete* (much like Java's abstract classes) in that they reference methods that they do not themselves define. Such *required* methods must eventually be defined in any concrete class that uses such a trait. Trait methods that are obtained by a class that uses a trait have exactly the same semantics as if they were defined in the class directly. Trait composition is orthogonal to inheritance; as a result, traits are a good way to factor out code that cannot be shared by single-inheritance (or that cannot be shared without abusing inheritance).

Example. Suppose we want to capture circle behavior in a trait. In their most basic form, circles need a center and a radius from which we can calculate area, bounds, and so on. Figure 1 shows a TCircle trait that captures this basic functionality. TCircle *provides* the methods getArea(), getBounds() and scaleBy(...) (on the left) and *requires* methods getCenter(), getRadius() and setRadius(double) (on the right). For TCircle to be used by a concrete class, all of these requirements must be satisfied, perhaps by accessor methods. Figure 2 shows a class ColorCircle composed from TCircle and from another trait TColor. To satisfy trait requirements, ColorCircle provides state (for center, radius, and rgb data) and the associated accessor methods.

2.1 Implementing Traits for Java

In our implementation, traits are represented as stateless Java classes. Requirements are expressed as abstract methods and, unsurprisingly, if a trait class has unsatisfied requirements it must be declared abstract. Classes representing traits can be composed with other classes to produce composite classes. Composite classes that supply all the methods that they require are concrete and can be instantiated.

The alert reader will note that given this representation of traits, a class can be used both through inheritance *and* through composition. That is, a class can be *extended* and also *used*. This has the nice effect that when a class is to be enlisted as a *type* it can be extended and when it is meant solely as a unit of *implementation reuse* it can be used as a sub-component. This is in contrast with our earlier proposal for typed traits, which treated them as special program units [10], and very much in line with the unified view of traits provided by the language Scala [8]. However, we did not use this feature in refactoring java.io, so cannot comment on its practicality.

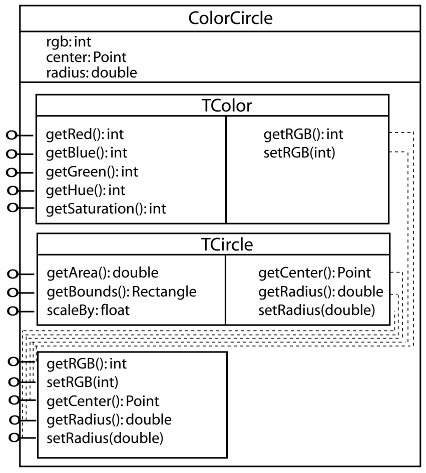

Figure 2: The ColorCircle class.

The advantage of representing traits as (suitably restricted) classes is that we can leverage existing Java development tools. Our traits programming environment is built on the Java tools provided by Eclipse and is described in detail elsewhere [12].

Example. An implementation of the TCircle trait is shown in Listing 1. Notice that the *required* methods, getRadius, setRadius(double) and getCenter are declared abstract. When TCircle is composed with another class or trait these abstract methods are replaced by any concrete implementations provided. For instance, were we to compose TCircle into a class ColorCircle as in Figure 2, the required methods getRadius, setRadius and getCenter would be replaced by ColorCircle's concrete ones.

```
abstract class TCircle {
    /* required */
    abstract double getRadius();
    abstract void setRadius(double radius);
    abstract Point getCenter();
    /* provided */
    double getArea() {
        return PI*pow(getRadius(), 2);
    }
    void scaleBy(double scale) {
        setRadius(getRadius()*scale);
    }
    Rectangle getBounds() {
        Point center = getCenter();
        return new Rectangle(
            new Point(center.x − radius,
                center.y − radius),
            new Point(center.x + radius,
                center.y + radius));
    }
}
```

Listing 1: A TCircle trait in Java.

3. REFACTORING JAVA IO

The Java IO library (java.io) provides abstractions for system input and output. Java IO forms a medium-sized code base; in Java version 1.4.2_05, there are 79 public classes and about 25K lines of code. Unless otherwise noted, the code snippets discussed in this paper come from this version.

While planning the project, we laid out a number of steps to achieve our goals. We intended first to use existing duplication detection tools to identify duplication in the java.io library, then to filter those results manually to ascertain which instances of duplication were candidates for refactoring into traits, next to use our tools to perform the refactoring, and finally to measure and characterize the benefits of using traits.

3.1 Finding Duplication Automatically

Our plan to identify candidates for duplication elimination using tools was only partially successful, because the available tools proved inadequate. We identified three candidate tools: the Duplication Management Framework [4], SimScan [14], and PMD [9]. We selected PMD, a language-agnostic tool for static code analysis that includes a subtool called CPD (for Copy-Paste-Detector), which utilizes the Karp-Rabin string matching algorithm [5]. While other duplication detection tools have more sophisticated, Java-specific detection algorithms based on abstract syntax tree matching, we found that in practice they did not return more sophisticated kinds of duplication than did CPD.

CPD led us to some clearly duplicated code, but we found that it could be used only as a starting point from which we would find more duplication. For instance, CPD identified only 3 duplicated instances of the following code:

```
} else if (b == null) {
    throw new NullPointerException();
} else if ((off < 0) || (off > b.length) ||
    (len < 0) || ((off + len) > b.length) ||
    ((off + len) < 0)) {
        throw new IndexOutOfBoundsException();
} else if (len == 0) {
    return 0;
}
```

We called code of this form a *bounds check*. We had an intuition that we had seen similar code elsewhere, so we used two manual techniques to find other instances.

3.2 Finding Duplication Manually

The first technique was based on the reasoning that however a method does bounds checking, an IndexOutOfBoundsException would be thrown. By searching for references to this exception class, we found 27 methods that perform a bounds check of some sort. It appeared that methods that implement bounds checking all take one of the following forms, where <primitive> denotes either char or byte:

- write(<primitive>[ ] array, int index, int length) - Copies length primitives from array, starting at index.
- read(<primitive>[ ] array, int index, int length) - Copies length primitives into array, starting at index.
- readFully(<primitive>[ ] array, int index, int length) - Similar to read, except that less than length primitives may be read.

The bounds check code is necessary in each of these methods to ensure that index and length are reasonable with respect to the array.

The second technique was to find all methods with these signatures. We found a total of 45 methods: 18 implementations of write, 23 of read, and 4 of readFully.

In addition to finding duplication, we thought we might discover some write, read, or readFully methods that mistakenly omit the bounds check. Of these 45 methods, 27 actually do the bounds check. Generally, methods that delegate to other java.io objects do not need to do a bounds check, because the check is done by the delegate. We reasoned that a significant amount of code would be reused if we could factor out the 27 instances of bounds checking into a trait or set of traits. Furthermore, if any of the remaining 18 methods should do bounds checks but do not, then using these traits would make the implementation more robust. However, since there appeared to be several different versions of bounds checking, we had to manually identify which were candidates for traits refactoring — those that are semantically identical, even if syntactically distinct — and which candidates were genuinely special cases.

3.3 Filtering and Applying Traits

Our expectation that we would need to filter the set of duplicates to find good candidates for trait refactoring was based on a previous study of Java's Swing graphics libraries. In that study we identified a good deal of duplication that could be removed by traditional means [10, 11]. In some cases the duplication was between sibling classes in the inheritance hierarchy and so could be removed by introducing a common superclass; in other cases the duplication was contained in the *very same class* and could be factored into one method. In contrast, we found that the java.io hierarchy is quite well-factored. None of the duplication that we discovered could be removed neatly by inheritance; all of it could be removed using traits.

After we had identified as much duplication as CPD and our intuition allowed, we used our refactoring tools to remove that duplication by applying traits.

4. RESULTS

The results of our refactoring are shown in Figures 4-6, where rectangles represent classes and rounded rectangles represent traits. All of the duplication that we removed was found in the hierarchy of Streams, Readers and Writers. Figure 7 gives a bird's eye view of the classes to which we applied traits. Figures 4-7 should give the reader a sense of where duplication was removed and how it pervades the library. In total, we used 14 traits to refactor 12 classes, removing 30 duplicated methods.

All of our applications of traits factor out redundancy between classes. (This is different from the refactoring of the Smalltalk collections classes [2] where some traits were introduced purely for conceptual partitioning and thus were used only once.) Moreover, all of the redundancy would either be impossible to remove using single inheritance, or would be difficult to remove because it would require putting the shared behavior in an inappropriate place. This is the problem identified by Schärli and his colleagues of putting behavior *too high* in the hierarchy just so that it can be shared [13]. This is an abuse of inheritance: the problem is that behavior placed too high in the hierarchy is (by definition) inherited

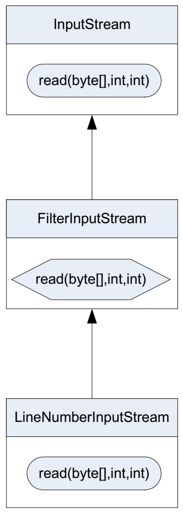

Figure 3: Method cancellation in FilterInputStream.

by classes that should not have it, straining the conceptual integrity of the inheritance hierarchy and making the true functionality of a class harder for a programmer to grasp.

Figure 7 helps us illustrate an example of this problem. The TClosing trait encapsulates the methods close() and receivedLast() whose implementations are shared by the classes PipedInputStream and PipedReader. Were we to factor out this duplication by putting the methods in a shared superclass (by introducing a common superclass for InputStream and Reader), the intervening classes (all the other Readers and InputStreams) would wrongly inherit close() and receivedLast().

To counteract the effects of behavior that is shared by being put too high, developers might seek to *cancel* the behavior. Indeed, this is just what happens in the case of FilterInputStream (see Figure 3). FilterInputStream redefines read(...) and skip(...) behavior from InputStream only to have its subclass LineNumberInputStream restore (by copy and paste) the versions from InputStream. Traits provide a clean solution to this problem: the similar behavior is defined in a trait and applied *above* and *below* the cancellation (that is to InputStream and LineNumberInputStream). We addressed a similar cancellation of behavior in the Reader hierarchy where skip(...) is redefined by BufferedReader between Reader and LineNumberReader (see Figure 5).

5. REFLECTIONS

Our experience refactoring java.io with traits has led us to evaluate tools for duplication detection, explore other methods for identifying duplication, and identify problems that are caused by duplication in a large code base.

5.1 The Trouble with Automation

Our experience with duplication detection tools was disappointing because they allowed us to discover only the tip of the iceberg. However, we realize that detecting duplication is difficult in many cases. In our experience, the theoretically easiest case to detect, copy-and-paste duplication, turns out to be difficult to detect in practice. We have found instances of identically copied code, even down to strange formatting, where expression order is changed as a slight improvement in efficiency. This is a variation of the Rogue Tile bug [1], which we observed repeatedly while reviewing different versions of the library. While an argument can be made that duplication is harmless when it can simply be refactored out later, this argument fails when copied and pasted code slowly shifts out of sync over the life of the code base. Elimination of such duplication is difficult because:

1. it is difficult to automatically detect all instances of slightly different code with tools, and

2. a significant amount of programmer time is wasted deciding when slightly different code is refactorable, that is, when it contains no semantic differences.

We found that these difficulties were major barriers to identifying duplication in java.io.

These problems were compounded by more subtle duplication. We encountered code that was apparently not copied and pasted, but was none-the-less highly similar. For example, consider the write methods in BufferedOutputStream and ByteArrayOutputStream in JDK 1.5:

```
public synchronized void write(byte b[],
        int off, int len) throws IOException {
    if (len >= buf.length) {
        flushBuffer ();
        out.write(b, off, len);
        return;
    }
    if (len > buf.length − count) {
        flushBuffer ();
    }
    System.arraycopy(b, off, buf, count, len);
    count += len;
}
```

Listing 2: BufferedOutputStream.

```
public synchronized void write(byte b[], int off,
        int len) {
    if ((off < 0) || (off > b.length) ||
        (len < 0) || ((off + len) > b.length) ||
        ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return;
    }
    int newcount = count + len;
    if (newcount > buf.length) {
        byte newbuf[] =
          new byte[Math.max(buf.length << 1, newcount)];
        System.arraycopy(buf, 0, newbuf, 0, count);
        buf = newbuf;
    }
    System.arraycopy(b, off, buf, count, len);
    count = newcount;
}
```

Listing 3: ByteArrayOutputStream.

Since they are both subclasses of OutputStream, we would expect them to have similar semantics. Indeed this is the case, but that fact is not at all obvious by manually inspecting the code. For instance, these methods appear to be different in that one performs a bounds check and the other does not, so we might conclude that these methods handle exceptions differently. However, upon closer inspection, we see that in BufferedOutputStream, the check is actually delegated to out.write(...) or System.arraycopy(...). The expectation that the delegate method will perform a bounds check is not specifiable in Java and not documented as far as we can tell. The assumption may have been understood when the code was written but an unstated assumption is worthless during software maintenance.

The point of the last example is that some duplication is very subtle. Since it took us a significant amount of time to identify duplication between these methods, the tools' inability to detect duplication is unsurprising. Understandability of the code is diminished by this type of duplication.

5.2 Manual Duplication Detection

Although duplication in its many forms is difficult to detect in practice, we have discovered some valuable heuristics for identifying it.

We have already mentioned that once we spotted duplication between methods of the same signature, we looked for duplication in other methods with that signature. This turned out to be a fruitful approach, since many instances of both obvious and subtle duplication were identified in this way. However, tool support for this task was lacking since we were unable to find all implementations in less than four searches. This is because Eclipse offers searching only for concrete method signatures, while we wanted all methods named either "read" or "write" with the first argument either "char[ ]" or "byte[ ]." Text search power tools such as Unix egrep could have done this with a single search. CME [3] promises to bring powerful query support to Eclipse.

Another way to identify duplication might be to look for implementors of Java interfaces. We hypothesize that if two methods share the same interface, they are likely to share a similar implementation. We were not able to test this reasoning during this project, because there are very few interfaces in the java.io library.

Yet another way to find candidates for duplication is to follow exception handling. We hypothesize that if a special type of exception is thrown, then the conditions under which the exception occurs are similar, and the test for the exception should be similar. This technique emerged as a fast and easy way to find a certain kind of duplication.

5.3 Perils of Duplication

Code that contains duplication is an anathema; it is difficult to understand and even more difficult to maintain. Matters become much worse as the system evolves and pieces that were once identical drift slowly apart. Such code poses the question (unanswerable at maintenance-time) of whether the differences are intensional or accidental. Refactoring such diverging duplication is hard to do with any confidence because the relationships between the duplicated pieces are only implicit in the system and so cannot be explicitly managed by tools.

To illustrate these issues, we describe the different kinds of non-uniformity that we encountered while looking at the bounds checking code. The first type of divergence was whether the array being checked contains either chars or bytes. Both versions were semantically equivalent, since the bounds checking does not access any of the elements in the array. However, any method-level refactoring of the bounds checking code cannot take the array as an argument, because char[ ] and byte[ ] have no common supertype.

The second type of non-uniformity was the syntax of the bounds checking code itself. We discovered four categories of bounds checking. The "basic" bounds check is shown in Listing 4. This is the most widely used category. A "condensed" bounds check is shown in Listing 5. It is a slightly more efficient version of "basic", where there is one less comparison and one less addition. A "bit-wise" bounds check is shown in Listing 6. This is also a slightly more efficient version of "basic"; it uses a bit-wise OR to perform only one comparison. The final category of bounds checking is "other", in which we could see no pattern.

```
public int read(char cbuf [], int off , int len)
        throws IOException {
    synchronized (lock) {
        ensureOpen();
        if ((off < 0) || ( off > cbuf.length ) ||
            (len < 0) || (( off + len) > cbuf.length ) ||
            ((off + len) < 0)) {
                throw new IndexOutOfBoundsException();
        } else if (len == 0) {
                return 0;
        }
        ...
    }
    ...
}
```

Listing 4: The "basic" bounds check (StringReader).

```
public int read(byte [] buf, int off , int len)
        throws IOException {
    if (buf == null) {
        throw new NullPointerException();
    }
    int endoff = off + len;
    if ( off < 0 || len < 0 || endoff > buf.length || endoff < 0) {
        throw new IndexOutOfBoundsException();
    }
    ...
}
```

Listing 5: The "condensed" bounds check (ObjectInputStream).

```
public synchronized int read(byte b [], int off , int len)
        throws IOException {
    getBufIfOpen(); // Check for closed stream
    if ((off | len | (off + len) |
            (b.length - (off + len))) < 0) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return 0;
    }
    ...
}
```

Listing 6: The "bitwise" bounds check (BufferedInputStream).

```
public int read(char cbuf [], int off , int len)
        throws IOException {
    synchronized (lock) {
        ensureOpen();
        try {
            if (len <= 0) {
                if (len < 0) {
                    throw new IndexOutOfBoundsException();
                } else if ((off < 0) || ( off > cbuf.length)) {
                    throw new IndexOutOfBoundsException();
                }
                    return 0;
                }
            ...
        }
    }
    ...
}
```

Listing 7: The "other" bounds check (PushbackReader).

We were curious as to why this non-uniformity existed, so we compared different versions of Java: 1.2.2, 1.3.0_05, 1.4.0_04, 1.4.2_05, and 1.5.0_01. With respect to bounds checking, we found no changes in 1.4.2_05 and later. Ultimately, we found no reasonable pattern to describe which type of bounds check was done in what context. In version 1.2.2, 17 uses of "basic" were encountered, but by 1.4.0_04, two of those checks were replaced by "bitwise," while one "bitwise" and three "condensed" emerged elsewhere. This indicates that duplication was consistent in early versions of the library, but the duplication gradually diverged as the library evolved. The exceptional case is the "other" pattern, found only in PushbackReader, which has remained untouched and unreconciled since version 1.2.2.

The third type of non-uniformity is in exception handling. While we suspect that the actual exception handling implemented in read, readFully, and write conforms to the exception handling described in each method's documentation, we found it exceedingly hard to verify that this is the case. For example, consider the read methods for InputStream and PushbackInputStream:

```
public int read(byte b [], int off, int len)
        throws IOException {
    if (b == null) {
        throw new NullPointerException();
    } else if ((off < 0) || (off > b.length) ||
                    (len < 0) || ((off + len) > b.length) ||
                            ((off + len) < 0)) {
                        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return 0;
    }
     ...
}
```

Listing 8: InputStream's read(...) method.

```
public int read(byte [] b, int off, int len)
        throws IOException {
    ensureOpen();
    if ((off < 0) || (off > b.length) ||
        (len < 0) || ((off + len) > b.length) ||
                    ((off + len) < 0)) {
            throw new IndexOutOfBoundsException(); }
    if (len == 0) {
        return 0;
    }
     ...
}
```

Listing 9: PushbackInputStream's read(...) method.

Apart from the fact that the top-level control flow is oddly different, different exceptions may be thrown in each case. Suppose, in both cases, b is null and off is −1. In InputStream, a NullPointerException will be thrown. In PushbackInputStream, an IndexOutOfBoundsException will be thrown, since || evaluates lazily.

The fact that we spent hours inspecting, rationalizing, and reconciling the many slightly different bounds checks is a stern warning against duplication. In fact, we were so confounded by the many slightly different versions of bounds checks, that we were unable to confidently refactor this code using traits. A test suite or program verifier would be beneficial in refactoring java.io, but lacking these tools we were not confident that our changes would not introduce subtle bugs.

Nevertheless, we can describe how traits can alleviate this duplication. Any version of bounds checking shown thus far can be written into two methods, boundsCheck(char[ ],int,int) and boundsCheck(byte[ ],int,int). These methods can be incorporated into a new trait. Any class that wishes to do a bounds check may use this trait, and may call the appropriate version of boundsCheck(...), as if it were defined in its own class. Using traits in this way would solve most of the non-uniformity problems we have described. Someone visiting java.io could see that many methods share a common bounds checking routine, and could infer that those that did not were special cases. If in the future an inventive programmer found a way to make bounds checking more efficient, only one trait would have to be changed.

6. FUTURE WORK

Over the course of our study we identified a number of ways that our traits environment as well as the language feature itself could be improved.

Environment. A major tool improvement that we identified was the ability to apply multiple traits to a class and the ability to apply a trait to multiple classes, both in a single graphical operation. In the Squeak traits implementation [15], you may apply multiple traits to one class in a single operation, but you cannot apply a trait to multiple classes in one operation. In our Eclipse-based traits environment, you can do neither; the tools allow you to apply only one trait to one class at a time. Although implementing this feature did not occur to us earlier, our experience makes the importance of this feature abundantly clear; almost without exception, whenever we applied a trait, there were two or more classes to which it was applied. We feel this feature would be beneficial in Squeak and Eclipse, or in any other future traits environment.

Language. A significant limitation of our Java implementation traits is their lack of parameterized types. We hinted at this in Section 5.3, while discussing the bounds checking code. Essentially, we have seen two different versions with respect to the type of array whose bounds should be checked – one an array of chars, the other an array of bytes. In order to create a method in a trait that handles both of these cases, we would need a trait with parameterized types:

```
abstract class TBoundsCheck<T1> {
    boolean boundsCheck(T1 b[ ],int off, int len) {
          ...
    }
}
```

A class that uses such a trait would specify the type at *use-time*:

```
class ObjectOutputStream uses TBoundsCheck<byte> {
    public void write(byte [] buf, int off, int len)
        throws IOException {
            this.boundsCheck(buf,off,len);
            ...
    }
     ...
}
```

Note that this parameterization is not supported by Java 5 generics because chars and bytes are primitive types and only reference types can be used as type parameters. Our usage in traits, as we have seen throughout this experiment, would require parameterization by primitive types as well as by reference types.

7. RELATED APPROACHES

Elsewhere, Schärli and his colleagues weigh traits against multiple inheritance, mixins, and use of delegation (such as in the strategy design pattern) [13]. There are yet other approaches; especially relevant to our refactoring are aspects and static utility classes.

Aspects. Aspects [7], as realized in AspectJ [6], an aspect-oriented extension to Java, are a promising technology that supports the modularization of features that cross-cut the class hierarchy. Behavior that we have parcelled up into traits could also be put into an aspect and *introduced* into the target by a mechanism like AspectJ's *intra-class declarations* (essentially open classes). Indeed, for the bounds-check example, aspects are especially attractive. Using AspectJ we could define bounds check behavior in an apsect and define a pointcut that captures all calls to write or read methods. This scheme would allow us to neatly define the protocol in one place and enforce it globally. With traits, it is still up to the programmer to apply the TBoundsCheck trait to the appropriate methods.

Our other factorings are less amenable to aspectization. While, strictly, they could all be factored into aspects, we feel this would be inappropriate. Traits are much lighter weight and conceptually more straightforward than aspects and our other refactorings benefit from the relative simplicity of traits. Traits like TPrinting would gain nothing if they were implemented as aspects and would arguably become less conceptually cohesive.

Static Utility Classes. Another approach to the bounds-checking refactoring would be to factor the bounds check into a static method of a utility class. Though tenable, this solution is unsatisfying. It eliminates duplication at the expense of conceptual integrity and encapsualtion. We argue that bounds-checking code *belongs* in the class that uses it (or a superclass), rather than in a utility class. Moreover, bounds-checking is an atypical example because it is, in practice, uncommon to have traits that define only static behavior. More commonly, static methods would require client classes to expose private state (for example, ensureOpen() in TIOOperation accesses the underlying OutStream reference which is protected; for ensureOpen() to be externalized, this field would need to be exposed, breaking its encapsulation).

8. CONCLUSION

In this paper we have described the process of refactoring a major Java library using a language feature called traits and tools and heuristics to detect duplication. While the literature often warns against the problems arising from duplication, we have witnessed the subtlety, pervasiveness, and danger first hand. We regard our refactoring as only a taste of what traits could do for the java.io library, since we were often hindered by the complexities. With more sophisticated tools for duplication detection, and with further refinement to our Eclipse-based trait tools, we believe that traits can be effective in removing duplication from industrial strength Java libraries.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0520346. Many thanks to Dick Hamlet, Chuan-kai Lin, Daniel Vainsencher and the anonymous reviewers whose comments helped to improve its content and presentation.

10. REFERENCES

[1] E. Allen. *Bug Patterns in Java.* APress, 2002.

[2] 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.

[3] CME: Concern Manipulation Environment. http://www.eclipse.org/cme/. April, 2005.

[4] Duplication Management Framework for Eclipse. http://freshmeat.net/projects/dupman/. April, 2005.

[5] R. M. Karp and M. O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.

[6] 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*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[8] M. Odersky. The Scala Programming Language. http://scala.epfl.ch/. (April, 2005).

[9] PMD. http://pmd.sourceforge.net/. April, 2005.

[10] P. J. Quitslund. Java traits — improving opportunities for reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, 2004.

[11] 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 Systèms de Sophia Antipolis (I3S), 2004.

[12] P. J. Quitslund, E. R. Murphy-Hill, and A. P. Black. Supporting Java traits in Eclipse. In *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange*, pages 37–41, New York, NY, USA, 2004. ACM Press.

[13] N. Schärli, S. Ducasse, O. Nierstrasz, and A. 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.

[14] SimScan (Similarity Scanner). http://www.blue-edge.bg/download.html. April, 2005.

[15] Traits Prototype in Squeak. http://www.iam.unibe.ch/ ˜schaerli/smalltalk/ traits/traitsPrototype.htm. July, 2005.

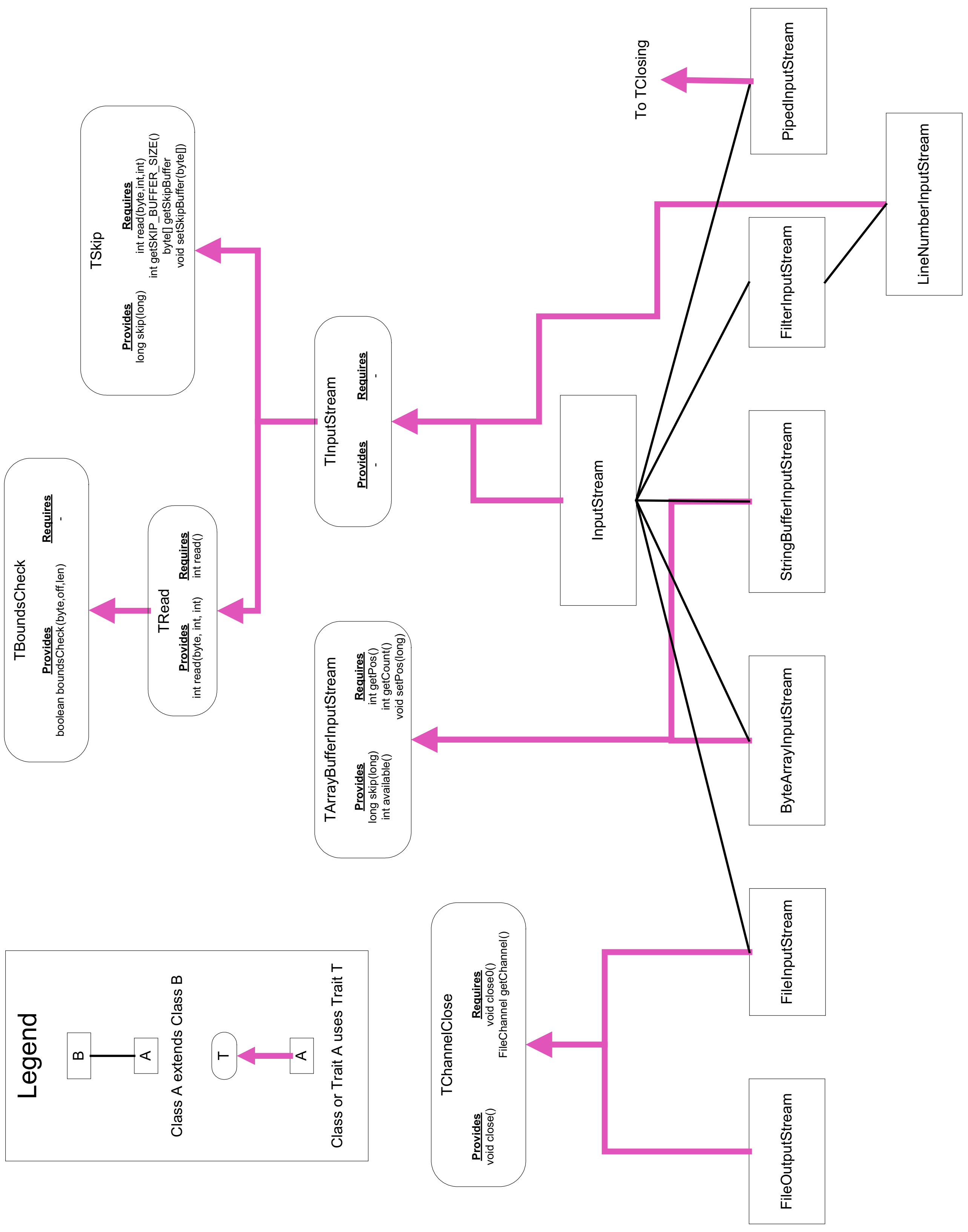

Figure 4: Traits applied to InputStreams.

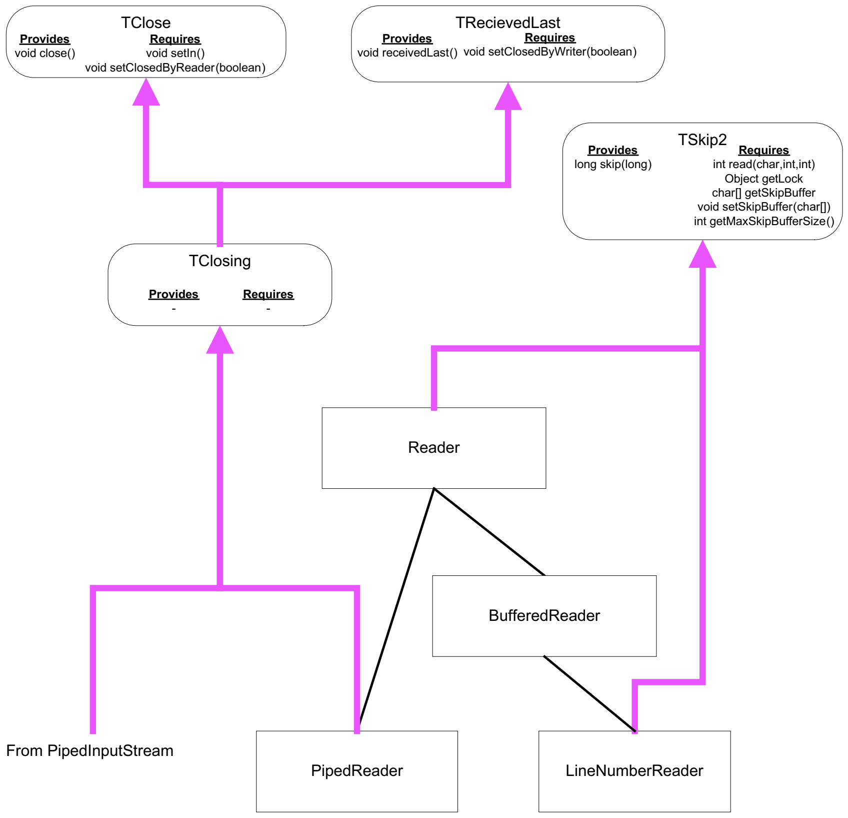

Figure 5: Traits applied to Readers.

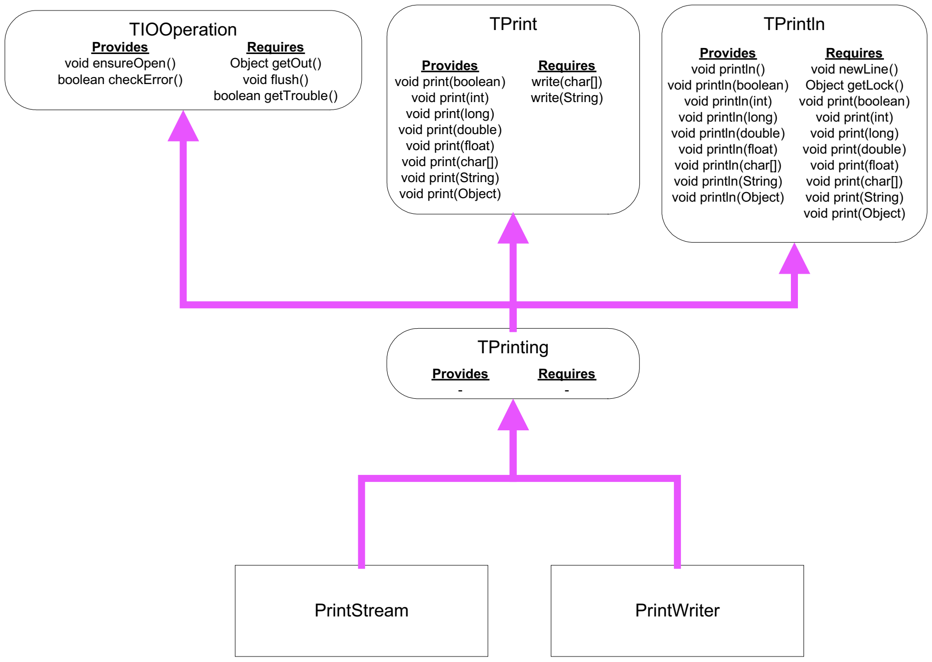

Figure 6: A TPrinting applied to PrintStreams and PrintWriters.

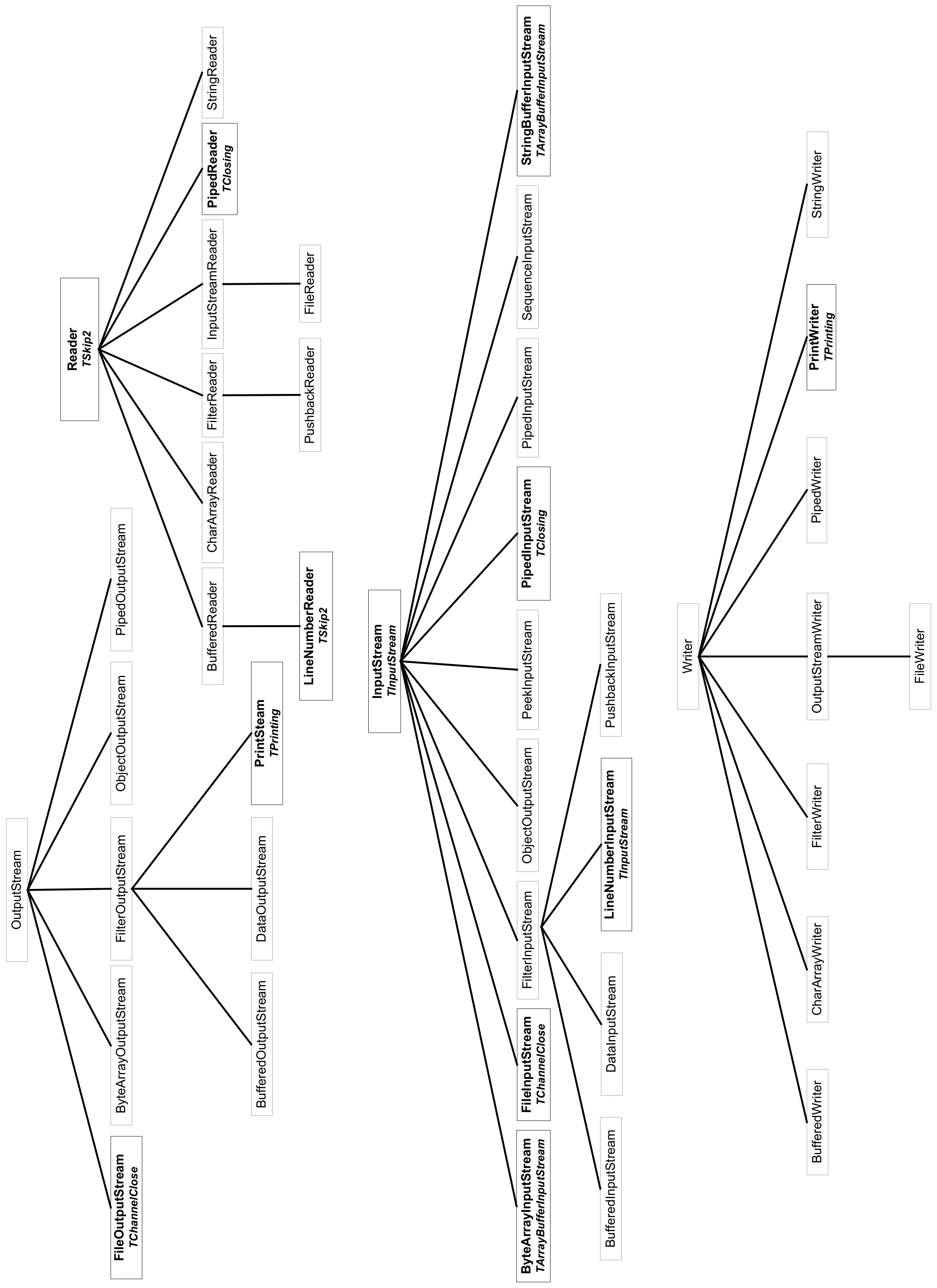

Figure 7: Application of traits to Streams, Readers and Writers in java.io.