Bug Detection through Static Analysis

Compiler Project for EDAN70 - Project in Computer Science at Lund University, Faculty of Technology

Ella Eriksson
D10, Lund Institute of Technology, Sweden
ada10eer@student.lu.se

Zimon Kuhs
D11, Lund Institute of Technology, Sweden
eng08zku@student.lu.se

Abstract

This paper describes a static code analyser for Java code implemented with the ExtendJ compiler and JastAdd, a Reference Attribute Grammar (RAG) system for Java. Our tool implements some of the analyses implemented by another static analysis tool called ErrorProne. We investigate the efficacy of using aspect-oriented programming with attribute grammars compared to traditional Java compiler coding patterns, like visitors, and what advantages and disadvantages compiler implementation with a declarative coding approach might have over a traditional, imperative approach. We found that our static analysis tool implements comparable analyses using fewer lines of code.

Keywords Attribute Grammars, Bug Detection, ErrorProne, ExtendJ, JastAdd, Static Analysis.

1. Introduction

Static program analysis is used to find bugs or other flaws in source code by detecting undesired code patterns. Static analysis can help ensure that the best coding patterns are followed.

In this project we present a static analyser called the eXtended Analyser for Java Code, or XAJC. XAJC presents warnings and error messages for Java code. XAJC is an extension to ExtendJ, an extensible Java compiler. ExtendJ is implemented in JastAdd, an aspect-oriented system for compiler development using reference attribute grammars and declarative programming.

We have implemented a subset of the analyses in ErrorProne [1], and evaluated the difference between ErrorProne and XAJC both in terms of code complexity and in performance. We also investigated the difference between constructing a static analyser using declarative and procedural coding.

Our analyser is not intended to find all bugs in Java code. As is the case for ErrorProne, XAJC only performs simpler types of analysis. By only using static approaches to evaluate code, and thereby not evaluating *executed* code, the aim is to locate sections in source code where design patterns [2] are not followed. If best design patterns are not followed, it might be difficult to modify or understand, or the program might produce unforeseen errors [3].

2. Background

2.1 Attribute grammars

Attribute grammars have several beneficial properties for compiler construction. Born in the mid-60s by Donald E. Knuth, attribute grammars was the start to process program language semantics without using intricate algorithms [4]. They are declarative, meaning that the code describes what should be done instead of how it should be done, leaving the details to the compiler. Attribute grammars are thereJastAdd is aspect-oriented, meaning it supports cross-cutting concerns that normally cannot be refactored into a separate module. It also means that it is possible to add new features to the code, without changing parts of the old code.

The attributes are evaluated on demand or as needed. This is also referred to as lazy evaluation. There are two types of attributes; synthesized attributes and inherited attributes.

JastAdd is a modular system for coding which uses Reference Attribute Grammars (RAGs). With RAGs it uses attribute grammars, and further allows attributes to have both reference values and parameters. The inclusion of reference values means that an attribute can be a reference to another node in the AST. If an attribute has both references and parameters, it allows the node to access information in a distant AST node. ExtendJ is a compiler written on JastAdd code.

2.2 ErrorProne Bug Patterns

ErrorProne is an analyser that focuses on catching common made Java coding mistakes. It is open-source and maintained by Google. ErrorProne uses the javac compiler while searching for bugs by extending the JavacCompiler class. The different warnings and errors that ErrorProne catches are called Bug Patterns. It does so by adding custom error checks after the flow phase of the default compiler. It uses BinaryTrees to visit each of the AST nodes. With each AST node, it runs the intended bug matching code for that node type. ErrorProne further offer automated code revisions which they name fix suggestions. We chose six different bug patterns to implement using JastAdd and ExtendJ. The patterns we chose are ComparisonOutOfRange, DependencyAnnotation, EmptyIf, ClassNames, StringEquality and WaitNotInLoop. We also chose to utilize the test cases already created by ErrorProne, in order to keep correctness analysis as consistent as possible for the two tools.

3. Implementation

Our tool is an extension to ExtendJ's implementation of new JastAdd aspects (.jrag files). The different analysers to be matched are dealt with by one JastAdd aspect respectively. The patterns are matched in the same manner code-wise; a *contribution* from AST members of the type or types that are responsible for detecting that a certain analyser exists. An example can be seen in fig. 1.

A `MethodAccess` node can contain a `contribution` attribute which reports a bug where the `wait` function is not in a loop (matched by the attribute `waitNotInLoop()` in fig. 1).

When a match is found, a `contribution` from the responsible node is added to the root AST node `Program`'s `collection` attribute, which can be seen in fig. 2.

Before XAJC terminates, it prints all of the messages in the `collection`.

```
aspect WaitNotInLoop {
    MethodAccess contributes waitNotInLoopMessage()
        when waitNotInLoop()
        to Program.errorMessages() for program();
```

Figure 1. Contribution to warning messages.

```
coll Collection<String> Program.errorMessages()
    [new LinkedList<String>()]
    with add root Program;

syn Collection<String> Program.errorStrings() =
    errorMessages();
```

Figure 2. Example warning message `contribution`.

The different bug patterns are implemented without dependency of one another meaning that it would be possible to disable other pattern detections in order to "ignore" other errors in search for a specific type of bug in the code.

3.1 Comparison out of Range

When a comparison is made between two numerical values of different numeric types there is the possibility that a value will be compared to another value outside of its numerical range, making a conditional statement effectively a literal boolean, resulting in an unnecessary conditional statement or dead code in the then-clause (fig. 3.

```
// Bug; Long.MAX_VALUE exceeds Integer.MAX_VALUE.
public boolean compareIntToLong(int a, long x) {
    return x > Long.MAX_VALUE;
}

// OK; comparison within Float's range.
public boolean compareFloatToDouble(float y) {
    return y < (double) (Float.MAX_VALUE);
}
```

Figure 3. Compaing variables with out-of-range literals.

This analysis detects such errors when at least one of the operands is a constant value[1].

The ErrorProne implementation needs to manage the `Byte` and `Character` cases differently, resulting in some extra code. It should be noted that this implementation only considers equality and non-equality. It does not cover other kinds of comparisons but extending the implementation to handle them would be trivial since the numeric check is already in place. Only inclusion of other operands (greater than, greater than or equal to...) would be needed.

XAJC's implementation needs some code to know the number ranges in addition to its matching logic, but the actual matching is performed more or less the same. Notably the implementation would require explicit addition for each new type to be added, but this is unlikely regarding primitive types, since their behaviour is integral to Java and thus they are rarely modified.

Both implementations need to manage floating point numbers and integral numbers types differently, having no effective way of covering comparison expressions in general since each primitive type needs to be managed explicitly (`Float.MAX_VALUE`) instead of utilizing a generic field (e.g. `((Numeral) x).MAX_VALUE()`).

[1] Analysis of a conditional involving two variables can not be done prior to code execution, and while comparing two literals is suspicious coding, the error is not pertinent to the Comparison out of Range analysis.

3.2 Dependency Annotation

This analysis detects inconsistencies in the JavaDoc tag `@deprecated` and its corresponding declarations. If the declaration of a class, constructor, field, interface or method is not preceded by the `@Deprecated` annotation when the javadoc tag specifies otherwise, e.g. as presented in fig. 4, an error is reported.

```
/**
 * N.B, method is deprecated.
 *
 * @deprecated
 */

// Bug; missing @Deprecated annotation as
// specified by the above javadoc.
public boolean method(int x) {...}
```

Figure 4. JavaDoc tag in comment but not in method annotation.

ErrorProne's implementation uses a unique visitor for each type, reducing the required code amount by having the visitor for each AST class call the same matching method. It uses the bit pattern generated by javac for javadoc comments directly as filtering method.

The JastAdd implementation follows a similar vein but uses the collection construct instead of the visitor construct, and filters javadoc comments with an `@deprecated` tag using `contains<String>("@deprecated")`.

3.3 Empty If

If an `if`-statement has no `else` clause but has no statements in the then `block` it's highly likely that the programmer has made an error. If not, it is redundant code, e.g. in fig. 5.

```
// Bug; redundant conditional statement due to
// empty if-statement.
public boolean emptyIf(int x) {
    if (x > 0)
        ;
    return 1;
}

// OK; code performs something.
public int emptyIfWithElse(int x) {
    if (x > 2)
        ;
    else
        return 3;
    return 4;
}
```

Figure 5. `If`-statements with empty `then`-branches.

ErrorProne's implementation collects if-statements and reports errors only for those devoid of else-clauses and then-statements. In the JastAdd implementation empty statements can exist in the then-clause as long as there is at least one statement which is non-empty. The reasoning behind this is that while the empty statements are redundant, the if-statement is in that case not redundant.

3.4 Class Names

The name of the source file should match the name of the top-level class it contains, for example as illustrated in figure 6, for the erroneously placed class declaration of class `B`.

```
// File A.java
public class A {...}
public class B {...}
```

Figure 6. Example of a class defined in a file with mismatching name.

In Java, it is legal [5] to define several top-level classes in the same file. E.g., a source file with the name `A.java` could have two public classes `public class A` and `public class B` at the top level. Such implementation does necessarily cause any problems, but could if a third class, `C` requires `B`, when `A` is not part of the current compilation. Then the compiler would search in `B.java` for `B` and find nothing, or another class with the same name. For the bug pattern ClassNames, a warning is generated for such kinds of class declarations. In the ErrorProne implementation, the pattern compare the source file name and names of possible public top-level classes by using visitors to search through the compilation units. When comparing the source file names and top-level class names, instead of searching for when a warning should be issued, the bug pattern eliminates the instances when the pattern should not generate a warning. If none of those checks generates a `true`, the name of the other top level class is reported.

In our implementation, one aspect is used for checking that in each ClassDecl node, the name of the source file is compared to the class declaration name. If the names do not match, the pattern generates a warning.

3.5 String Equality

A common mistake in Java is to compare strings by using either `==` or `!=`. When such comparison is made the strings are compared by reference equality (or non-equality) instead of value equality. When such a comparison is made, as can be seen in fig. 7., the `StringEquality` analysis generates a warning.

```
public boolean compare(String x, String y){
    boolean retValue = (x == y);
    retValue.toString();
    return retvalue;
}
```

Figure 7. Incorrect string comparison using ==.

The two strings should in this case be compared using the `String` class' `.equals()` method as in fig. 8.

The implementation of this bug pattern in ErrorProne uses two public methods and four private methods. This could be the reason why the implementation is verbose. For the two public methods, `matches()` returns true if the operand are strictly strings and `matchBinary()` searches for the cases where the operands

```
    boolean retValue = x.equals(y);
```

Figure 8. Correct string comparison using `.equals()`.

are compared by using `==` or `!=`. Both methods use vistors on `BinaryTree`. The latter method uses all of the private methods and consists of several `if`-statements.

With JastAdd it is easier to check if the operands are both Strings as the AST node implementation has a node `EqualityExpr` that is a super class to both `EQExpr` (==) and `NEExpr` (!=). Consequently in XAJC, the matching attribute `StringEquality` only needs to check if both of its operands are `String` instances. If both operands are strings, a warning is generated (fig. 9).

```
EqualityExpr contributes
    invalidStringComparisonMessage()
    when invalidStringComparison()
    to Program.errorMessages()
    for program();

syn boolean EqualityExpr.invalidStringComparison() =
    getLeftOperand().type().isString() &&
        getRightOperand().type().isString();
}
```

Figure 9. Using JastAdd to find string comparisons using reference equality.

3.6 Wait not in Loop

This pattern generates warnings if `Object.await()` or `Object.wait()` is called outside a loop. Although Java does permit such calls, calling `wait()` like this could cause problems due to *spurious wake ups*.

When a set of threads waits for another thread to release a lock or fulfill some condition, the awaited thread can wake the watching threads via the method `notifyAll()`. In that case, the waiting threads will continue running even though the condition they are waiting for has not been fulfilled (see fig. 10 for an example). Hence, they need to re-evaluate the condition, and to ensure that this happens the waiting threads need to check their repeatedly with a loop.

ErrorProne's implementation finds all method accesses for waiting using a field called "`waitMethod`", and a boolean recursive search method "`inLoop()`" as specifications for their matcher object.

In XAJC the matching is done equivalently, albeit utilizing a few `equation` attributes; a `MethodAccess` node needs to ask its parents recursively higher in the AST whether or not they are loops. If a parent is of a looping type the `MethodAccess` node is contained within it.

4. Evaluation

4.1 In-depth comparison of WaitNotInLoop as implemented in our tool and ErrorProne

ErrorProne's implementation is done in idiomatic Java following *Google's Style Guide* [6], which means that any division of source code contains a lot of imports, one for each type used that is not contained within the same package. JastAdd, which weaves together the aspect files when building, does not need to do this

```
\\ Awaited Thread:
boolean condA = false, condB = false;
void run() {
    while (true) {
        wait(1000);
        condA = true;
        notifyAll();            // (1).
        wait(1000);
        condB = true;
        notifyAll();            // (2).
   }
}

// Waiting thread with loop:
void runGood(Thread master) {
    while (!master.condB) {     // Condition checked
                                // until condB
                                // is true.
        wait();
    }
    performFunction();          // Will be reached
                                // after (2).
}

// Waiting thread without loop:
void runBad(Thread master) {
    if (!master.condB) {        // Condition checked
                                // once.
        wait();
    }
    performFunction();          // Will be reached
                                // after (1).
}
```

Figure 10. Correct and incorrect usages of `wait()`.

explicitly. The point is minor but results in a lot more lines of code compared to our implementation, why imports are ignored when counting lines of code.

4.1.1 Error reporting

The structure of our implementation utilizes a JastAdd `collection` of Strings at a program's top level, i.e. the `Program` AST node (section 3), for error reporting. Each aspect is responsible for the detection of a particular bug, and contains specification about when an AST node capable of "containing" a bug can detect it and add a message to the top-level collection.

The "`when WaitNotInLoop()`" expression means that the error message returned by `waitNotInLoopMessage()` will be added when the boolean returned from the `waitNotInLoop()` attribute for a given `MethodAccess` AST node instance is true. This means that the `waitNotInLoop()` method specifies our filtering criteria with which we look for pattern matches and consequently detect bugs.

Each of ErrorProne's bug pattern matcher implementations has an annotation `@BugPattern` which specifies information about the pattern, as given by fig. 11.

The ErrorProne implementation uses a constant String field (fig. 12) for the case where no match was found, and a message template for producing an error report specifically in regards to a certain ID.

If the AST, for a given Visitor (called `state` in the implementation), is currently configured as such that it contains a certain pattern (fig. 13), the `matcher` `matches()` method will find the pertinent symbol `sym`, which will be used in the message template.

```
@BugPattern(name = "WaitNotInLoop",
    summary = "Because of spurious wakeups,
    Object.wait() and Condition.await() must always
    be called in a loop",
    category = JDK, severity = WARNING,
        maturity = MATURE)
```

Figure 11. ErrorProne annotation `@BugPattern`.

```
private static final String MESSAGE_TEMPLATE =
     "Because of spurious wakeups, \%s must always
       be called in a loop";
```

Figure 12. ErrorProne message template.

```
  @Override
  public Description matchMethodInvocation(
  MethodInvocationTree tree, VisitorState state) {
    if (!matcher.matches(tree, state)) {
      return Description.NO_MATCH;
    }

    Description.Builder description =
        buildDescription(tree);
    MethodSymbol sym = ASTHelpers.getSymbol(tree);
    if (sym != null) {
      description.setMessage(String.format(
        MESSAGE_TEMPLATE, sym));
    }
```

Figure 13. ErrorProne visitor usage.

4.1.2 Matching

In order for our pattern matching to work, it is required that the "loop" nodes, i.e. the AST nodes generated after parsing a `do`, `for`, or `while`, can tell whether or not it is a loop to any inquiring child node. This information is implemented using attribute equations (fig. 14), and the `CompilationUnit`[2] node can report to every child, and their children recursively, that they are outside a loop.

```
eq  CompilationUnit.getChild().inLoop() = false;
eq  DoStmt.getChild().inLoop() = true;
eq  EnhancedForStmt.getChild().inLoop() = true;
eq  ForStmt.getChild().inLoop() = true;
eq  WhileStmt.getChild().inLoop() = true;
```

Figure 14. Equation attributes in loop nodes for informing child nodes whether or not they are within loops.

Equivalently named attributes and equations in JastAdd will override the ones specified in a higher node, so the `inLoop()`

[2] Representative of a file in a built program.

method will be true for the four type of loop nodes and their recursive children, but not for any other node.

The bug being sought after is when a method called "`await()`" or "`wait()`" is called, statically or not, outside a loop context. The use of a method generates a `MethodAccess` node, so we declare that each such node has to ask its parent nodes recursively whether or not it is in a loop, as done in fig. 15.

```
inh boolean MethodAccess.inLoop();
```

Figure 15. The inherited `inLoop()` attribute will look in parent nodes to know whether or not it is in a loop.

This inherited attribute is then investigated for each `MethodAccess` node, and when the `waitNotInLoop()` method (fig. 16) returns true, we have found a bug pattern.

```
syn boolean MethodAccess.waitNotInLoop() {
    return (name().equals("wait") || name.equals(
        "await") && !inLoop();
}
```

Figure 16. Attribute checking if the `MethodAccess` node represents a `wait()` call and is outside of a loop.

ErrorProne utilizes a `Matcher` (illustrated in fig. 17), an object collecting AST nodes dependent on some criteria. In this bug pattern the criteria are that the method should be one of `await` and `wait`, as well as not be part of a loop structure.

```
private static final Matcher<MethodInvocationTree>
    matcher = allOf(waitMethod, not(inLoop()));
```

Figure 17. ErrorProne's matcher finding `wait()`-calls outside of loops.

This works similarly to JastAdd collections, but instead of nodes contributing to a collection the matcher itself looks at the tree, a shift of view point.

4.1.3 Metric Comparison

Counting source lines of code (SLoC) for different implementations is a non-biased way of producing a comparable metric. The issue is that it says little, if anything, about the quality of the code other than maybe telling the analyst that her code might be too long. It could also be argued that a shorter implementation usually means a cleaner, simpler one, which can be indicative of the difficulty in implementation and management of the code.

Counting SLoC for ErrorProne's and XAJC's implementations for the respective bug patterns it is clear that XAJC requires less code than ErrorProne's in all cases (fig. 18).

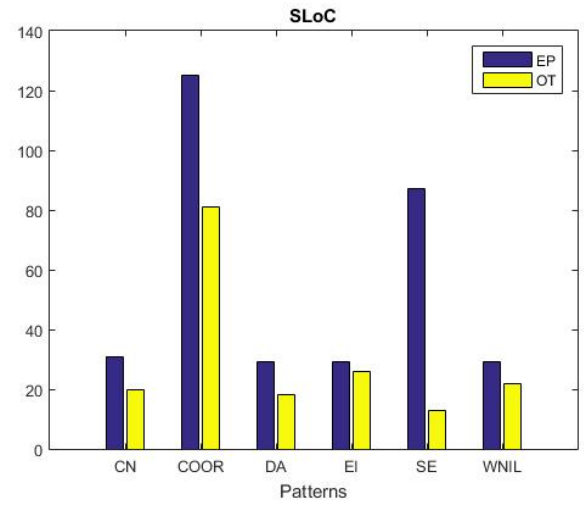

Figure 18. Source lines of code for the two tools.

Especially noteworthy is the case of String Equality; XAJC uses roughly 15% of the lines that ErrorProne does. On average, XAJC uses 62% of the code that ErrorProne does.

Another way of analysing source code objectively is to look more closely at what is actually used in the code, i.e. *tokens*. A token denotes any construct that has some form of semantic meaning within the programming language, e.g. a primitive type name, a `for`-token, or the usage of a method or variable. The advantage of this over counting SLoC is that things such as code formatting and the textual structure of the code is disregarded. Furthermore token counting avoids taking into consideration project-specific styles, such as whether or not to always use braces in `if`-branches regardless of whether or not the `then`- and `else`-blocks contain statements. It also takes into consideration e.g. long method chains; a long method chain might contain many tokens but occupy a single line only, meaning that it could be considered more complex than execution-equivalent code spread across multiple line. Indeed, it is possible for code using more lines to be more navigational and readable than shorter code.

The general trend here is as evident as when counting SLoC; XAJC utilizes fewer tokens than ErrorProne does by a significant margin (see table 19).

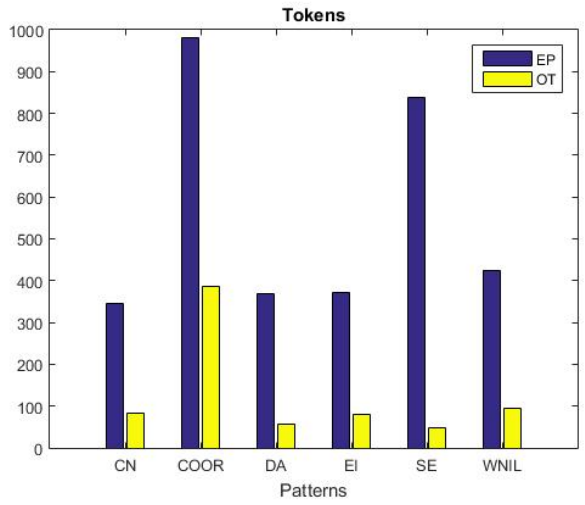

Figure 19. Java token count for the two tools.

The aforementioned case of String Equality requires only 6% of the amount of tokens in ErrorProne, and the average token amount of XAJC is approximately 22%.

It is important to note the fact that ErrorProne's implementations feature attempts at automated code revision, which inflates code length and token amounts when only considering the actual pattern matching.

4.2 Using our tool on an existing project

In order to validate the accuracy of XAJC in a real-world environment, we utilized the open-source Atlassian project JIRA [7], a tool for management of software development in agile settings. It comprises (among other things) 80 Java source code files with a total of 6850 lines in them.

Our tool did not find any matches in the project (table 1), despite its size.

Table 1. Errors found by XAJC in Atlassian's JIRA

Bug Pattern	Matches Found
Class Names	0
Comparison Out of Range	0
Dependency Annotation	0
Empty If	0
String Equality	0
Wait not in Loop	0

Running XAJC on all of the project files in *Linux Ubuntu 3.13.0-65* with the command in fig. 20. has a total execution time of approximately $9.155s$.

```
time java -jar <jar_file.jar>
    $(find testProject -name '*.java')
```

Figure 20. The Unix command used to time XAJC.

However, this execution time includes system delays and time spent waiting, why we instead look at the `real` time as specified by the `time` command. See fig. 2 for a full listing of the results.

The lack of matches can likely be explained by considering the fact that Atlassian is comprised of mostly experienced programmers using well-developed coding practices and diligent testing to produce stable and tidy code. As such, it is unlikely that the minor errors that XAJC matches can be found.

5. Related work

Static program analysis refers to the error checking of code before or without execution, usually at compile time. When it comes to code analysis, it can be used for anything from gathering metrics and data structure analysis to enforcing coding standards. Usually, static analysis tools are used by developers to check the code and can serve as an important part of software testing [8].

There are several static analysis bug detection tools available, such as FindBugs, ClangTidy and DocComments. These tools find flaws or bugs and sometimes suggests how to fix them. There are tools available for most popular programming languages, e.g. two analysers for Java are FindBugs and ErrorProne. Analysers can target either the source code or object code in order to detect unwanted patterns in the software. FindBugs analyses Java bytecode [9] and ErrorProne uses the javac compiler API.

Time type	XAJC Time	EP Time
Real	2.42388	0.504
User	9.15476	1.20
System	0.37242	0.08

Table 2. Timing results for XAJC and EP run on the JIRA project.

6. Conclusion

Due to the lack of aspect-oriented compiler extensions it is troublesome to make substantial comparisons between the two approaches in general. There is extensive literature on the use of static analysers, but in view of declarative contra procedural it is difficult to determine whether or not either one is decidedly advantageous for different aspects. In this report we have shown an example where the declarative approach for certain bug patterns meant a lot less code and less complexity in implementation.

Not only when it comes to ease of implementation, but also for accessibility the declarative approach that JastAdd uses provides a framework that represents abstract syntax trees in a clear, distinguished manner. This is partially due to the fact that it was created for that purpose specifically whereas Java as a language was designed with a broader perspective.

The amount of code required to build the respective implementations play importantly into this fact. The Java approach by ErrorProne requires a lot more lines and Java tokens than XAJC does, even though it could be argued that for an experienced Java programmer, the shift in programming style might require a certain learning period.

References

[1] Errorprone.info. *Error Prone*. N.p., 2015. Web. 16 Dec. 2015.

[2] E. Gamma, J. Vlissides, R. Johnson, and R. Helm, *Design Patterns*. Boston, Addison-Wesley, 1994.

[3] R. C. Martin, *Agile Software Development, Principles, Patterns, And Practices*, Prentice Hall, 2003.

[4] D. E. Knuth, *Semantics of context-free languages* Springer-Verlag, 1968.

[5] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java language specification, Java SE 8 Edition* Upper Saddle River, Addison-Wesley, 2014.

[6] Google.github.io. *Google Java Style*. N.p., 2015. Web. 13 Dec. 2015.

[7] JIRA. Atlassian, 2015.

[8] C. Sadowski et al. *Tricorder: Building a Program Analysis Ecosystem*. International Conference on Software Engineering, 2015, p598-608, 11p. Publisher: IEEE.

[9] N. Ayewah et al. *Using Static Analysis to Find Bugs*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering , 2015, p598-608, 11p. Publisher: IEEE.