Abstract

We explore a scalable programmer-in-the-loop approach to improving Java source code quality, which focuses on “routine” code quality concerns. We specifically explore overspecific variable declarations, for which we introduce a novel analysis, and ignored exceptions (for which the analysis is straightforward).

In order to assess feasibility of our approach, we have developed several style checkers for Java programs and applied them to a 2MLOC corpus of deployed production code. Our analysis of the corpus indicates that fully 20% of Java exceptions are ignored and 60% of those lack an explanatory comment. In addition, 4% of all variable declarations overspecify the variable type (i.e., fail to abstract to an appropriate level of the inheritance hierarchy).

We consider issues of adoptability and the user experience. For example, we introduce an annotation scheme for documenting programmer intent with respect to violation of these style rules. We also describe a capability in our prototype tool to offer repairs to the programmer when quality issues are raised. This is a factor in the design of the analyses.

We call the overall approach “models of thumb” because it is designed to assist in identifying and addressing code quality criteria usually expressed in the literature as informal rules of thumb. A “model of thumb” includes a precise characterization of the particular quality attribute, scalable analyses for compliance, and rules for suggesting repairs to violation.

1 Introduction

“There may be ten ways to write code for some task $T$. Of those ten ways, seven will be awkward, inefficient, or puzzling.” — Guy L. Steele Jr. [3]

For code and low-design, rules of thumb develop that describe programming practices that have conventionally proven effective and maintainable—practices that are associated with high quality source code. While these practices are captured in books such as [17, 18, 3], and while there are tools to support checking of style and correctness, there is nonetheless limited tool support for advising programmers on improving semantic aspects of quality.

The need for tool support—that is effectively adoptable—is great. Programming teams generally work under schedule pressure and therefore focus on functionality, whose benefits are immediate, rather than quality, whose benefits are diffuse and deferred, and whose attributes appear to be subjective or difficult to measure. In addition, the explosion of API size and complexity (and corresponding documentation size and complexity) can lure busy programmers to simply “clone” example code provided with an API or found on the Internet—examples, as previous work has noted, that can contain significant problems [20].

We are exploring a scalable programmer-in-the-loop approach to improving Java source code quality that focuses on “routine” code quality concerns. These are concerns that may not be as semantically deep as overall functionality or correctness or performance, but that go beyond concrete-syntactic style or compliance with naming conventions.

In order to illustrate our approach and the adoptability considerations we address, we have developed two style checkers for Java programs and applied them to a large (2MLOC) code base of production code. In this paper, we
We call our approach “models of thumb” because it can help a lead or QA programmer identify and answer difficult code quality questions, and indeed provide some overall indication of the quality of code, at least with respect to the specific attributes analyzed. For example: Does my team’s code follow best practice? How well? Where are the problems? Similarly, individual programmers receiving notification of an identified code quality issue can expect help answering: Why is this a problem? What are possible solutions?

Generally speaking, a “model of thumb” for a particular quality attribute is based upon four key elements: (1) a precise characterization, (2) scalable analyses for compliance, (3) rules for suggesting repairs to violation, and (4) providing an appropriately collaborative user experience—respecting the programmer’s role as the final decision authority.

Note that scalability in the analyses may be achieved through the use of “mechanical” code annotations in the sense of [11, 4, 10, 9], with a potential slight impact in adoptability due to the “expression cost” incurred (as discussed in those papers).

We describe our experience with a prototype tool that implements two well accepted Java rules of thumb as “models of thumb”: overspecific variable declarations and ignored exceptions. We report empirical data and results from applying our tool on five large Java software systems. Our tool was developed as a plug-in to the Eclipse open source Java IDE. Our use of Eclipse provides useful infrastructure and a ready context for adoption, though at present it does not support the “nightly build” model.

Our principal results are (1) precise specifications for “models of thumb” for the two quality attributes mentioned above (and detailed below), (2) evidence that the supporting analyses for overspecified variable declarations and catching overly broad exception types, while semantically significant, are nonetheless tractable for large-scale Java programs without requirement for programmer annotation\(^1\), (3) empirical evidence of the extent of quality anomalies in widely used production code, (4) techniques for tools to provide concrete advice to programmers to improve code with respect to the two rules of thumb, and (5) techniques for programmers to formally document cases where there are sound reasons to violate a model of thumb in some code segment. Most significantly, we combine these elements into an overall technique that appears to be potentially both adoptable and useful.

We discovered that we needed to undertake several iterations in creating the precise definitions of the two models of thumb (based on the informally articulated rules in the literature). Once a candidate model was implemented, we then assessed the empirical data from its application to the corpus of code. On this basis, we refined the model in order to provide more useful results—i.e., fewer false positives, better solution recommendations, and better annotations for formally documenting exceptional cases.

Table 1 identifies the Java systems examined, including a short description, code size\(^2\), and wall clock time required for our prototype tool to perform its analysis\(^3\).

### Table 1. Java software systems examined for code quality.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Java kSLOC</th>
<th>Total Prototype Analysis Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jakarta Ant 1.5</td>
<td>Java-based build tool (similar to make)</td>
<td>64</td>
<td>2 min</td>
</tr>
<tr>
<td>Jakarta Tomcat 4.0.4</td>
<td>Java Servlet and Java Server Pages (JSP) web server</td>
<td>66</td>
<td>2 min</td>
</tr>
<tr>
<td>Sun J2SDK 1.4.0_01</td>
<td>Core APIs for Java 1.4</td>
<td>508</td>
<td>34 min</td>
</tr>
<tr>
<td>NetBeans 3.2.2</td>
<td>Java Integrated Development Environment (IDE)</td>
<td>571</td>
<td>60 min</td>
</tr>
<tr>
<td>Eclipse 2.0</td>
<td>Java IDE</td>
<td>792</td>
<td>121 min</td>
</tr>
</tbody>
</table>

Total: 2,001 219 min

1.1 This paper

We start by describing the code quality models, the analysis approach, and related empirical results for overspecific variable declarations and for ignored exceptions. We then consider user experience issues. We then report a surprising but straightforward finding regarding unnecessary import statements within the Java software we examined.

---

\(^1\)The analysis for ignored exceptions is trivial.

\(^2\)This does not include the extensive NetBeans test infrastructure—total NetBeans code size is 616 kSLOC

\(^3\)Our prototype tool was benchmarked on a dual-processor 1GHz Pentium III with 1GB of memory running stock Red Hat 7.3 Linux and Eclipse 2.0.1
import java.util.*;  
class ContainerFolly {  
  private ArrayList log = new ArrayList();  
  void log(String newEntry) {  
    log.add((new Date()) + "*: " + newEntry);  
  }  
  void print() {  
    ListIterator entry = log.listIterator();  
    while (entry.hasNext()) {  
      System.out.println(entry.next());  
    }  
  }  
}  

class ContainerSavvy {  
  private Collection log = new ArrayList();  
  void log(String newEntry) {  
    log.add((new Date()) + "*: " + newEntry);  
  }  
  void print() {  
    Iterator entry = log.iterator();  
    while (entry.hasNext()) {  
      System.out.println(entry.next());  
    }  
  }  
}  

Figure 1. ContainerFolly and ContainerSavvy.

2 Overspecific variable declarations

Java libraries and APIs often declare a hierarchy of public classes and interfaces. It is common for the interface hierarchy to allow selection of specific functionality while the class hierarchy allows selection of a concrete implementation. It is conventional practice for a savvy programmer to select: (1) an interface at the highest level of abstraction appropriate to the problem and (2) a concrete class with desirable implementation characteristics (e.g., performance, memory usage, etc.). By following this practice, the source code is more maintainable because the design decision about the concrete class selection is localized. An overspecific variable is a variable declared using a type that is not at the highest abstraction appropriate for its actual use. In this section we describe a code quality model used to assist a programmer with discovering where declared variable types have been overspecific and with taking corrective action.

As a concrete example, contrast the ContainerFolly implementation with the ContainerSavvy implementation listed in Figure 1. In both cases a simple log class is implemented using the standard Java container class ArrayList. Both implementations compile with no warnings and exhibit identical behavior. We assert, and believe the implementers of the Java container library would agree, that ContainerSavvy makes superior use of the library, because the ContainerSavvy code makes “better” type selections for the variables log and entry. Although the types in ContainerFolly match the concrete object types created by the program, they are overspecific based upon their actual use within the program. That is, they overconstrain the range of possible implementations, possibly (1) hindering evolution, and (2) confusing readers of the code.

2.1 Analysis

We describe the analysis used by our tool prototype by stepping through an example evolution from ContainerFolly to ContainerSavvy. We also discuss programmer interaction with this analysis. Our desired result, for each declared variable, is a set of more abstract supertypes that the declaration may be changed to without altering program behavior.

The analysis first creates a list of all variable declarations within the source code for the software component—for this example we include only ContainerFolly. Declarations using Java primitive types are ignored because they cannot be overspecific. For ContainerFolly this list includes:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>log</td>
<td>ArrayList</td>
<td>line 3</td>
</tr>
<tr>
<td>newEntry</td>
<td>String</td>
<td>line 4</td>
</tr>
<tr>
<td>entry</td>
<td>ListIterator</td>
<td>line 8</td>
</tr>
</tbody>
</table>

For each variable three sets are constructed: \( m, f, \) and \( t \) containing invoked methods, \( f \) containing accessed fields, and \( t \) containing types at which the variable is assigned. In Figure 1 variable use is underlined, field and method use are italicized, and declared types are in bold. For our example \( m, f, \) and \( t \) are:

- \( m: \) the implementers of the Java container library would agree, that ContainerSavvy makes superior use of the library, because the ContainerSavvy code makes “better” type selections for the variables log and entry. Although the types in ContainerFolly match the concrete object types created by the program, they are overspecific based upon their actual use within the program. That is, they overconstrain the range of possible implementations, possibly (1) hindering evolution, and (2) confusing readers of the code.

- \( f: \) the implementers of the Java container library would agree, that ContainerSavvy makes superior use of the library, because the ContainerSavvy code makes “better” type selections for the variables log and entry. Although the types in ContainerFolly match the concrete object types created by the program, they are overspecific based upon their actual use within the program. That is, they overconstrain the range of possible implementations, possibly (1) hindering evolution, and (2) confusing readers of the code.

- \( t: \) the implementers of the Java container library would agree, that ContainerSavvy makes superior use of the library, because the ContainerSavvy code makes “better” type selections for the variables log and entry. Although the types in ContainerFolly match the concrete object types created by the program, they are overspecific based upon their actual use within the program. That is, they overconstrain the range of possible implementations, possibly (1) hindering evolution, and (2) confusing readers of the code.
statement, this situation occurs when a variable appears in a return statement, in a method call as an actual parameter, and in an array literal expression.

As \( m, f, \) and \( t \) are constructed we record the locations within the program where each use occurs. This information is used by the user interface to communicate the supporting evidence to the programmer.

Next, a set \( a \) of ancestor types is constructed for each variable from its declared type. We define a function \( \text{ancestor()} \) that returns the set of all supertypes of the set of types passed to the function. For Java this includes all superclasses and superinterfaces.

\[
\log: \quad a = \text{ancestor}([\text{ArrayList}])
\]

\[
= \{ \text{AbstractList, AbstractCollection, Object, Serializable, RandomAccess, List, Collection, Cloneable} \}
\]

\[
\text{newEntry}: \quad a = \text{ancestor}([\text{String}])
\]

\[
= \{ \text{Object, Comparable, CharSequence, Serializable} \}
\]

\[
\text{entry}: \quad a = \text{ancestor}([\text{ListAdapter}])
\]

\[
= \{ \text{Iterator} \}
\]

The set \( a \) gives us all the potential compatible types for a variable, however this set must be constrained by \( m, f,\) and \( t.\) First, we constrain \( a \) by \( t \) so that a type change will not introduce any undesired down-casting. We remove all types in \( t \) plus their ancestor types from \( a \) and call this set \( a_c.\)

\[
a_c = a \setminus (t \cup \text{ancestor}(t))
\]

For our example \( a_c \), becomes:

\[
\log: \quad a_c = a
\]

\[
\text{newEntry}: \quad a_c = \emptyset
\]

\[
\text{entry}: \quad a_c = a
\]

The set \( a_c \) is constrained by method and field use, recorded in \( m \) and \( f,\) to create a final set of compatible types we call \( c.\) To assist with the definition of \( c,\) we define a function \( \text{fields()} \) that returns the set of defined fields for a specified type and a function \( \text{methods()} \) that returns the set of defined methods for a specified type.

\[
c = \{ x : a_c \mid \forall y : f \bullet \forall z : m \bullet \ y \in \text{fields}(x) \land z \in \text{methods}(x) \}
\]

The definition of \( c \) may notionally be viewed as a table with field and method use \( (f \) and \( m \) as columns and the possible types \( (a_c) \) as rows. A “defined” is placed in a table position if the type at that row defines the method or field at that column. If every entry for specific a row is “defined” then that type is considered compatible and becomes a member of set \( c.\) Table 2 provides an example for the variable \( \log.\) The types marked with \( *\) are compatible and are members of \( c.\) Hence, for our example we find:

\[
\log: \quad c = \{ \text{AbstractList, List} \}
\]

\[
\text{newEntry}: \quad c = \emptyset
\]

\[
\text{entry}: \quad c = \{ \text{Iterator} \}
\]

Our analysis is now able to present to the programmer that the type of \( \log \) may be changed to either \( \text{AbstractList} \) or \( \text{List} \) and that the type of \( \text{entry} \) may be changed to \( \text{Iterator}.\) Because \( \text{newEntry} \)’s result is the empty set, its type must remained unchanged.

Our analysis has not changed \( \text{ContainerFolly} \) into \( \text{ContainerSavvy}:\) specifically we did not find that the programmer should use the \( \text{Collection} \) interface for \( \log \) rather than the identified List interface. This is because of the programmer-specified call to \( \text{listIterator}() \) at line 8 which is not available in the \( \text{Collection} \) interface. We have, however, improved the the implementation of \( \text{ContainerFolly}.\) If the programmer implements our suggestions, it might be noticed that line 13 now appears strange:

\[
\text{Iterator entry} = \log.\text{listIterator}();
\]

Investigation would lead the programmer to discover that \( \text{List} \) has a method called \( \text{iterator}() \) returning a \( \text{Iterator} \) object rather than a ListIterator object. With this change in place, our analysis would find \( \text{Collection} \) to be a compatible type for the variable \( \log \) and report such to the programmer, thus concluding our evolution from \( \text{ContainerFolly} \) to \( \text{ContainerSavvy}.\)

### 2.2 Annotations

A limitation of our analysis is that it is unable to recommend, in most cases, one specific “best” type. This limitation exists because (1) there is a practical tension between abstraction and understandability—The most abstract type may obfuscate the code, and (2) the results may be ambiguous—which interface from multiple possible inheritance paths is optimal or is the most abstract superclass the right choice? To enable a “best” type recommendation we propose a formal annotation which allows the class author to specify what supertypes are ideal for variables referencing objects of that class. This annotation is placed in the Javadoc comment for a class definition. Here is a reasonable annotation for the ArrayList class:

```java
/** @typerecommendation Collection, List */
public class ArrayList extends AbstractList
    implements List, RandomAccess, Cloneable,
    java.io.Serializable {...}
```
### Table 3. Overspecific variable declaration empirical results.

<table>
<thead>
<tr>
<th>Name</th>
<th>Variable Decl. Uses (u)</th>
<th>Heuristically Removed</th>
<th>Reported</th>
<th># with Inter.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total (t)</td>
<td>#</td>
<td>%t</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>Throwables Subclass</td>
<td>String Variable</td>
<td>Only Object</td>
<td></td>
</tr>
<tr>
<td>Ant</td>
<td>13,953</td>
<td>1,892</td>
<td>13.6</td>
<td>855</td>
</tr>
<tr>
<td>Tomcat</td>
<td>13,970</td>
<td>1,953</td>
<td>14.0</td>
<td>890</td>
</tr>
<tr>
<td>J2SDK</td>
<td>116,397</td>
<td>9,074</td>
<td>7.8</td>
<td>3,068</td>
</tr>
<tr>
<td>NetBeans</td>
<td>99,201</td>
<td>13,689</td>
<td>13.8</td>
<td>4,676</td>
</tr>
<tr>
<td>Eclipse</td>
<td>178,872</td>
<td>17,178</td>
<td>9.6</td>
<td>5,670</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>422,393</td>
<td>43,786</td>
<td>10.4</td>
<td>15,159</td>
</tr>
<tr>
<td></td>
<td>443</td>
<td>22.9</td>
<td>6.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>485</td>
<td>24.8</td>
<td>7.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3,650</td>
<td>40.2</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5,851</td>
<td>42.7</td>
<td>10.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8,325</td>
<td>48.5</td>
<td>10.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18,745</td>
<td>42.8</td>
<td>9.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5,647</td>
<td>11.051</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The @typerecommendation annotation contains an ordered list of supertypes, best to worst, that are recommended for variables referencing the class. For ArrayList objects, Collection is annotated as the best type, with List also recommended if Collection is not possible (due to variable use). The annotation is intended to be complete in the sense that if neither Collection nor List are allowed, then ArrayList is to be used directly as the variable type. If a class is designed to always use itself as the variable type this can be annotated via a @typerecommendation annotation with no types following it. This annotation directly conveys class author design intent which we incorporate into our model to define the best practice against which compliance is assured. Note that the information conveyed by the @typerecommendation can be formatted and displayed in the class JavaDoc, making it useful for documentation as well.

To allow a programmer to communicate that a specific type is needed for a variable in the case where our model disagrees, a second annotation is introduced:

```java
/*@spec*/ ArrayList fooList = new ArrayList();
```

The @spec annotation communicates that a variable declaration is specifically required, it effectively “overrides” our model. This annotation can also be used at the package or project level to specify that specific types can be overspecific. In this case, the annotation is placed within the programmer provided JavaDoc summary for the package (package.html) or the project (overview-summary.html) and its effect is limited to the implied scope respectively. Because these files are HTML rather than Java source code the comment context of the annotation is different, for example:

```html
<!-- @spec ArrayList,LinkedList -->
<BODY> This package ... </BODY>
```

The above annotation, within a package.html file, specifies that within this package the types ArrayList and LinkedList can be overspecific. Use of the package or project level annotation avoids tedious annotation of a large group of code which has some reason (e.g., legacy code, subcontractor software, etc.) to not comply with our model.

### 2.3 Empirical results

Use of the overspecific variable declarations model within our prototype tool on the five Java systems investigated found 4% of all variable declarations overspecify their type. More detailed empirical results for this code quality model are reported in Table 3. The second column of the table provides a count of the total number of variable declarations found within source code, providing an upper bound on how many could possibly be overspecific. Analysis results are reported starting with the total number of overspecific variables found, followed by the number of these heuristically removed, and finally the number reported via the user interface. The percentages reported are of the total and the “/kSLOC” column reports occurrence rate, on average, per one thousand lines of Java source code.

#### 2.3.1 Effective heuristics

Experience with our prototype caused us to add three heuristics to our model that removed over half the results found by our original analysis and one which identified a “questionable” use of Java interfaces and removed it from our model’s recommendations. The number of results that would have been reported, but instead were heuristically removed, and their percentage of the number of results originally found is reported in Figure 3 for each heuristic. These heuristics, combined with use of our formal annotations, allow removal of false positive occurrences.

**Throwable Subclass:** Early trials with our prototype immediately uncovered a conflict between our code quality model and the semantics of exception types as defined within the Java language. Replacing the type of an exception type, defined as java.lang.Exception and its
subclasses, with a more abstract type significantly changes program behavior. Hence, our prototype removes any variable with a type java.lang.Thrower or its subclasses from consideration by our analysis.

**String Variables:** Our prototype often recommended replacing the String type with one of its interfaces. Such a substitution, while technically legal, detracts from understandability of the source code, so we remove these recommendations from the results. In addition, the String class has special syntax within the Java language which its interfaces are not allowed to use—recommending use of an interface for a String variable denies maintenance programmers use of this syntax.

**Only Object:** A surprising number of recommendations suggested Object, the root type of all Java classes, as the sole compatible type replacement. This recommendation, as in the case of String, detracts from understandability of the source code, so we also remove these recommendations from the results.

**Constant Interfaces:** A technique for declaring a group of constant values in Java involves creating an interface containing these values, often referred to as a constant interface, and “implementing” this interface by classes that use the constants. This type of interface is always a bad recommendation for our model to make—it is not really a type, hence we ignore any interface we discover that does not define at least one method. This heuristic only effects recommendations, so it is not reported in Table 3, however from Ant it removed 4 constant interfaces that would have been suggested for 658 variables, from Tomcat it removed 5 constant interfaces that would have been suggested for 855 variables, from J2SDK it removed 1 constant interface that would have been suggested for 1 variable, from NetBeans it removed 13 constant interfaces that would have been suggested for 4,343 variables, and from Eclipse it removed 4 constant interfaces that would have been suggested for 4,527 variables.

### 2.3.2 Effectiveness

All changes recommended by the overspecific variable declarations model are sound in the sense that, if implemented, they will compile and not change program behavior. Because it is our goal to ensure best practice, this measure is insufficient. Our measure of success, for this model, is that the recommendations improve maintainability by better use of abstraction without detracting from source code understandability. We believe, based upon informal analysis of the reported results, the number of false positives (in the sense that increased abstraction would detract from code understandability) is low. But, as a small caveat, we are not experts in the type hierarchy design of the Java systems examined. Within the type hierarchies we are knowl-

**Figure 2. An annotated ignored exception.**

```
public class Level implements java.io.Serializable {
    private static java.util.ArrayList known = new java.util.ArrayList();
    private static java.util.LinkedList known = FactoryObj.getInstance();
    try {
        get();
    } catch (AnyException e) {
        //ignore, we will return null
        return result;
    }
}
```

Another case, found within Tomcat, is highlighted in Figure 3.

### 3 Ignored exceptions

The multitude of exceptions and the hierarchy of exceptions thrown by various Java libraries can be overwhelming to any programmer. In this section we describe a code quality model to assist programmers with identifying and correcting instances of a particular questionable approach to exception handling. We target ignored exceptions within Java code, where an exception is caught within the code but no action is taken, specifically code similar to the following:

```
try { ... } catch (AnyException e) {}  
```

Ignored exceptions were used by Hissem et al. in [14] as a concrete indication of open source Java source code quality. In addition, Bloch in [3] strongly advises not ignoring exceptions, and further urges that in rare cases where it is appropriate, “the catch block should contain a comment explaining why it is appropriate to ignore the exception.”

4 In Ant, 70 in Tomcat, 87 in J2SDK, 633 in NetBeans, and 580 in Eclipse

5 The private field known called add() twice, size() four times, and get() four times and should be declared as type List
The analysis to detect ignored exceptions is utterly straightforward: We examine the abstract syntax tree of each Java compilation unit until we encounter empty catch statements. Note we must take care in defining “empty”—there are numerous cases in the corpus where catch blocks contain a single “;”.

### 3.1 Annotations

We provide a formal annotation to express that a specific exception is being deliberately ignored. The annotation `//@ignore` within an empty catch block identifies an intended ignored exception, an example is listed in Figure 2.

We demonstrate our ruthless empiricism by selecting a form for the formal annotation on the basis of experience, listed below. The top three informal comments within empty catch blocks are “ignore,” “do nothing,” and “swallow.”

### 3.2 Empirical results

Use of the ignored exceptions model within our prototype on the five Java systems investigated found 20% of all caught exceptions are ignored and 60% of these contain no comment explaining why. More detailed empirical results for this code quality model are reported in Table 4. The second column provides a count of the total number of catch blocks found within source code, providing an upper bound on how many could possibly be empty. Analysis results are reported in three parts. First, the total count of ignored exceptions is reported with what percentage of overall number of exceptions it represents. Second, ignored exceptions are separated into those that have informal annotations, or `iAnnotations`—defined as “ignore,” “do nothing,” and “swallow,” and those that do not. The percentages reported are of the total number of ignored exceptions. The “/kSLOC” column reports occurrence rate, on average, per one thousand lines of Java source code.

#### 3.2.1 Effective heuristics

Experience with our prototype caused us to incorporate informal annotations by accepting the presence of “ignore,” “do nothing,” or “swallow” within any comment inside a catch block as design intent to deliberately ignore that exception. Allowing informal annotation allows heuristic removal of a large number of false positive results by the analysis. Nevertheless, the formal annotation is superior because it avoids potential false negative cases (although we noted none).

Accepting any comment found within an ignored exception as an indication of design intent turns out to be a rather bad heuristic. It misses false negative cases that document further work is required or that a programmer didn’t know how to deal with the caught exception. Some typical examples are:

```java
J2SDK: javax.swing.text.html.HTMLDocument
   // Should handle this better
NetBeans: org.netbeans.core.execution.SysIn
   // TODO
   org.netbeans.modules.httpserver.HttpServerModule
   // pending - why do I get SecurityException?
   Tomcat: org.apache.jasper.compiler.XmlOutputter
   // Can never happen? I assume all platforms
   // support UTF-8
```

#### 3.2.2 Effectiveness

We estimate that roughly 90% of the issues reported by the ignored exceptions model are false positives from the perspective of program correctness—they follow a pattern we call `default-try-ignore`. From the quality perspective, almost all require a comment explaining why the exception is
The default-try-ignore pattern is illustrated in the example below:

```java
someVariable = null; // default value
try {
    someVariable = someMethodCall();
} catch (SomeException e) {}
```

A default value is set for `someVariable`, then a `try` block is entered and a more specific assignment to `someVariable` is attempted via a method call. If this attempt fails the exception thrown is ignored and the variable continues with the default value it was given. The example listed in Figure 2 conforms to this pattern and is annotated correctly.

Although not part of our current prototype, the default-try-ignore pattern has mechanical properties we are using to create a "nested" code quality model. Enforcing this model would detect the default-try-ignore pattern and recommend the programmer insert the annotation "//@ignore, default value for `someVariable` will be used." Where the ignored exception model works to detect a possible code quality problem, the default-try-ignore model works to suggest a reasonable resolution—a helpful synergy.

The cases identified by our prototype which we believe are actual program errors typically involved a bad variant of the default-try-ignore pattern where the exception ignored was overly broad. Catching an overly broad exception refers to a catch statement that claims to handle a very general exception, but does not, at least not completely. An example from Ant illustrates this problem:

```
ANT: org.apache.tools.ant.taskdefs.compilers.DefaultCompilerAdapter
try {
    out.close();
} catch (Throwable t) {}
```

The variable `out` is a `PrintWriter` which is declared to possibly throw an `IOException`. Catching `Throwable`, the root of all Java exception classes, in this case is overly broad—`IOException` should be caught. This case is very likely a program bug.

Although not part of our current prototype, we are building a code quality model of catching an overly broad exception (which may have application independent of the ignored exceptions model). The analysis for this model can examine the statements within a `try` block and construct the set of possibly thrown exceptions based upon the signatures of invoked methods. This set can then be presented to the programmer to aid analysis of any ignored exception. Furthermore, we can compare this set to the excep-
tion caught within the empty exception handler and inform the programmer if the caught exception is not within the set. In addition, a useful special case exists: when the set contains only one exception and the try block only one catch statement. If the exception in the set and the caught exception do not match the analysis can suggest changing them to match. This special case would apply in the Ant example above.

4 User presentation

A screenshot of our prototype system within the Eclipse Java IDE is shown in Figure 3. The large bottom-right pane, labeled Code Quality Advice is the user interface for our prototype. The surrounding panes (made smaller than in typical use) are elements of the Eclipse Java IDE. Figure 3 is reporting that the variable locales, within the RequestBase class of Jakarta Tomcat 4.0.4, could be changed from an ArrayList to a List or an AbstractList (Tomcat contains no annotations to further direct type choice). All supporting information, including the variable’s method use and use as an actual parameter is clearly displayed (including total count and source code locations for each and every use) to assist programmer analysis of this quality recommendation. Source code references, such as the two locations where the clear() method is invoked, are able to focus the editor to the referenced location on demand.

Our initial approach was to output our quality advice in a form similar to compiler error messages. This approach was abandoned after some experience with the prototype. Any tool that immediately dumps “advice” in the triple digits, with no rationale or categorization quickly becomes annoying. Furthermore, expecting a busy programmer to slog through an en masse dump of hundreds of possible issues does not facilitate adoption. We came to the realization that our user interface is really listing advice, not errors, and must include detailed rationale. In addition, different programmers have different roles and interests within a project. One programmer may only be interested in a focused portion of the code, a second may have global interests about use of a specific API, while a third with QA responsibilities may have interest in aggregate standards enforcement across the entire source code. This issue brought out the need for flexible categorization of our quality advice under user control. Our prototype addresses these needs in the following ways:

1. A separate Eclipse window is used to report our potential code quality issues. This can be seen in Figure 3 as the Code Quality Advice tab along the bottom located alongside the standard Tasks tab used within Eclipse to report compiler error messages and other problems.

2. Code quality advice is organized into multiple hierarchies to allow programmers to reorder and recategorize the results according to their interest. Our prototype allows categorizing by Java package and class, by type of quality issue, and by Java type the quality issue relates to.

Our prototype tool’s user interface requires further tuning. Further categorization flexibility is required, including programmer selected filtering within categories (e.g., show only issues within a specified group of packages by Java type). In addition, while the tree view has proved useful and flexible, other approaches need to be investigated. For example, views capable of displaying many short code snippets may be more effective in communicating rationale.

5 A footnote on unnecessary imports

In addition to the two models of thumb described in previous sections, we also investigated the frequency of unnecessary Java import statements—i.e., imports of packages or classes into that are not required by a particular compilation unit. (Sadly, such imports create dependencies between the unnecessary packages or classes and the compilation unit.) While the analysis is utterly straightforward, the empirical results are nonetheless interesting. The counts of unnecessary imports are listed below, which demonstrates the potential value of tool support in detecting such routine anomalies.

<table>
<thead>
<tr>
<th>Name</th>
<th>import Uses (u)</th>
<th>Unused Imports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>3,526</td>
<td>172</td>
</tr>
<tr>
<td>Tomcat</td>
<td>4,275</td>
<td>966</td>
</tr>
<tr>
<td>J2SDK</td>
<td>15,101</td>
<td>3,216</td>
</tr>
<tr>
<td>NetBeans</td>
<td>30,102</td>
<td>6,626</td>
</tr>
<tr>
<td>Eclipse</td>
<td>49,097</td>
<td>2,859</td>
</tr>
<tr>
<td>Total:</td>
<td>102,101</td>
<td>13,839</td>
</tr>
</tbody>
</table>

6 Related work

The source code improvement models developed in this paper are based on well documented and understood best practices for Java software development, which are rooted in well-established software engineering principles [19, 16]. Bloch promotes them in [3] as “Refer to objects by their interfaces” and “Don’t ignore exceptions.” Vermeulen et al. promote them in [21] as “Maximize abstraction to maximize stability” and “Do not silently absorb a run-time or error exception.”

Our interest in, and investigation of, open source Java projects stems from earlier work which investigated quality practices of many successful open source projects and attributes for adoptable quality-related interventions [13].
We have also considered issues related to the use of annotations to capture design intent, particularly related to concurrency [4, 11, 20].

Like Extended Static Checking for Java (ESC/Java) [10, 6, 15], our approach uses annotations to capture design intent. Our annotations and analysis focus on flagging code quality issues and, unlike ESC/Java, make no attempt to find bugs within a program or give any assurance of functional correctness. The SLAM project [2] focuses on verifying that C device driver code obeys API usage rules. SLAM employs model checking, static analysis, and theorem proving. SLAM works to a greater semantic depth, but is focused on a more narrow domain.

The commercial JTest product by Parasoft can use static analysis to enforce 300 “coding standards” rules [1], mostly accomplished through direct analysis of abstract syntax trees. It includes a rule to detect ignored exceptions, but does not detect overspecified variable declarations except in the particular case of List and Set variables.

The Meta-level Compilation (MC) project [7, 12] has been very successful finding bugs within large C programs [8, 5]. This project has focused on finding bugs within C software, we believe the metal language developed as part of this project to formally encode erroneous program behavior could be used to improve code quality. Our approach to annotations in this paper is similar to that of the MC project, in that in both cases annotations are used primarily for false positive control.

7 Conclusion and future work

This paper presents a programmer-in-the-loop approach to assure Java source code with respect to precise models of accepted programming rules of thumb. We have used the code quality models implemented in our prototype to collect empirical data from a 2MLOC corpus in order to inform our assessment of the significance and value of quality checking.

Our approach is motivated by the desire to improve and assure source code quality. We are working to extend our current work by developing additional quality models and analyses. We are addressing issues related to the design of the user experience. We are also linking this work with our more invasive analyses [4, 11, 20] related to concurrency and API compliance.

8 Acknowledgements

Special Thanks to Aaron Greenhouse, Jim Herbsleb, and George Fairbanks. From the Eclipse team thanks to Adam Kiezum for comments on this work, Olivier Thomann for fixing several bugs in the Eclipse Java core, and all the Eclipse developers for their outstanding work.

References


