Building and Using Pluggable Type-Checkers

Werner Dietl Stephanie Dietzel Michael D. Ernst Kıvanç Muşlu Todd W. Schiller
University of Washington
{wmdietl,sdietzel,mernst,kivanc,tws}@cs.washington.edu

ABSTRACT

This paper describes practical experience building and using pluggable type-checkers. A pluggable type-checker refines (strengthens) the built-in type system of a programming language. This permits programmers to detect and prevent, at compile time, defects that would otherwise have been manifested as run-time errors. The prevented defects may be generally applicable to all programs, such as null pointer dereferences. Or, an application-specific pluggable type system may be designed for a single application.

We built a series of pluggable type checkers using the Checker Framework, and evaluated them on 2 million lines of code, finding hundreds of bugs in the process. We also observed 28 first-year computer science students use a checker to eliminate null pointer errors in their course projects.

Along with describing the checkers and characterizing the bugs we found, we report the insights we had throughout the process. Overall, we found that the type checkers were easy to write, easy for novices to productively use, and effective in finding real bugs and verifying program properties, even for widely tested and used open source projects.

Categories and Subject Descriptors: D3.3 [Programming Languages]: Language Constructs and Features—data types and structures; F3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning About Programs; D1.5 [Programming Techniques]: Object-Oriented Programming
General Terms: Languages, Documentation, Verification
Keywords: intern, nonnull, enum, enumeration, fully qualified name, binary name, field descriptor, Java, annotation, bug finding, case study, pluggable type, type qualifier, type system

1. INTRODUCTION

A type checker provides a compile-time guarantee that certain errors cannot occur. For example, Java's type checker guarantees that a standard Java program cannot exit with a method-not-found exception. Unfortunately, standard type systems and checkers can't help developers find and prevent all the errors that they care about in practice. Therefore, developers often reason manually about code correctness — a daunting task, especially in the face of incomplete or inconsistent documentation.

Pluggable type-checking is one approach to addressing the limitations of a language's built-in type-checker. Developers write type qualifiers, such as `@NonNull` or `@Immutable`, that express extra information about types and serve as machine-checked documentation. The type-checker — a plug-in to the compiler — verifies their correctness or indicates potential errors in the program.

Despite their potential advantages, pluggable type systems are only beginning to be used outside of research. For example, Eiffel includes non-null and nullable ("attached" and "detached") types [5], but offers no way for users to refine the type system. Tools such as the Checker Framework [4] aid in the construction of these type-checking plug-ins.

The lack of uptake of pluggable type systems may stem from developers' beliefs about the cost-benefit tradeoff:

- *Building* custom type-checker plugins requires PhD-level understanding and extensive implementation effort.
- *Learning* the pluggable type-checking mind-set requires extensive training in a foreign way of thinking — a high upfront cost before any benefit can be gained.
- Even after learning, *using* a pluggable type-checker requires large amounts of time and many written annotations, cluttering code and reducing readability.
- Testing finds all important bugs, so (pluggable) type-checking wouldn't have a significant *benefit* even if it were easy to use.

We wished to investigate these beliefs, providing concrete evidence, rather than hearsay, to guide practitioners. Previous pluggable type system case studies are limited in scope and often showcased specific framework features rather than considering practicality or utility. We set out to fill in the knowledge gap by exploring pluggable type checkers in three ways: (1) using existing checkers to find interning and nullness errors, (2) writing new checkers and using them to find signature and enumeration errors, and (3) observing novice computer scientists using (and misusing) a nullness checker.

Like any case studies, ours shed light on one specific set of tools, tasks, and users. Within this context, it yielded results that contradict some of the gloomy predictions of skeptics:

- It it easy to *build* a new pluggable type-checker. For example, we found 14 errors in the Oracle JDK using a checker that was written entirely declaratively, with no code or rules of its own. Later, we added a few lines of code to eliminate many false positive checker warnings. Our toolset, the Checker Framework [4], offers a range of expressiveness that allows a designer to find the "sweet spot" for the task at hand.
- It is easy to *learn* how to use a pluggable type-checker. In a case study, inexperienced users started using the basic features of a checker quickly, obtaining immediate benefits. Although the novices did not use all the features optimally, their usage improved with time.
- It is easy to *write* annotations and few annotations are required — vastly fewer than required by Java's generic types, for example. The type-checker quickly leads a programmer to problems in the code. Occasionally, however, the annotation process becomes bogged down when the programmer is trying to understand a badly designed or undocumented program or fixing a hard bug.

This research was supported by NSF grant CNS-0855252 and Google. We thank M. Ali for development and Checker Framework users for feedback.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

These are inherently difficult processes that we do not expect to be fast. If the programmer does not want to fix the problem yet, he or she can suppress the checker warning and return to it later.
- Pluggable type-checking reveals important latent bugs even in well-tested code that is in active use. Fixing these bugs improves the program design and prevents future problems.

In addition to reinforcing the above observations, we make several other contributions. We built new checkers and extended existing checkers, making them available for public use. Using these checkers, we performed the largest study of pluggable type-checking of which we are aware. Based on the checking process, we improved real-world open source systems by finding and reporting bugs, improving their documentation and design. Finally, we discuss insights about pluggable type-checking itself.

This paper is organized as follows. Section 2 describes the Checker Framework. Section 3 presents a general methodology for using the framework, which we followed during the case studies. Sections 4–8 report the results of our studies that use the Compiler Message Checker, Fake Enumeration Checker, Signature String Checker, Interning Checker, and Nullness Checker, respectively. The sequence of checkers roughly progresses from simple yet limited checkers to more involved and expressive checkers. Section 8 additionally describes the experience of 28 first-year computer science students using the nullness checker. Section 9 discusses lessons learned, Section 10 discusses related work, and Section 11 concludes.

2. THE CHECKER FRAMEWORK

The pluggable type checkers used in our case studies were built using the Checker Framework [4, 25]. A type system designer defines new type qualifiers and their semantics in a declarative and/or procedural manner, and the Checker Framework creates a type checker in the form of a compiler plug-in. To use the type checker, a programmer can write type qualifiers throughout a program as machine-checked documentation that the plug-in verifies. The framework is compatible with Java, so it handles pluggable type systems that refine, not incompatibly change, Java's built-in type system.

The Checker Framework is integrated with build tools such as Ant, Maven, and IDEs. In particular, a Google Summer of Code project created a plug-in for Eclipse that makes it easy to run one or more type checkers on (parts of) a project.

The Checker Framework distribution comes with annotations for the most commonly-used parts of the JDK. The Checker Framework is integrated with external tools for inferring annotations, such as Julia, Nit, and JastAdd, via a common file format.

The Checker Framework's design makes it easy to create a new checker, declaratively for simple checkers or first drafts, and using procedural code for more complex logic. The distribution includes a dozen checkers, of which this paper only mentions five. Many other checkers have been written by users.

The Checker Framework optionally takes advantage of forthcoming Java syntax for type annotations (known by its Oracle codename "JSR 308" [9]); however, it is backward-compatible with Java 5 or later and supports annotations in comments. To ease the transition from other tools, the Checker Framework recognizes and processes the annotations of tools such as IntelliJ IDEA, FindBugs, JML, JSR305, and NetBeans.

Surprisingly many problems can be reduced to type checking. Examples include checking the correct initialization of fields, checking the encapsulation of objects, and checking usage protocols for methods. However, using a type checker, and therefore the Checker Framework, is not the correct solution for every problem. For a checker to work well, the types being checked can depend on local dataflow properties, but should not depend non-trivially on run-time values. In some situations, refining the underlying type hierarchy might provide a better solution; in other cases, no type system might be able to capture the property in a natural way.

3. METHODOLOGY

Suppose that a developer notices a potential problem, or a pattern of observed problems, and wishes to eliminate all such errors in the future. The developer could use pluggable type-checking to improve his or her code, following this general methodology:

1. The developer chooses an existing checker that can verify the absence of the error. If no such checker exists, the developer creates a new checker. The developer starts with a simple declarative checker that is easy to create because it consists solely of the definitions of the annotations, with no additional logic required.
2. The developer selects part or all of the program to check. The Checker Framework is effective even when used on a part of a program, and the developer might choose the part that is most error-prone, most confusing, most critical, or best fitted to the checker. Within the checked portion of the code, work should start with libraries and proceed to clients, because the checker requires annotations for the signatures of called methods.
3. The developer writes annotations. The developer searches the code (and any relevant libraries) for phrases related to the type system. The developer converts such informal documentation into machine-checkable documentation, by writing an annotation.
4. The developer iteratively runs the checker on the annotated code until there are no warnings, at which time the developer has verified the absence of errors. For each warning, there are 3 possibilities:
   (a) The developer finds and fixes an error in the code.
   (b) The warning is due to a missing annotation, which is caused by missing documentation. The developer adds the required annotation, which corrects the documentation error.
   (c) The warning is a false positive. The developer suppresses the warning by writing a `@SuppressWarnings` annotation.
5. If the developer notices a pattern of false positive warnings, (s)he extends the checker to automatically handle that case and removes the related suppressions from the code.

We followed the above methodology for our case studies. Since the Interning and Nullness Checkers existed before the study, we started by annotating the programs according to their documentation. For the other three checkers, we started with simple declarative checkers and extended them as needed.

When performing our case studies, we wrote whatever annotations were easy and natural to write — we made no effort to minimize the number of type annotations or `@SuppressWarnings` annotations.

Measurements. We quantify the effort to build the checker by its code size, the effort to use the checker by the number of programmer-written annotations and warnings suppressed (we did not record time spent), and the effectiveness of the checker by the number of bugs found.

Throughout the studies, we report two metrics for code size: total lines and non-comment non-blank (NCNB) lines. Both are as computed using the `sclc` tool[1].

The sections that follow present the case studies; Figure 1 shows summary statistics.

[1] `http://www.cmcrossroads.com/bradapp/clearperl/sclc-cdiff.html`

Checker	Size Files	LOC	NCNB	Subject program	Size Files	kLOC	kNCNB	ALocs	Annos	Errors Runtime	Other	False pos.
Compiler messages	3	70	40	Checker Framework	242	31	18	13765	11	8	0	2
Fake enumerations	15	489	299	Swing (OpenJDK b99)	1289	610	293	211415	879	0	2	32
				JabRef 2.6	576	117	74	47290	8	0	3	0
				GanttProject-2.0	509	69	49	37498	11	0	0	3
Signature strings	9	262	131	OpenJDK b99 (17 packages)	498	231	85	62510	103	11	3	26
				ASM 2.2.2	116	33	18	11651	168	5	0	36
				Annotation File Utils 3.1	126	17	10	7992	159	8	1	30
Interning	7	960	519	Xerces 2.10.0	824	257	138	68490	218	0	185	4
				Lucene 4.0	2587	479	296	99097	167	0	24	80
Nullness	30	4311	2556	Google Collections	378	78	49	50543	936	9	154	362
				Daikon	565	222	134	104852	2872	>90		365
				Classroom study (mean)	45	9	5	3414	57	–	–	–

Figure 1: Case study statistics. Sizes are given in files, lines, number of possible annotation locations, and number of annotations written by the programmer. Runtime errors are runtime-reproducible problems revealed by the checker. The other errors — primarily incorrect documentation — could lead to runtime problems in the future. We do not count design infelicities or code smells as errors. False positives are caused by a weakness in either the type system or the checker implementation; the programmer suppresses them by writing a `@SuppressWarnings` annotation (which is not counted in the "Annos" column). This paper uses "false positive" and "suppressed warning" synonymously.

4. COMPILER MESSAGE CHECKER

Property files and resource bundles, which both act like maps from keys to values, should only be accessed with valid keys. An access without a valid key returns either `null` or a default value, which can lead to a `NullPointerException` or hard-to-trace behavior, respectively.

We wrote a Compiler Message Checker, which verifies that compiler message keys used in the Checker Framework are declared in a property file. The subject program for this checker is the Checker Framework itself. The Checker Framework uses property files to store the human-readable strings for a compiler message key. Using keys instead of literal strings in the source code is less error-prone and enables easier testing. In other contexts, property files are used for localization.

4.1 Ease of Creation

The Compiler Message Checker was very easy to write — it is only 40 LOC NCNB long, containing no logic, only boilerplate such as `import` statements in annotation definitions. It extends the Property File Checker, which is a framework class that provides generic checking of property files; it is also used to check for correct internationalization of applications [4].

4.2 Ease of Use

It was very easy to completely annotate the Checker Framework. Checker warnings and errors are constructed by methods `failure` and `warning` in class `checkers.source.Result`, so we added `@CompilerMessageKey` annotations to the string parameters that are used for the message key. Then, a simple search for the uses of class `Result` quickly revealed the other 9 parameters that needed to be annotated.

All annotations are in the framework, not in individual checkers. Thus, a type system designer who has written a new checker can run the Compiler Message Checker without any annotation burden.

4.3 Effectiveness

We discovered three message keys that did not have a corresponding localized message: the Checker Framework was missing messages for keys `constructor.invocation.invalid` and `cast.redundant`; the Linear Checker was missing a message for key `use.unsafe`. In five other places in the Checker Framework a message key contained a typo. In all 8 cases, users would have seen the message key instead of a comprehensible message.

The two false positives are caused by a checker (used only for testing) that manually modifies the set of compiler messages and by a generic construct that is handled in a simplified way, respectively.

5. FAKE ENUMERATION CHECKER

Some Java programs use a set of `int` or `String` constants rather than a proper enumeration created by Java's `enum` construct; this pattern is commonly called *fake enumeration*. One reason is backward-compatibility. A public API that predates Java's `enum` construct may use `int` constants; the API cannot be changed, because doing so would break existing clients. For example, Java's JDK uses `int` constants in the AWT and Swing frameworks. Another reason is performance, especially in environments with limited resources. For example, the Android mobile phone platform recommends the use of fake enums when only an integer value is needed, in order to reduce code size and run time[2].

The Fenum (fake enum) Checker gives the same safety guarantees as a true enumeration type, without the space and run-time overhead. The values of a fake enum are treated as distinct from all values of the base type and from all other fake enums.

We annotated the Swing GUI library (from OpenJDK 7 build 99) and two applications: JabRef version 2.6[3], a bibliography management tool, and GanttProject-2.0[4], a project management tool.

5.1 Ease of Creation

The Fenum Checker consists of 3 checker classes and 4 annotation classes. We created 8 custom Fenum types for Swing (see Figure 2). Some constants are members of multiple Fenum types; for example, `CENTER` is in `@SwingCompassDirection`, `@SwingHorizontalOrientation`, and `@SwingVerticalOrientation`. Fenum `@SwingBoxOrientation` is a supertype of `@SwingHorizontalOrientation` and `@SwingVerticalOrientation`; there are no additional `@SwingBoxOrientation` constant definitions and all constants of the two subtypes can be used.

The checker also supports a generic, parameterized `Fenum` type, which allows the simple introduction of a new Fenum type without adding annotation classes. For example, `@Fenum("A")` and `@Fenum("B")` are two separate Fenum types. Handling this extensibility took about a third of the checker source code.

To declare a Fenum constant, a (type-incompatible) assignment of an unqualified type to the Fenum type is used and the resulting

[2] `http://developer.android.com/guide/practices/design/performance.html#avoid_enums`

[3] `http://jabref.sourceforge.net/`

[4] `http://www.ganttproject.biz/`, Mercurial snapshot from August 2, 2010.

Annotation	Swing Defs	Swing Uses	JabRef Uses	GanttProject Uses
`@AwtCursorType`	28	13	0	0
`@SwingBoxOrientation`	0	100	0	2
`@SwingCompassDirection`	10	80	2	0
`@SwingElementOrientation`	9	432	0	5
`@SwingHorizontalOrientation`	6	76	1	4
`@SwingSplitPaneOrientation`	2	27	5	0
`@SwingTextOrientation`	4	1	0	0
`@SwingVerticalOrientation`	4	39	0	0
`@FenumTop`	0	16	0	0
`@SuppressWarnings`	24	8	0	3

Figure 2: Usage statistics for fake enum annotations. Column *Defs* gives the number of constant definitions and *Uses* gives the number of uses.

warning is suppressed. For example:

```
@SuppressWarnings("fenum")
class Sizes {
    public static final @Fenum("Size") int SMALL = 1;
    public static final @Fenum("Size") int BIG = 2; }
```

A special feature for the definition of a Fenum constant could eliminate these warnings, but would make the type system more complex.

5.2 Ease of Use

Figure 2 presents the number of Fenum annotations used in our case study. For example, `@AwtCursorType` was used 28 times for the constants in `java.awt.Cursor` and the deprecated constants in `java.awt.Frame`. Only 13 method signatures and fields had to be annotated with `AwtCursorType`. JabRef and GanttProject did not manipulate cursors.

The `@FenumTop` annotation at the root of the subtyping hierarchy is used for code that mixes unqualified and qualified elements. As an example, the `defaults` field in class `BasicLookAndFeel` is an array where the even indices contain `String` keys, and the odd indices may contain Fenum constants.

In Swing, 24 `@SuppressWarnings` annotations are used for the definition of constants. Note that in class `SwingConstants` we used a single `@SuppressWarnings` to suppress all warnings from this class; therefore, the number of definitions and `@SuppressWarnings` are not equal. The other 8 suppressed warnings were all related to code that converted integer values (typically from configuration files) to constants or for configuration arrays that mix values of different enumerations in an unsafe way.

We needed three `@SuppressWarnings` in GanttProject: one for a string that is converted to a constant and two for situations where the local type inference gave too coarse a result.

5.3 Effectiveness

5.3.1 JabRef

The Fenum Checker revealed three bugs in JabRef. The JabRef developers acknowledged and fixed all three bugs[5]. None of the errors affects JabRef's run-time behavior.

The constructor for class `EntryTypeDialog` calls `JComponent.setAlignmentX` with `SwingConstants.LEFT` (= 2). The correct argument for the call is `Component.RIGHT_ALIGNMENT` (= 1.0f). Fortuitously, the implementation of `setAlignmentX` treats all values larger than 1.0f as 1.0f.

Method `FileListEditor.autoSetLinks` instantiates a progress bar:

```
new JProgressBar(JProgressBar.HORIZONTAL, types.length-1);
```

However, the JProgressBar constructor expects minimum and maximum values as the arguments. The value of constant `HORIZONTAL` is zero, which is probably the intended minimum value.

The constructor of class `FieldNameLabel` calls method `JLabel.setVerticalAlignment` using `NORTH` as argument. However, the correct argument to the call is `TOP`. Both constants have the value one, but this is not guaranteed. The Javadoc for `setVerticalAlignment` only lists `TOP`, `CENTER`, and `BOTTOM` as valid values, which we captured using the `@SwingVerticalOrientation` Fenum annotation.

5.3.2 Swing

We identified two inconsistencies in the implementation of Swing.

Class `MetalRootPaneUI` uses the literal zero where the `Cursor.DEFAULT_CURSOR` constant should have been used.

Method `ParsedSynthStyle.paintTabbedPaneTabBackground` has a formal parameter with the misleading name `direction`. We added a directional annotation, and the checker issued a warning. In the superclass, this parameter is called `tabIndex` and takes a non-enum integer. The Fenum Checker revealed that the parameter should be renamed.

In addition to the errors, we discovered these code smells:

In 107 places, -1 was used to signify that a field, which should hold a Fenum constant, was not initialized. We introduced a new constant `NOTSET` in interface `SwingConstants` with value -1, which belongs to all other enumeration types. We replaced the uses of -1 by references to the `NOTSET` constant, making the code easier to follow and removing fenum type errors in these locations.

Class `java.awt.Adjustable` and `java.awt.Scrollbar` have duplicate definitions of some constants. We annotated them as `@SwingElementOrientation`, making the intended interactions explicit.

6. SIGNATURE STRING CHECKER

Java defines three main formats for the string representation of a type within source and class files. Programmers must use different representations when writing source code, when using reflection, and when manipulating class files. Using the wrong string format leads to a run-time exception or an incorrect result.

A *fully qualified name* [16, §6.7][20, §2.7.5], such as `package.Outer.Inner`, is used in Java code and in messages to the user. A *binary name* [16, §13.1], such as `package.Outer$Inner`, is the representation of a type in its class file. A *field descriptor* [20, §4.3.2], such as `Lpackage/Outer$Inner;`, is used in a class file's constant pool, for example to refer to other types; it abbreviates primitives and arrays, and uses internal form [20, §4.2] for class names.

There are types for which all 3 string formats differ, and there are strings that are legal in all 3 formats, but no string represents the same type in all 3 formats. Fully qualified and binary names are nearly the same — they differ only for nested classes and arrays. This makes them particularly easy to misuse, and makes bugs easy to overlook during testing.

The Signature String Checker verifies that string representations of types are used correctly.

We annotated the following subject programs.

1. OpenJDK 7,[6] build 99: the standard implementation of the Java platform. We annotated packages `java.io`, `java.lang`, and `java.util`, and their subpackages. These packages represent 32% of the occurrences of the phrases "fully qualified name", "binary name", and "field descriptor" in the JDK.

[5] `https://sourceforge.net/tracker/?func=detail&aid=3083499&group_id=92314&atid=600308`

[6] `https://jdk7.dev.java.net/`

```
@TypeQualifier @SubtypeOf({Unqualified.class})
public @interface BinaryName {}

@TypeQualifier @SubtypeOf({Unqualified.class})
public @interface FullyQualifiedName {}

@TypeQualifier @SubtypeOf({Unqualified.class})
public @interface FieldDescriptor {}

@TypeQualifier @SubtypeOf({BinaryName.class,
    FieldDescriptor.class, FullyQualifiedName.class})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
public @interface SignatureBottom {}

@TypeQualifiers({Unqualified.class, BinaryName.class,
    FullyQualifiedName.class, FieldDescriptor.class,
    SignatureBottom.class})
public final class SignatureChecker
    extends BaseTypeChecker {}
```

Figure 3: Complete source code for a simple version of the Signature String Checker (except for `package` and `import` statements). The four annotation declarations define the type qualifiers and their subtyping hierarchy. The class declares the checker itself and lists all supported qualifiers. `Unqualified`, `BaseTypeChecker`, and the meta-annotations are provided by the framework.

2. ASM,[7] version 2.2.2: a Java bytecode manipulation and analysis framework.
3. The Annotation File Utilities (AFU),[8] version 3.1: programs that read annotations from, and write annotations to, `.java` and `.class` files. AFU contains a modified copy of ASM, which we omit from our counts to avoid double-counting, because Section 6.3.2 discusses ASM independently.

6.1 Ease of Creation

The Signature String Checker was very easy to write. Initially, it was written fully declaratively and consisted of just 10 classes, all of which had empty bodies (84 LOC; 56 LOC NCNB). Figure 3 shows its essence. The Signature String Checker also verifies the string representation of method signatures, but we omit this for brevity. Later, to reduce the number of false positive warnings, we added a factory class defining regular expressions that determine the type of a string literal. This addition reduced the number of suppressed warnings by 48%, 33%, and 54% for projects JDK, ASM, and AFU respectively.

6.2 Ease of Use

Overall, we wrote fewer than 1 annotation per 500 lines of code. Most false positives (66 out of 92) were due to string operations such as concatenation, substring, parsing, etc. The Signature String Checker does not reason about these operations.

6.3 Effectiveness

The Signature String Checker revealed 28 errors.

6.3.1 JDK

We found 14 documentation errors in JDK 7. All of these errors are also present in JDK 6, the current release. For 11 of the errors, we wrote a small program that obeys the documentation and that results in a crash (a thrown exception) or other incorrect behavior. For the remaining 3 errors, the documentation is inconsistent, but no incorrect behavior ensues from following the documentation. The errors are in four classes: `Class`, `ResourceBundle`, `LockInfo`, and `MonitorInfo`.

`Class` contains 2 errors. Two overloaded versions of `forName` are documented in Javadoc as taking a fully qualified name, but they actually require a binary name[9]. If a developer obeys the documentation and passes a fully qualified name that is not also a binary name, then `forName` throws an exception.

`ResourceBundle` contains 9 errors. Six overloaded versions of `getBundle` are documented as taking a fully qualified name, but in fact they require a binary name and otherwise throw an exception. In addition, there are errors in `newBundle` and `needsReload` that cause the functions to erroneously return (respectively) `false` and `null`. Finally, `toBundleName` returns an incorrect string.

`LockInfo` contains 2 errors, but these do not cause incorrect behavior. The constructor takes a parameter that is incorrectly documented as a fully qualified name, and `getClassName` is incorrectly documented as returning a fully qualified name. We are not sure of the right fix, because this documentation is inconsistent both with how `LockInfo` is invoked elsewhere in the JDK and with how the values are used within `LockInfo`. When invoked in the JDK, the actual value is always a binary name. Within `LockInfo`, the string can be arbitrary. Thus, the `LockInfo` documentation's mentions of fully qualified names should be changed either to binary names, or to no constraint.

`MonitorInfo` is a subclass of `LockInfo`. Its constructor contains the same error.

In addition to finding errors, we found 38 places in which the JDK documentation was deficient, such as missing a description of the format of a string. Of these places, 12 had public or protected visibility, and 26 were private or package-private. The pluggable type-checker made it easy to add the missing documentation and ensure that the documentation remains correct in the future.

6.3.2 ASM

We found 5 documentation errors, which cause run-time anomalies when used as documented, in ASM. The ASM developers acknowledged and fixed the bugs.

`ClassReader`'s constructor is documented to take a fully qualified name, but requires a binary name. The constructor converts the argument to a file system path, which follows the naming conventions of a binary name.

The return type of `Type.getClassName` is documented as a fully qualified name, but it is actually a binary name. The return value is obtained by converting a (properly documented) field descriptor into a binary name.

Three externally-visible programs, that users may invoke from the command line, have incorrect documentation. The input to `CheckClassAdapter.main`, `TraceClassVisitor.main`, and `ASMifierClassVisitor.main` are all documented as a "fully qualified class name or class file name". In fact, the input must be a *binary* name or a class file name.

6.3.3 Annotation File Utilities

We found 4 execution errors and 5 documentation errors in the Annotation File Utilities (AFU). All of them have since been fixed. The errors fall into 3 categories:

- Execution failures that occur only when an annotation is defined as an inner class
- Execution failures that occur only when an annotation has an element that is itself an annotation
- Documentation errors

Annotations defined as inner classes. It is legal to define an annotation inside another class, as in

```
class Outer { @interface Anno {} }
```

[7] `http://asm.ow2.org/`
[8] `http://code.google.com/p/annotation-tools/`
[9] A field descriptor, for arrays; the binary name format is not defined for arrays.

Such annotations are rare, and AFU had no such tests. The Signature String Checker found 2 bugs in AFU's handling of such annotations.

In `AnnotationDef.fromClass`, a call to `Class.getCanonicalName` should be `Class.getName`, which returns a binary name. When supplied with an annotation that is an inner class, AFU creates a class file with a malformed field descriptor. When processing the class file, JDK's `AnnotationParser.parseAnnotations` indicates that the annotation is not present, and JDK's `AnnotationParser.parseMemberValue` throws an exception.

A similar error is in `IndexFileParser.parseAnnotationDef`. A fully qualified name is parsed and passed to `AnnotationDef`'s constructor, whereas a binary name is needed.

Annotations with elements that are annotations. An annotation's element may itself be an annotation, as in [16, §9.6]:

```
@Author(@Name(first = "Joe", last = "Hacker"))
```

This type of declaration is infrequent, so use and testing failed to find the following two bugs.

Class `ClassAnnotationSceneWriter` calls ASM's `CheckMethodAdapter.checkDesc` and passes `AnnotationDef.name`, which is a binary name, but the method requires a field descriptor. We easily fixed this bug, since the converter method (`classNameToDesc`) was already annotated to take a binary name and return a field descriptor and therefore was our first choice.

The constructor of `NestedAnnotationSceneReader` takes two string arguments: `name`, a simple name, and `desc`, a field descriptor. The constructor incorrectly passes `name` to its superclass constructor, but it should pass `desc`. This causes an exception when an annotation has a parameter that is another annotation. Our pluggable type-checker differentiates the two varieties of strings and prevents this type of error.

Documentation Errors. Four documentation errors cause incorrect behavior if the client relies on the documentation. Field `AnnotationDef.name` is incorrectly documented as a fully qualified name. In fact, it is a binary name. Methods `IndexFileSpecification.parseClass` and `ClassFileWriter.insert`, and the constructor for class `InClassCriterion`, are all documented to take a fully qualified name, but each one actually requires a binary name.

Another documentation error does not cause incorrect behavior, only confusion to developers, as it is in a code comment rather than public Javadoc. Field `ReceiverCriterion.methodName` contains the comment "no return type". However, the field name is misleading, and the comment is wrong. The field contains a signature (not a name), and that signature does have a return type. In fact, `methodName` contains a method descriptor string.

7. INTERNING CHECKER

Interning is a design pattern in which a single object is used instead of multiple different, but equal, objects. Interning is also known as canonicalization or hash-consing, and it is related to the flyweight design pattern [12, 17, 1]. Interning has two benefits: it can save memory, and it can speed up testing for equality by permitting use of reference equality rather than structural equality (in Java, == vs. `equals`). Many studies have shown the benefits of interning in reduced memory footprint and improved speed; for example, 38% and 47% speedups on two SpecJVM benchmarks [21], and 10% speedup and 14% memory savings in the Eiffel compiler [27]. Interning is such an important design pattern that Java builds it in for strings, and programmers can define it for other datatypes.

Interning has compelling benefits, but code that uses both interned and non-interned values is prone to correctness errors or to missed performance opportunities. A correctness error is the use of == on non-interned values. For example:

```
Integer x = new Integer(22);
Integer y = new Integer(22);
System.out.println(x == y); // prints false!
```

A missed performance opportunity is the failure to use interning where intended. The program uses extra copies of objects, which causes the program to use more space and to run more slowly. The Interning Checker prevents both of these types of programming problems.

We ran the Interning Checker on two open-source Apache projects. Apache Xerces2 Java version 2.10.0 (henceforth, just "Xerces") is an XML parser and related utilities[10]. Apache Lucene 4.0 is a text search engine library[11].

7.1 Ease of Creation

The Interning Checker already existed at the time of our case study. An undergraduate with no previous experience (in fact, one of the subjects described in Section 8.2) was able to extend the checker to reduce the number of false positive warnings in the bodies of the `equals` and `compareTo` methods.

Based on common code patterns in Lucene, the student created a new `@UsesObjectEquals` annotation to indicate a type hierarchy that does not override `Object.equals`. In this case, comparison with == is allowed because it is semantically equivalent to `equals`. Use of this annotation in Lucene eliminated the need for 25 out of 105 `@SuppressWarnings` annotations.

7.2 Ease of Use

The Interning Checker checks every == operation and issues a warning if either argument is not annotated as `@Interned`. A warning suppression is required if == is used in optimizations such as caching or returning an argument when no side effect is necessary. Other == comparisons are flagged as potentially erroneous, but are legal because they are among values that are mutually unique, even if not globally unique.

Lucene was well-documented and contained few errors, making it a pleasure to annotate. This confirms our experience that the annotation process itself is simple; it is understanding the code and correcting its flaws that takes up most of the effort.

In Xerces, over 2/3 of the `@Interned` annotations (147 out of 218) were on `static final` variables that were initialized to a `String` literal; these annotations were trivial to add. The 4 false positives stemmed from optimizations that guarded possibly-expensive calls to `equals` with a short-circuiting == test.

7.3 Effectiveness

7.3.1 Xerces

The checker revealed at least 26 erroneously missing calls to `intern`. There are probably more, but the poor quality of the Xerces code and documentation made it impossible for us to determine the intended design. As just one example, the `QName` constructor documentation states, "To be used correctly, the strings must be identical references for equal strings." However, uninterned values were passed to it in at least 7 places. This is an efficiency concern because the non-interned copies take up heap space and are more expensive to compare. It can also cause run-time errors. For example, in `XSDHandler`, the `QName` is created with non-interned fields, and immediately passed into the method `getGlobalDecl(...)`, which compares the `QName.uri` using the == operator. The == comparison is guaranteed to evaluate to false. We counted all these as "other errors" in Figure 1 because we did not understand Xerces well enough to produce a test case that caused Xerces to crash.

[10] `http://xerces.apache.org/`

[11] `http://lucene.apache.org/`, SVN snapshot from July 14, 2010.

The checker also revealed 159 unnecessary calls to `intern` — the receiver was already interned. These calls clutter the code, are confusing to readers, and they reduce run-time performance. The direct run-time cost of the extra calls is negligible, but their presence may disable compiler optimizations: if they were removed, many static fields would contain compile-time constant values.

7.3.2 *Lucene*

We found 5 unnecessary calls to `intern`. For example, `FieldInfos.read` interns a value before passing it as the `name` argument to `FieldInfos.addInternal`, which then re-interns it. As another example, `FieldFlag` contains a call to `intern` with a comment "QUESTION: Need we bother here?". By using the Interning Checker, we can authoritatively state that removing the call would not cause a correctness error.

At least 5 fields that are always interned are not documented as such. For example, we were able to verify that field `StandardTermsDictReader.field` is always interned, and that Lucene is not missing any opportunities to reduce the field's memory usage. We believe this lack of documentation is unintentional, because many fields in Lucene are documented as interned, and often uses of == are justified by an explicit comment about interning.

In another 14 cases, an argument to a constructor of `Term` or `Field` was unnecessarily re-interned. However, each class has a special constructor that does not re-intern its argument, and these special constructors are used a total of 4 times. We speculate that programmers are afraid to use the special constructor more often, even when it would be legal, because they are afraid that their reasoning is incorrect or that a later code change will invalidate the program properties upon which they depended. Machine-checked documentation of interning properties, such as offered by the Interning Checker, could give programmers confidence to use these interfaces, which would lead to better code design and lower overhead.

8. NULLNESS CHECKER

The Nullness Checker verifies the absence of null pointer exceptions (NPEs) in a program by checking that (1) expressions with nullable type are never dereferenced and (2) variables with non-null type are never assigned a null value.

We performed two types of case studies: one in which first-year computer science students used the Nullness Checker on their class assignments (Section 8.2), and one in which we applied the Nullness Checker to open-source software (Section 8.3).

8.1 Ease of Creation

A reader might expect that the nullness type system is so simple that it was trivial to create and is trivial to use. In fact, the Nullness Checker is the largest checker by far that has been built with the Checker Framework. Its core is small, and most of the rest is special cases to reduce the number of false positives. This is necessary because programs use `null` as a special case in many different contexts, and program logic is often dependent on whether a variable is null. These facts also make *using* the nullness type system — determining and expressing nullness properties — harder than using any other type system, among the ones that we present.

The simple core of the Nullness Checker was easy to create, and has been incrementally enhanced since then. Overall, the effort of creating it was commensurate with its functionality. Recent extensions include the following. It recognizes when a value is a key for a map, in which case `Map.get` does not return null because of a failed lookup, but only because of null values in the map. It infers when an object is initialized, even before the end of its constructor. It supports lazy initialization, method annotations indicating absolute or conditional pre- and post-conditions, and a `@Pure` annotation that indicates a deterministic, side-effect free method that does not invalidate dataflow facts. It optionally warns about redundant nullness tests.

Of the 30 files in the Nullness Checker, 19 contain no code (e.g., qualifier annotations), 4 are for two distinct checkers (for rawness and map keys) that are packaged with the Nullness Checker, 4 handle flow-sensitivity and heuristics, and 3 are the core of the checker.

8.2 Classroom Assessment

To evaluate the Nullness Checker's ease of use for total novices, we observed 28 first-year computer science students enrolled in CSE 331[12] at the University of Washington. Prior to the course, many of the students had not been exposed to the object-oriented features of Java.

Over the course of 7 weekly problem sets, the students each developed a route-finding program over real road data. For each problem set, students submitted their solutions, received feedback, and then re-submitted their solutions. Then, as an additional problem set, the students ran the Nullness Checker to eliminate the possibility of null pointer exceptions in their code.

Students received a one-hour demo, a one-hour class lecture on pluggable type systems, the Checker Framework manual, a problem set write-up, a build file, and two small documented example files. Students had to annotate both code they had written and some staff-provided code, but no testing code; additionally, calls to the Swing library and a staff-provided library were unchecked.

8.2.1 *Ease of Use*

Students reported spending only a mean 5.4 hours reading, learning to run the tool, annotating, and bug fixing[13]. Including all time spent, students checked and fixed their code at a mean 1743 LOC per hour (913 LOC NCNB; σ =563 LOC, 321 LOC NCNB).

Students had no significant trouble using the basic `@NonNull` and `@Nullable` type qualifiers, which were the only ones that had been demoed. Students' usage improved with time, even over the 5 hours they spent. When annotating the first assignments, students sometimes wrote redundant type qualifiers that repeated a default or would have been inferred. Redundant type qualifiers were less common when annotating the later assignments.

Students under-used the other annotations supported by the Nullness Checker. Only 9 of the 28 students (32%) used the `@LazyNonNull` qualifier despite a member variable being lazily initialized in the staff-provided code for the first assignment. Similarly, students should have used `@KeyFor` more frequently to indicate membership in a map's key set. Finally, only 8 of the 28 students (29%) used method annotations about purity and pre- and postconditions — though they were appropriate for both staff-provided code and staff-specified student-written code. We conclude that the level of instruction that we gave to these novices was inadequate. It discussed theory but failed to focus on the pragmatic issues that they had most trouble with. And, it did not even mention the annotations for lazy initialization, methods, or maps, though these were covered in the Checker Framework manual.

8.2.2 *Effectiveness*

All 28 students found and fixed at least one null pointer error in their code[14]. On the original submissions, 5 of the 28 students

[12] `http://www.cs.washington.edu/education/courses/cse331/10sp/`

[13] This data is from the 11 students who categorized time spent. Min: 3 hours; median: 5; max: 12. Students who worked from home had to install the Checker Framework, but the staff had pre-installed it on the lab workstations.

[14] Two students incorrectly reported that type checking had revealed no errors, but our manual examination of their code changes indicated that they had fixed a null pointer error.

(18%) had NPEs that were detected by the staff's test suite, which consisted of system tests and some unit tests, for classes the staff had specified. After receiving feedback and resubmitting their work for automated testing, 3 students (11%) still had NPEs. After running the Nullness Checker, the staff tests detected no NPEs.

The two most common errors that students fixed were: using a return value from `BufferedReader.readline` without testing for null first (it returns null when the end of the stream has been reached) and using the formal parameter to `equals` without testing for null. Students reported that they had not thought of these cases.

On average, each student was unable to write sufficient annotations fewer than 6 times. These occurrences were indicated by suppressed warnings, improper workarounds, and (possibly inadvertent) use of a bug in the Nullness Checker. Students used `@SuppressWarnings` to suppress false positives warnings a mean 2.6 times. This modest number is encouraging, especially since one use per student was needed in staff-provided code. Students worked around the type checker in less than 2 methods on average (and in no more than 4 methods, except for one outlier student). These workarounds converted a NPE into a different erroneous behavior, which does not address the root cause of the problem. Examples are null-guarding a block of code and performing no action if the guard failed, or throwing a different runtime exception, such as `IllegalArgumentException`, when a null pointer was encountered. Most students made the same few workarounds, e.g., in the method containing calls to `BufferedReader.readline`. Another error in the student annotations was that 12 students took advantage of an unsoundness in the checking of the `@KeyFor` annotation (which is now fixed) by declaring variables as keys for a map without establishing the relationship. The 20 students that used the `@KeyFor` annotation each introduced this error in less than 2 methods on average; one student made the error in 9 methods.

We asked students what would be the best use of time, if they wished to improve the quality of their code. Less than half (13 out of 28) unequivocally stated that they would use another tactic such as reasoning about their code, writing assertions, writing tests, or using a different pluggable type system to prevent representation exposure. We were surprised by these results. In our view, using a pluggable type checker from the *beginning* of a project is well worth the effort, but other activities are usually more effective than annotating all of a *legacy* codebase. The students concurred regarding the relative effort: the most common comment, mentioned by 9 students (32%), was that the checker would be more useful if used throughout the development process, rather than at the end after all of the code has been written.

8.2.3 Followup

In a subsequent offering of the same course, we again assigned students to use the Nullness Checker on their code. There were two differences: the students received half as much training (but the training was better-focused), and they type-checked their code starting with the second assignment rather than annotating everything at the end of the term. We cannot offer definitive conclusions because the course is underway as of this writing. However, the early indications are very positive: students have had little trouble using the type-checker this time around.

8.3 Open Source Programs

We used the nullness checker to check the absence of null pointer exceptions in two codebases: Google Collections, a utility library[15], and Daikon, a dynamic invariant detector [10, 11][16].

[15] `https://code.google.com/p/google-collections/`
[16] `http://code.google.com/p/daikon/`

8.3.1 Ease of Use

Google Collections is heavily tested: it has 45,000 tests, and 2/3 of the NCNB lines of code consist of tests. FindBugs [19] has been run on it to verify the absence of null pointer errors, and it contains 275 FindBugs `@Nullable` annotations. It has been extensively used in the field.

Despite all of the previous verification effort, the Nullness Checker found its first error in Google Collections within *5 minutes* of starting work. We quickly annotated and checked the entire codebase, which is a testament to the quality of the documentation and code.

For Daikon, the annotation process was performed as a routine part of development and maintenance, over the course of several years. The annotations appear in the public Daikon version control repository at `daikon.googlecode.com`. We have no way to measure the effort involved, but we do know that the effort to write annotations, verify correct code, and find errors was completely dwarfed by the effort to fix errors. Annotation and type-checking effort was not burdensome, but bug-fixing was.

8.3.2 Effectiveness

Google Collections. We found 9 situations in which an exception is thrown within Google Collections when a client follows the documentation. We wrote a small test program that triggers each of these errors.

`ToStringFunction.apply` is annotated to take a `@Nullable` (possibly-null) argument, but uses it without checking against null. `ForMapWithDefault.hashCode` throws a null pointer exception if the map's default value is null (which is permitted). In `BiMap`, `put` and `forcePut` are documented as permitting null arguments, but subclass `ImmutableBiMap` violates the specification by throwing an exception if `null` is used. Similar problems occur with `Multimap`'s methods `put`, `putAll`, and `replaceValues` (subclass `ImmutableListMultimap` violates the specification and throws an exception), and with `StandardListMultimap.put`. Finally, `Multiset.add` is documented as taking a nullable argument, but `ImmutableMultiset` violates the specification and throws an error. The last 7 errors are a result of improper subclassing. Whereas programmers may explicitly reason about the calls their code makes, it is easy to forget to check the specification of the overridden method. A type-checker assists programmers by checking this requirement.

We found 154 missing `@Nullable` annotations, which we believe to be an oversight because the same methods (e.g., `equals`) were properly annotated elsewhere in the codebase. (We also found about 800 more missing annotations, but in those cases the code was not inconsistent, because the annotations were missing throughout. We do not count them as errors.) The chief developer said "we added what was necessary to make findbugs [sic] happy ... it's not worth our extensive time and thought to think everything over across the whole library." He apparently viewed FindBugs as an obstacle to be worked around rather than as an aid to understanding. By contrast, we were unfamiliar with the codebase but quickly checked it with the Checker Framework, and detected errors that FindBugs missed.

Daikon. Daikon's version control history indicates over 90 errors whose discovery can be unambiguously attributed to the Nullness Checker. Additionally, the Nullness Checker revealed unnecessary tests, deficient documentation, and design flaws; most of these have now been fixed. Because of the nature of the annotation process, we do not have an exact count of the errors. However, we can say that the errors are important. In one case in December 2009, a user submitted a bug report regarding a Daikon crash just days after the issue had been identified by the Nullness Checker and fixed (but before a new version had been released).

The Daikon case study reinforced the value of a static analysis tool as a nose for finding bad code smells: if code is too complex for

the type-checker to reason about and too complex for the programmer to have confidence in suppressing a false positive, then the code is probably too complex. For example, Daikon's code for testing collinearity of points has optimizations for incremental processing, sorting, and point removal. We have been unable to manually verify that no null pointer dereference will occur, nor to find a case where it does. The code should be rewritten for clarity. As another example, over time the `Invariant` class took on two distinct purposes: representing program properties, and being a factory object. The two uses must not be mixed — fields that are null in the factory objects are needed when computing program properties — and should really be represented as distinct classes.

Most suppressed warnings are due to application invariants: properties specific to the control flow and logic of this program. An example is `XorVisitor.visit`, which takes two arguments, at least one of which is non-null at all call sites. The type system cannot express nor check this property. Some other common reasons for suppressing a checker warning were when nullness of one variable is dependent on the run-time value of another variable, initialization of circular data structures, test code that did not fully initialize objects used as mocks, and re-initialization of variables in new program phases.

9. LESSONS LEARNED

Declarative syntax. The Checker Framework supports both a declarative and a procedural syntax. A user should start with a simple, declaratively-specified checker. When the limitations of the declarative syntax become too restrictive, the user can incrementally add the procedural syntax to implement more powerful features. The type rules can even utilize information beyond the program source code; as an example, the Compiler Message Checker looks up message strings in an external file.

This design has made it easy to get started and easy to extend a checker. While good for initial exploration, declaratively-specified checkers are inadequately expressive for practical use, in the current state of the art.

To the best of our knowledge, the Checker Framework is the first pluggable type checker framework to propose, and support, augmenting a fully declarative implementation with procedural code.[17]

Soundness. Given the complexity of real-world code, it is unrealistic to expect a tool to issue no false positives. The Checker Framework is designed for analyses that value soundness over limiting false positives. The user should be willing to suppress some warnings. The user obtains a correctness guarantee via a combination of the automated checker and limited manual reasoning. If a pattern of false positives emerges, a developer can incrementally add logic to soundly reduce the number of false positives.

The Checker Framework's clear semantics makes its warnings more comprehensible. Users who do not understand a warning or how it is computed are prone to ignore or misuse the warning and to miss errors. The Google engineers using FindBugs and some students using the Nullness Checker are examples.

Expressiveness. We believe type checkers represent a sweet spot in program analysis power. The modular nature of type-checking makes it fast and its warnings easy to understand. More powerful tools like model checkers and symbolic execution can prove more properties, but they scale poorly and are harder to use. Unsound heuristic-based tools capture fewer properties and give fewer guarantees.

There is a temptation to indefinitely expand a type system to increase expressiveness and capture more special cases. At some point, this makes the type system harder, not easier, to use.

We believe it is best to be conservative in adding features. For example, the Interning Checker supported only the single `@Interned` annotation for four years. After analyzing the false positives from the case studies, we added a class annotation that eliminated 24% of Lucene's false positives. The Nullness Checker has 17 annotations that have proved worth adding over the same period.

Instruction. Understanding the theory of type qualifiers is not necessary to use the Checker Framework because it fits into developers' toolchain and is a natural extension to Java's existing type system. In fact, novices preferred instruction in the nuts and bolts of using the tool to an explanation of the theory.

Annotation effort. Well-written code, with a clear design and documentation, is easy to annotate. Annotating a library is hard because *understanding* a foreign library is hard — especially understanding its undocumented features and its bugs. Writing the annotations, once the design is understood, takes negligible time.

Nonetheless, developers continue to fear the annotation burden — at least until they try it. We think that, similar to the lack of initial enthusiasm for generic types, more and more developers will come to appreciate the benefits of pluggable types, which are no more burdensome than generics. For example, the nullness type system is by far the most verbose of our type systems, with about 1 annotation per 80 lines (1 per 36–52 LOC NCNB) in Google Collections and Daikon (Figure 1). But, these projects have 13.9 and 1.9 times as many type parameters as nullness annotations.

Applicability. We have repeatedly discovered that users underestimate how much can be captured by a type system; the examples in this paper are the tip of the iceberg. Once users get in the habit of thinking how to utilize a type system and its enforcement of consistent usage among different parts of a program, the opportunities seem to arise everywhere.

On the other hand, not all code is worth annotating and checking: like any methodology, pluggable-type checking should be applied where it is appropriate and most needed. A type system is not applicable in all circumstances where its guarantees might be desired. Code that depends on user input, dynamic checks, and reflection is hard to handle in a static type system. For any type system, some program invariants are not expressible or provable.

Benefits. Many of the design, documentation, and code errors revealed by a pluggable type-checker could be found in other ways, such as testing, reviews, and other tools. Compilation is the right time to detect them, because of the well-known fact that the earlier an error is detected, the easier it is to fix. When used throughout the development process, errors can be corrected early while the design is fresh in the programmer's mind and other components have not begun to rely on it. Because our subject programs are mature, tested, and fielded, our evidence likely overstates the difficulty and understates the benefits of pluggable type checking.

10. RELATED WORK

Since our key contribution is case studies, we focus our discussion of related work on other case studies.

Our work uses the Checker Framework for creating custom type systems in Java. Several similar frameworks exist, such as Polyglot [24], JavaCOP [22, 2], JQual [18], and JastAdd [6]. The latter two do inference as well as checking. For a comparison of the frameworks, see [25]. Case studies such as ours are lacking for the other frameworks. Eckman et al. [7] give timing numbers and summary statistics for running JastAdd inference for nullness, but do not evaluate the accuracy or usefulness of the annotations. Markstrum et al. [22, 2] give timing numbers for running JavaCOP, but focus on structural pattern-matching and do not demonstrate that its type-checking results are correct or useful. Their inability to create

[17] Markstrum et al. [22] mistakenly claim otherwise. In fact, JavaCOP obtained a fully declarative syntax after JavaCOP authors viewed our demo [8] of the feature. Their paper also mischaracterizes other aspects of the Checker Framework.

a correct, scalable checker for either Javari or nullness is suggestive of the opposite. Greenfieldboyce and Foster [18] do evaluate the quality of JQual's mutability inference. They examined 50 (out of approximately 10,000) of the readonly references reported by their tool. They report that 35 are correct and desirable, and 15 are not: 3 are inferred as readonly but are immutable strings; 5 are marked as readonly because no analyzed clients use them; 7 are mutable (due to the authors not marking enough fields as "tracked", or field-sensitive). Papi et al. [25] ran 5 type checkers built on the Checker Framework over a total of 600k LOC, and detected 66 errors; this paper can be thought of as a logical extension of that one, using new checkers and enhancements of older ones, and bringing to bear an additional 3 years of experience.

Eiffel builds in a nullness type checker [23] that is similar to ours in that it adds, to the standard features of a nullness type system, a limited form of flow-sensitive type refinement (called Certified Attachment Patterns or "CAPs"), warning suppression (called the "check instruction"), and "stable attributes" that correspond to our `@LazyNonNull`, though it lacks some other features of our nullness type system. Their justifications for these features are similar to ours [25]. The Eiffel Software team annotated "thousands of classes", and Meyer et al. [23] provide summary statistics for the number of lines changed (3–11%, which is more than our 1%). Details about the conversion process and results would be interesting.

The work by Chalin et al. [3] presents the nullness checker of the Eclipse JML JDT tool. They evaluated their checker on 700 kLOC and state that 75% of all references were intended to be non-null. Their type system does not seem to support features that we found necessary, such as generics and qualifier polymorphism. The paper also presents a detailed overview of programming languages that support non-null type checks. They did not implement a framework that could be used for other type systems.

While there have been previous case studies of type systems for nullness and immutability, we are not aware of any previous type checker (or case study) for any of the other type systems in our case studies. We speculate that this is because only the Checker Framework makes it sufficiently easy to build and use a pluggable type-checker.

We are not aware of a previous implementation of interning as a type qualifier. As a result, our system is more flexible, and less disruptive to use, than previous interning approaches [21, 13, 26] in that it neither requires all objects of a given type to be interned nor gives interned objects a different Java type than uninterned ones.

Java 5 added support for `enum` types. However, we are not aware of a case study that shows the effectiveness of converting a program from fake enums to `enum` types. In our approach fake enums can be refined using the Fenum Checker, without incurring backwards-compatibility problems or performance penalties.

CQual [14, 15] is a type qualifier framework for C. It provides qualifier polymorphism and inference. The framework supports subtyping between the qualifiers and well-formedness constraints for types, both specified declaratively using a configuration file. Adding imperative type checks is not possible without modifying the framework itself. The framework has been used for const, locking, and user/kernel-level bug detection.

11. CONCLUSION

We evaluated pluggable type-checking, as embodied in the Checker Framework, on over 2 million lines of code and found hundreds of bugs, including over 40 that cause incorrect user-visible behavior, in well-tested and widely used open source programs. We conclude that the type-checkers are easy to build, to learn, and to use; they scale to realistic applications; and they find real errors that were overlooked by other verification methodologies such as testing and manual reasoning.

It is easy to improve the quality of your Java code, and you should start today! Visit the Checker Framework website at `http://types.cs.washington.edu/checker-framework/`.

REFERENCES

[1] J. R. Allen. *Anatomy of LISP*. McGraw-Hill, New York, 1978.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, pages 57–74, Oct. 2006.

[3] P. Chalin, P. James, and F. Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software*, 2(6):515–531, Dec. 2008.

[4] Checker Framework website. `http://types.cs.washington.edu/checker-framework/`.

[5] ECMA Technical Group TG49-TG4 (Eiffel) of ECMA Technical Committee 49 (Programming Languages), editor. *Standard ECMA-367 and ISO/IEC 25436:2006, Eiffel Analysis, Design and Programming Language*. ECMA International and International Standards Organization, Geneva, June 2006.

[6] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18, Oct. 2007.

[7] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *J. Object Tech.*, 6(9):455–475, Oct. 2007.

[8] M. D. Ernst. Building and using pluggable type systems with the Checker Framework. In *ECOOP*, July 2008. Tool demo.

[9] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, Sep. 12, 2008.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, May 1999.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 69(1–3):35–45, Dec. 2007.

[12] A. P. Ershov. On programming of arithmetic operations. *CACM*, 1(8):3–6, Aug. 1958.

[13] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *ML*, pages 12–19, Sep. 2006.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, June 1999.

[15] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, June 2002.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.

[17] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, Tokyo, Japan, May 1974.

[18] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, Oct. 2007.

[19] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE*, pages 13–19, Sep. 2005.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1999.

[21] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA*, pages 313–325, Nov. 2003.

[22] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative pluggable types for Java. *ACM TOPLAS*, 32(2):1–37, Jan. 2010.

[23] B. Meyer, A. Kogtenkov, and E. Stapf. Avoid a void: The eradication of null dereferencing. In *Reflections on the Work of C.A.R. Hoare*, chapter 9, pages 189–211. Springer, London, 2010.

[24] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152, Apr. 2003.

[25] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[26] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *ECOOP*, pages 54–78, Aug. 2007.

[27] O. Zendra and D. Colnet. Towards safer aliasing with the Eiffel language. In *IWAOOS*, pages 153–154, June 1999.