Università
della
Svizzera
italiana

Faculty of Informatics

Dissertation Proposal

December 20, 2017

What Do We Write?

Discovering Unexpected Language Features Usages at Large-Scale by Empirical-based Patterns

Luis Mastrangelo

Abstract

Programming languages offer a wide range of features that aim to improve programmers productivity. However, to better drive the future evolution of any programming language, we believe it is paramount to have a thorough understanding of how these features are actually being used in real codebases.

Understanding how developers make use of language features can be helpful to a broad audience besides language designers. It can aid tool builders to make more realistic assumptions; researchers to improve the state-of-the-art; and developers to implement more efficient and effective solutions by providing them best practices.

In this proposal, we target four specific JAVA features, namely, *casting*, *reflection*, *exception handling* and the *unsafe* API. We plan to divise language and API usage patterns at large-scale to properly assess this broad audience. We hope that having a better understanding on how these features are used, we can make informed decisions for these driving forces.

Research Advisor

Prof. Matthias Hauswirth

Research Co-advisor

Prof. Nathaniel Nystrom

Internal Committee Members

Prof. Walter Binder?, Prof. Antonio Carzaniga?, Prof. Gabriele Bavota?, Prof. Patrick Eugster?

External Committee Members

Prof. Hridesh Rajan?, Prof. Tobias Wrigstad?, Prof. Stefan Hanenberg?

This proposal has been approved by the dissertation committee and the director of the Ph.D. program:

Prof. Matthias Hauswirth, Research Advisor and Committee Chair, Università della Svizzera Italiana, Switzerland

Prof. Nathaniel Nystrom, Research Co-Advisor, Università della Svizzera Italiana, Switzerland

Prof. Walter Binder?, Università della Svizzera Italiana, Switzerland

Prof. Antonio Carzaniga?, Università della Svizzera Italiana, Switzerland

Prof. Gabriele Bavota?, Università della Svizzera Italiana, Switzerland

Prof. Patrick Eugster?, Università della Svizzera Italiana, Switzerland

Prof. Hridesh Rajan?, Iowa State University, United States?

Prof. Tobias Wrigstad?, Uppsala University, Sweden?

Prof. Stefan Hanenberg?, University Duisburg-Essen, Germany?

Prof. Walter Binder, Ph.D. Program Director, Università della Svizzera Italiana, Switzerland

Prof. Michael Bronstein, Ph.D. Program Director, Università della Svizzera Italiana, Switzerland

Contents

Chapter 1

Introduction

Programming language design has been always a hot topic in computer science literature. It has been extensively studied in the past decades. For instance, there is a trend in incorporating functional programming features into mainstream object-oriented languages, *e.g.*, lambdas in JAVA 8[1], C++11[2] and C# 3.0[3]; or parametric polymorphism — *i.e.*, generics — in JAVA 5[4,5].

Adding new features to a language should — *in theory* — increase programmers productivity. But once a language feature is released, little is known about how it is actually used by the developer community. Therefore, it is extremely difficult to assess how features in a programming language impact on programmers productivity. We argue that this information is of great value, because can give many insights to drive the future of any programming language.

On the other hand, Hanenberg [2010, 2014] argue that human behavior, *i.e.*, controlled experiments, should be applied to programming language usage and design. With this approach, it should be possible — in principle — to understand to what degree a language feature impacts on programming productivity. However, for any kind of controlled experiment to be valid, it must reflect reality. Otherwise, any conjecture derived from a controlled experiment can be considered truthful but useless.

Finally, understanding what developers write is not only useful in the field of language design and controlled experiments. For instance, Livshits et al. [2015] argue that most software analysis tools exclude certain dynamic features, *e.g.*, reflection, `setjmp`/`longjmp`, JNI[6], `eval`, *etc.*, from their analyses. They claim that in order to understand how the limits of analysis tools impact software, we also need to understand what kind of code is being written in the real world.

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf
[3] https://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7
[4] https://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html
[5] http://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf
[6] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

Looking at the aforementioned examples, Mazinanian et al. [2017] and Uesbeck et al. [2016] studied how developers use lambdas in JAVA and C++ respectively; while Parnin et al. [2011, 2013] did the same for generics in JAVA. This kind of studies give an insight of the adoption of lambdas and generics; which can drive future direction for language designers and tool builders, while providing developers with best practices.

1.1 Research Question

Understanding how language features are used can give many insights to language designers, tools builders, researchers and developers. This triggers our research question:

Research Question

Are there *unexpected usages of language features* in-the-wild that can give new insights to language designers, tools builders, researchers and developers?

We believe that we — as a research community — should understand what kinds of programs are written in real codebases. We can use this information to improve several aspects of the software development process and supporting informed decisions for the driving forces mentioned above. This fact opens the door for empirical studies about language features, and their use in source code repositories, *e.g.*, *GitHub*, *GitLab* or *Bitbucket*, and package managers repositories, *e.g.*, *Maven Central*[7] or *npm*[8]. Since any kind of language study must be language-specific, our plan is to focus on JAVA given its wide usage and relevance for both research and industry.

In this proposal, we plan to target four specific JAVA features, namely, *casting*, *reflection*, *exception handling*, and the *unsafe API*. We have divised — for the *unsafe API* — and we plan to divise language and API usage patterns. We believe that having usage patterns can help us to better categorize features and thus understanding how the feature is actually used.

Table 1.1: Holasd que tal

Feature	Research Question
Unsafe API	Is Java Safe?
Casting	Dynamic Features
Reflection API	Is Java someting?
Exception Mechanism	Are they used properly?

1.2 Proposal Outline

The rest of this proposal is organized as follows: Chapter 2 gives a review of the literature in the *state-of-the-art* of the different aspects related to our goal. More specifically, Chapter 2.8

[7] `http:/central.sonatype.org/`
[8] `https://www.npmjs.com/`

presents already existing code patterns related to the language features we plan to analyze. The following four chapters introduce our proposal plan for the selected features: Chapters 3.2, 3.3, 3.4 presents our *casting*, *reflection* and *exception handling* study respectively. Finally, Chapter [cha:unsafe]] shows the study we already made on the unsafe API in JAVA.

While the literature review gives a broad overview in the field, each of the following chapters have their own "Related Work" section. The rationale behind this organization is that we prefer to show how we improve over the *state-of-the-art* after having presented our plan for each feature.

Chapter 2

Literature Review

Understanding how language features and APIs are being used is a broad topic. There is plenty of research in computer science literature about empirical studies of programs; which involves several directions directly or indirecly related. Along the last decades, researchers always has been interested in understanding what kind of programs programmers write. The motivation behind these studies is quite broad and — together with the evolution of computer science itself — has shifted to the needs of researchers.

The organization of this chapter is as follows: In §2.1 we present empirical studies regarding compilers writers. How benchmarks and corpuses relate to this kind of studies is presented in §2.2. §2.3 gives an overview of other large-scale studies either in JAVA or in other languages. Related to our cast study, in §2.4 we show studies on how static type systems impact on programmers productivity. Code Patterns discovery is presented in §2.5. Finally, §2.6 gives an overview of what tools are available to extract information from a software repository, while §2.7 of how to select good candidates projects.

2.1 Compilers Writers

Already Knuth [1971] started to study FORTRAN programs. By knowing what kind of programs arise in practice, a compiler optimizer can focus in those cases, and therefore can be more effective. Alternatively, to measure the advantages between compilation and interpretation in BASIC, Hammond [1977] has studied a representative dataset of programs. Adding to Knuth's work, Shen et al. [1990] made an empirical study for parallelizing compilers. Similar works have been done for COBOL Salvadori et al. [1975]; Chevance and Heidet [1978], PASCAL Cook and Lee [1982], and APL Saal and Weiss [1975, 1977] programs.

But there is more than empirical studies at the source code level. A machine instruction set is effectively another kind of language. Therefore, its design can be affected by how compilers generate machine code. Several studies targeted the JVM Collberg et al. [2007]; O'Donoghue et al. [2002]; Antonioli and Pilz [1998]; while Cook [1989] did a similar study for Lilith in the past.

2.2 Benchmarks and Corpuses

Benchmarks are crucial to properly evaluate and measure product developement. This is key for both research and industry. One popular benchmark suite for JAVA is DaCapo Blackburn et al. [2006]. This suite has been already cited in more than thousand publications, showing how important is to have reliable benchmark suites.

Another suite is given in Tempero et al. [2010]. They provide a corpus of curated open source systems to facilitate empirical studies on source code.

For any benchmark or corpus to be useful and reliable, it must faithfully represent real world code. Therefore, we argue how important it is to make empirical studies about what programmers write.

2.3 Large-scale Codebase Empirical Studies

In the same direction to our plan, Callaú et al. [2013] perform a study of the dynamic features of SMALLTALK. Analogously, Richards et al. [2010, 2011] made a similar study, but in this case targeting JAVASCRIPT's dynamic behavior and in particular the `eval` function. Also for JAVASCRIPT, Madsen and Andreasen [2014] analyzed how fields are accessed via strings, while Jang et al. [2010] analyzed privacy violations. Similar empirical studies were done for PHP Hills et al. [2013]; Dahse and Holz [2015]; Doyle and Walden [2011] and SWIFT Rebouças et al. [2016].

Going one step forward, Ray et al. [2017] studied the correlation between programming languages and defects. One important note is that they choose relevant project by popularity, measured *stars* in *GitHub*. We argue that it is more important to analyse projects that are *representative*, not *popular*.

For JAVA, Dietrich et al. [2017] made a study about how programmers use contracts in *Maven Central*. Landman et al. [2017] have analyzed the relevance of static analysis tools with respect to reflection. They made an empirical study to check how often the reflection API is used in real-world code. They argue, as we do, that controlled experiments on subjects need to be correlated with real-world use cases, *e.g.*, *GitHub* or *Maven Central*. Winther [2011] have implemented a flow-sensitive analysis that allows to avoid manually casting once a guarded `instanceof` is provided. Dietrich et al. [2014] have studied how changes in API library impact in JAVA programs. Notice that they have used the Qualitas Corpus Tempero et al. [2010] mentioned above for their study.

Exceptions

Kery et al. [2016]; Asaduzzaman et al. [2016] focus on exceptions. They made empirical studies on how programmers handle exceptions in JAVA code. The work done by Nakshatri et al. [2016] categorized them in patterns. Whether Coelho et al. [2015] used a more dynamic approach by analysing stack traces and code issues in *GitHub*.

Collections and Generics

The inclusion of generics in JAVA is closely related to collections. Parnin et al. [2011, 2013] studied how generics were adopted by JAVA developers. They found that the use of generics do not significantly reduce the number of type casts.

Costa et al. [2017] have mined *GitHub* corpus to study the use and performance of collections, and how these usages can be improved. They have found out that in most cases there is an alternative usage that improves performance.

2.4 Controlled Experiments on Subjects

There is an extensive literature *per se* in controlled experiments on subjects to understand several aspects in programming, and programming languages. For instance, Soloway and Ehrlich [1984] tried to understand the how expert programmers face problem solving. Budd et al. [1980] made a empirical study on how effective is mutation testing. Prechelt [2000] compared how a given — fixed — task was implemented in several programming languages.

LaToza and Myers [2010] realize that, in essence, programmers need to answer reachability questions to understand large codebases.

Several authors Stuchlik and Hanenberg [2011]; Mayer et al. [2012]; Harlin et al. [2017] measure whether using a static-type system improves programmers productivity. They compare how a static and a dynamic type system impact on productivity. The common setting for these studies is to have a set of programming problems. Then, let a group of developers solve them in both a static and dynamic languages.

For these kind of studies to reflect reality, the problems to be solved need to be representative of the real-world code. Having artificial problems may lead to invalid conclusions.

The work by Wu and Chen [2017]; Wu et al. [2017] goes towards this direction. They have examined programs written by students to understand real debugging conditions. Their focus is on ill-typed programs written in HASKELL. Unfortunately, these dataset does not correspond to real-world code. Our focus is to analyze code by experienced programmers.

Therefore, it is important to study how casts are used in real-world code. Having a deep understanding of actual usage of casts can led to Informed decisions when designing these kind of experiments.

2.5 Code Patterns Discovery

Posnett et al. [2010] have extended ASM Bruneton et al. [2002]; Kuleshov [2007] to implement symbolic execution and recognize call sites. However, this is only a meta-pattern detector, and not a pattern discovery. Hu and Sartipi [2008] used both dynamic and static analysis to discover design patterns, while Arcelli et al. [2008] used only dynamic.

Trying to unify analysis and transformation tools Vinju and Cordy [2006], Klint et al. [2009] built *Rascal*, a DSL that aims to bring them together.

2.6 Tools for Mining Software Repositories

When talking about mining software repositories, we refer to extracting any kind of information from large-scale codebase repositories. Usually doing so requires several engineering but challenging tasks. The most common being downloading, storing, parsing, analyzing and properly extracting different kinds of artifacts. In this scenario, there are several tools that allows a researcher or developer to query information about software repositories.

Dyer et al. [2013a,b] built *Boa*, both a domain-specific language and an online platform[1]. It is used to query software repositories on two popular hosting services, *GitHub*[2] and *SourceForge*[3]. The same authors of *Boa* made a study on how new features in JAVA were adopted by developers Dyer et al. [2014]. This study is based *SourceForge* data. The current problem with *SourceForge* is that is outdated.

To this end, Gousios [2013] provides an offline mirror of *GitHub* that allows researchers to query any kind of that data. Later on, Gousios et al. [2014] published the dataset construction process of *GitHub*.

Similar to *Boa*, *lgtm*[4] is a platform to query software projects properties. It works by querying repositories from *GitHub*. But it does not work at a large-scale, *i.e.*, *lgtm* allows the user to query just a few projects. Unlike *Boa*, *lgtm* is based on QL, an object-oriented domain-specific language to query recursive data structures Avgustinov et al. [2016].

On top of *Boa*, Tiwari et al. [2017] built *Candoia*[5]. Although it is not a mining software repository *per se*, it eases the creation of mining applications.

Another tool to analyze large software repositories is presented in Brandauer and Wrigstad [2017]. In this case, the analysis is dynamic, based on program traces. At the time of this writing, the service[6] was unavailable for testing.

2.7 Selecting Good Representatives

Another dimension to consider when analyzing large codebases, is how relevant the repositories are. Lopes et al. [2017] made a study to measure code duplication in *GitHub*. They found out that much of the code there is actually duplicated. This raises a flag when consider which projects analyze when doing mining software repositories.

Nagappan et al. [2013] have developed the Software Projects Sampling (SPS) tool. SPS tries to find a maximal set of projects based on representativeness and diversity. Diversity dimensions considered include total lines of code, project age, activity, and of the last 12 months, number of contributors, total code churn, and number of commits.

[1] `http://boa.cs.iastate.edu/`
[2] `https://github.com/`
[3] `https://sourceforge.net/`
[4] `https://lgtm.com/`
[5] `http://candoia.github.io/`
[6] `http://www.spencer-t.racing/datasets`

2.8 Existing Code Patterns

Name	Citation	Found-In
Specifying Application Extensions	Livshits [2006]	`columba`, `jedit`, `tomcat`
Custom-made Object Serialization Scheme	Livshits [2006]	`jgap`
Improving Portability Using Reflection	Livshits [2006]	`gruntspud`, `jfreechart`
Code Unavailable Until Deployment	Livshits [2006]	`columba`
Using `Class.forName` for its Side-effects	Livshits [2006]	`jfreechart`
Getting Around Static Type Checking	Livshits [2006]	`columba`
Providing a Built-in Interpreter	Livshits [2006]	`jedit`
Guarded Casts	Winther [2011]	-
Semi-guarded Casts	Winther [2011]	-
Unguarded Casts	Winther [2011]	-
Safe Casts	Winther [2011]	-
CorrectCasts	Landman et al. [2017]	
WellBehavedClassLoaders	Landman et al. [2017]	
IgnoringExceptions1	Landman et al. [2017]	
IgnoringExceptions2	Landman et al. [2017]	
IndexedCollections	Landman et al. [2017]	
MetaObjectsInTables	Landman et al. [2017]	
MultipleMetaObjects	Landman et al. [2017]	
EnvironmentStrings	Landman et al. [2017]	
UndecidableFiltering	Landman et al. [2017]	
NoProxy	Landman et al. [2017]	

1. Specifying Application Extensions

   (a) Snippet

```
public void addHandlers(String path) {
        XmlIO xmlFile = new XmlIO(DiskIO.getResourceURL(path));
        xmlFile.load();
        XmlElement list = xmlFile.getRoot().getElement("handlerlist");
        Iterator it = list.getElements().iterator();
        while (it.hasNext()) {
            XmlElement child = (XmlElement) it.next();
            String id = child.getAttribute("id");
            String clazz = child.getAttribute("class");
            AbstractPluginHandler handler = null;
            try {
                Class c = Class.forName(clazz);
                handler = (AbstractPluginHandler) c.newInstance();
```

```
                registerHandler(handler);
            } catch (ClassNotFoundException e) {
                if (Main.DEBUG) e.printStackTrace();
            } catch (InstantiationException e1) {
                if (Main.DEBUG) e1.printStackTrace();
            } catch (IllegalAccessException e1) {
                if (Main.DEBUG) e1.printStackTrace();
        }
    }
}
```

(b) Discussion

This pattern is not clear. It would be interesting to see how these extensions are used, and what is the rationale of being of using these extensions as plug-ins.

2. Custom-made Object Serialization Scheme

(a) Snippet

```
String geneClassName = thisGeneElement.
            getAttribute(CLASS_ATTRIBUTE);
Gene thisGeneObject = (Gene) Class.forName(
              geneClassName).newInstance();
```

(b) Discussion

`Unsafe` can be used to serialize/deserialze objects as well. Actually, some unsafe implementations have a fallback to reflection in case unsafe is not available.

3. Improving Portability Using Reflection

(a) Snippet

```
try {
            Class macOS  = Class.forName("gruntspud.standalone.os.MacOSX
            Class argC[] = {ViewManager.class};
            Object arg[] = {context.getViewManager()};
            Method init = macOS.getMethod("init", argC);
            Object obj  = macOS.newInstance();
            init.invoke(obj, arg);
        } catch (Throwable t) {
            // not on macos
}

Method m = c.getMethod("clone", null);
if (Modifier.isPublic(m.getModifiers())) {
```

```
        try {
            result = m.invoke(object, null);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    try {
        // Test for being run under JDK 1.4+
        Class.forName("javax.imageio.ImageIO");
        // Test for JFreeChart being compiled
        // under JDK 1.4+
        Class.forName("org.jfree.chart.encoders.SunPNGEncoderAdapter");
    } catch (ClassNotFoundException e) {
        // ...
    }
```

(b) Discussion

What can we say?

4. Code Unavailable Until Deployment

(a) Snippet

```
Method getVersionMethod =
    Class.forName("org.columba.core.main.ColumbaVersionInfo").
        getMethod("getVersion", new Class[0]);
return (String) getVersionMethod.invoke(null,new Object[0]);
```

(b) Discussion

How could be solve this problem by using information available at compile-time?

5. Using `Class.forName` for its Side-effects

(a) Snippet

```
public JDBCCategoryDataset(String url, String driverName,
                           String user, String passwd)
    throws ClassNotFoundException, SQLException
{
    Class.forName(driverName);
    this.connection = DriverManager.getConnection(url, user, passwd);
}
```

(b) Discussion

Commonly used by JDBC API to load database drivers.

6. Getting Around Static Type Checking

(a) Snippet

```
fieldSysPath = ClassLoader.class.getDeclaredField("sys_paths");
fieldSysPath.setAccessible(true);
if (fieldSysPath != null) {
    fieldSysPath.set(System.class.getClassLoader(), null);
}
```

(b) Discussion

Is it possible to achieve the same effect using `sun.misc.Unsafe`?

7. Providing a Built-in Interpreter

(a) Snippet

(b) Discussion

This pattern seems too much like a high level pattern. Although having `semantic` patterns is what we want, a pattern without a `snippet` is too high level and application-specific.

8. Guarded Casts

(a) Snippet

```
if (o instanceof Foo) {
    Foo foo = (Foo)o;
    // ...
}
```

```
if (o instanceof Foo && ((Foo)o).isBar()) {
    // ...
}
```

```
Bar bar = o instanceof Foo ? ((Foo)o).getBar() : null;
```

`dead-if-guarded` cast version

```
if (!(o instanceof Foo)) {
    return;
}
Foo foo = (Foo)o;
```

`ensure-guarded` casts

```
if (!(o instanceof Foo)) {
    o = new Foo();
}
Foo foo = (Foo)o;
```

`while`-guarded cast

```
while (o != null && !(o instanceof Foo)) {
    o = o.parent();
}
Foo foo = (Foo)o;
```

9. Semi-guarded Casts

   (a) Snippet

```
Foo foo = ...
if (foo.isBar()) {
    Bar bar = (Bar)foo;
    // ...
}
```

10. Unguarded Casts

    (a) Snippet

```
List list = ...{ // a list of Foo elements
for (Object o : list) {
    Foo foo = (Foo)o;
    // ...
}
```

```
Calendar copy = (Calendar)calendar.clone();
```

11. Safe Casts

    (a) Snippet

```
(char)42
```

```
(Integer)42
```

12. CorrectCasts

13. WellBehavedClassLoaders

14. IgnoringExceptions1

15. IgnoringExceptions2
16. IndexedCollections
17. MetaObjectsInTables
18. MultipleMetaObjects
19. EnvironmentStrings
20. UndecidableFiltering
21. NoProxy

Chapter 3

Understanding How JAVA Language Features Are Used

Understanding the Use of Language Features in Java. To understand patterns.Mining language features thesis. Methodological Contribution, to evolve your language. Motivate the umbrella that put together those 3 pillars. In our research proposal we investigate the feasibility of

To this date, there is no clear study on how and *why* language features are used. We want to study how *casts* and *reflection* are used within the JAVA language. We believe that we can leverage this information understanding how these features are used

We begin this chapter presenting our already published work on the Unsafe API in 3.1.

With the Unsafe API we answer the sub-research question:

3.1 The Unsafe API

The material in this chapter is based on our previously published paper [Mastrangelo et al., 2015].

Our study on unsafe we have divised several usage patterns. Java is a safe language. Its runtime environment provides strong safety guarantees that any Java application can rely on. Or so we think. We show that the runtime actually does not provide these guaranteesâĂŤ-for a large fraction of todayâĂŹs Java code. Unbeknownst to many application developers, the Java runtime includes a âĂIJbackdoorâĂİ that allows expert library and framework developers to circumvent JavaâĂŹs safety guar- antees. This backdoor is there by design, and is well known to experts, as it enables them to write high-performance âĂIJsystems-levelâĂİ code in Java.

For our study on `sun.misc.Unsafe`, we needed to discover usage patterns. Given its a singleton class, we have collected call sites, and proceed with a semi-automatic analysis. On the other hand, our study related to casts involved a much more complex analysis. Therefore we have decided to implement it with manual inspection.

The exceptions mechanism is orthogonal to the features we target in this proposal. For instance, we have detected a `sun.misc.Unsafe` pattern to `throw` undeclared exceptions. Similarly, closely related to *casting*, `ClassCastException` is thrown when a cast is invalid. Therefore, we believe that these kind of studies can be complementary for our research. They can help us to understand how programmers handle exceptions in these scenarios.

For our study on `sun.misc.Unsafe`, we first tried using *Boa* with *SourceForge*. We found out that only few projects were using `sun.misc.Unsafe`. In contrast, our final study using *Maven* found that an order of magnitude more were using `sun.misc.Unsafe`.

3.2 Casts

In JAVA, type cast operators provide a way to fill the gap between compile time and runtime type safety. There is an increasing literature on how casting affects development productivity. This is done usually by doing empirical studies on development groups, which are given programming tasks they have to solve.

However, those programming tasks are usually artificial. And it is unclear whether or not they reflect the kind of code that it is actually written in the "real" world. To properly assess this kind of studies, it is needed to understand how the type cast operators are actually used.

Thus, we try to answer the question: How and why are casts being used in "real" Java code? This paper studies the casts operator in a large Java repository.

To study how are they used, and most importantly, why are they used, we have analyzed 88GB of compressed .jar files on a mainstream Java repository. We have discovered several cast patterns. We hope that our study gives support for more empirical studies to understand how a static type system impacts the development productivity.

3.2.1 Related Work

Winther [2011] proposes a flow-sensitive analysis to eliminate redundant casts in `Java`. He presents some casts patterns that he needs to deal with in his analysis. Notice that these patterns are structural ones.

Staicu et al. [2017]

Buse and Weimer [2012]

It does not show the purpose of casts, neither the rationale. What we are trying to understand is why developers use casts, and how could we avoid them, if we have to.

3.3 Reflection Patterns

This list of patterns are more of semantic patterns.

When reflection and metaprogramming can be used.

3.3.1 Related Work

3.4 Exceptions

Bibliography

Denis N. Antonioli and Markus Pilz. Analysis of the Java Class File Format. Technical report, University of Zurich, 1998.

Francesca Arcelli, Fabrizio Perin, Claudia Raibulet, and Stefano Ravani. Design Pattern Detection in Java Systems: A Dynamic Analysis Based Approach. In *Evaluation of Novel Approaches to Software Engineering*, Communications in Computer and Information Science, pages 163–179. Springer, Berlin, Heidelberg, May 2008. ISBN 978-3-642-14818-7 978-3-642-14819-4. doi: 10.1007/978-3-642-14819-4_12. URL `https://link.springer.com/chapter/10.1007/978-3-642-14819-4_12`.

Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. How Developers Use Exception Handling in Java? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 516–519, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903500. URL `http://doi.acm.org/10.1145/2901739.2903500`.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented Queries on Relational Data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL `http://drops.dagstuhl.de/opus/volltexte/2016/6096`.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-348-5. doi: 10.1145/1167473.1167488. URL `http://doi.acm.org/10.1145/1167473.1167488`.

S. Brandauer and T. Wrigstad. Spencer: Interactive Heap Analysis for the Masses. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 113–123, May 2017. doi: 10.1109/MSR.2017.35.

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 220–233, New York, NY, USA, 1980. ACM. ISBN 978-0-89791-011-8. doi: 10.1145/567446.567468. URL `http://doi.acm.org/10.1145/567446.567468`.

Raymond P. L. Buse and Westley Weimer. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337316`.

Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, December 2013. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-012-9203-2. URL `https://link.springer.com/article/10.1007/s10664-012-9203-2`.

R. J. Chevance and T. Heidet. Static Profile and Dynamic Behavior of COBOL Programs. *SIGPLAN Not.*, 13(4):44–57, April 1978. ISSN 0362-1340. doi: 10.1145/953411.953414. URL `http://doi.acm.org/10.1145/953411.953414`.

Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 134–145, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-0-7695-5594-2. URL `http://dl.acm.org/citation.cfm?id=2820518.2820536`.

Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, May 2007. ISSN 1097-024X. doi: 10.1002/spe.776. URL `http://onlinelibrary.wiley.com/doi/10.1002/spe.776/abstract`.

R. P. Cook. An empirical analysis of the Lilith instruction set. *IEEE Transactions on Computers*, 38(1):156–158, January 1989. ISSN 0018-9340. doi: 10.1109/12.8740.

Robert P. Cook and Insup Lee. A contextual analysis of Pascal programs. *Software: Practice and Experience*, 12(2):195–203, February 1982. ISSN 1097-024X. doi: 10.

1002/spe.4380120209. URL `http://onlinelibrary.wiley.com/doi/10.1002/spe.4380120209/abstract`.

Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 389–400, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4404-3. doi: 10.1145/3030207.3030221. URL `http://doi.acm.org/10.1145/3030207.3030221`.

Johannes Dahse and Thorsten Holz. Experience Report: An Empirical Study of PHP Security Mechanism Usage. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 60–70, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771787. URL `http://doi.acm.org/10.1145/2771783.2771787`.

J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73, February 2014. doi: 10.1109/CSMR-WCRE.2014.6747226.

Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada. Contracts in the Wild: A Study of Java Programs. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.9. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7259`.

M. Doyle and J. Walden. An Empirical Study of the Evolution of PHP Web Application Security. In *2011 Third International Workshop on Security Measurements and Metrics*, pages 11–20, September 2011. doi: 10.1109/Metrisec.2011.18.

R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431, May 2013a. doi: 10.1109/ICSE.2013.6606588.

Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 23–32, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-2373-4. doi: 10.1145/2517208.2517226. URL `http://doi.acm.org/10.1145/2517208.2517226`.

Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the*

36th International Conference on Software Engineering, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568295. URL `http://doi.acm.org/10.1145/2568225.2568295`.

Georgios Gousios. The GHTorent Dataset and Tool Suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL `http://dl.acm.org/citation.cfm?id=2487085.2487132`.

Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: GitHub Data on Demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597126. URL `http://doi.acm.org/10.1145/2597073.2597126`.

John Hammond. BASIC - an evaluation of processing methods and a study of some programs. *Software: Practice and Experience*, 7(6):697–711, November 1977. ISSN 1097-024X. doi: 10.1002/spe.4380070605. URL `http://onlinelibrary.wiley.com/doi/10.1002/spe.4380070605/abstract`.

Stefan Hanenberg. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 933–946, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869536. URL `http://doi.acm.org/10.1145/1869459.1869536`.

Stefan Hanenberg. Why Do We Know So Little About Programming Languages, and What Would Have Happened if We Had Known More? In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 1–1, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661102. URL `http://doi.acm.org/10.1145/2661088.2661102`.

I. R. Harlin, H. Washizaki, and Y. Fukazawa. Impact of Using a Static-Type System in Computer Programming. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 116–119, January 2017. doi: 10.1109/HASE.2017.17.

Mark Hills, Paul Klint, and Jurgen Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2159-4. doi: 10.1145/2483760.2483786. URL `http://doi.acm.org/10.1145/2483760.2483786`.

Lei Hu and Kamran Sartipi. Dynamic Analysis and Design Pattern Detection in Java Programs. In *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, pages 842–846, January 2008.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 270–283, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866339. URL `http://doi.acm.org/10.1145/1866307.1866339`.

Mary Beth Kery, Claire Le Goues, and Brad A. Myers. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 484–487, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903497. URL `http://doi.acm.org/10.1145/2901739.2903497`.

P. Klint, T. v d Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, September 2009. doi: 10.1109/SCAM.2009.28.

Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, April 1971. ISSN 1097-024X. doi: 10.1002/spe.4380010203. URL `http://onlinelibrary.wiley.com/doi/10.1002/spe.4380010203/abstract`.

Eugene Kuleshov. *Using the ASM framework to implement common Java bytecode transformation patterns*. 2007.

D. Landman, A. Serebrenik, and J. J. Vinju. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518, May 2017. doi: 10.1109/ICSE.2017.53.

Thomas D. LaToza and Brad A. Myers. Developers Ask Reachability Questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806829. URL `http://doi.acm.org/10.1145/1806799.1806829`.

Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, Stanford, California, 2006.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Commun. ACM*, 58(2):44–46, January 2015. ISSN 0001-0782. doi: 10.1145/2644805. URL `http://doi.acm.org/10.1145/2644805`.

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, October 2017. ISSN 2475-1421. doi: 10.1145/3133908. URL `http://doi.acm.org/10.1145/3133908`.

Magnus Madsen and Esben Andreasen. String Analysis for Dynamic Field Access. In *Compiler Construction*, Lecture Notes in Computer Science, pages 197–217. Springer, Berlin, Heidelberg, April 2014. ISBN 978-3-642-54806-2 978-3-642-54807-9. doi: 10.1007/978-3-642-54807-9_12. URL https://link.springer.com/chapter/10.1007/978-3-642-54807-9_12.

Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814313. URL http://doi.acm.org/10.1145/2814270.2814313.

Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384666. URL http://doi.acm.org/10.1145/2384616.2384666.

Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.*, 1(OOPSLA):85:1–85:31, October 2017. ISSN 2475-1421. doi: 10.1145/3133909. URL http://doi.acm.org/10.1145/3133909.

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491415. URL http://doi.acm.org/10.1145/2491411.2491415.

Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of Exception Handling Patterns in Java Projects: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 500–503, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903499. URL http://doi.acm.org/10.1145/2901739.2903499.

Diarmuid O'Donoghue, Aine Leddy, James Power, and John Waldron. Bigram Analysis of Java Bytecode Sequences. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, PPPJ '02/IRE '02, pages 187–192, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland. ISBN 978-0-901519-87-0. URL http://dl.acm.org/citation.cfm?id=638476.638513.

Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 3–12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985446. URL `http://doi.acm.org/10.1145/1985441.1985446`.

Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, 18(6):1047–1089, December 2013. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-012-9236-6. URL `https://link.springer.com/article/10.1007/s10664-012-9236-6`.

D. Posnett, C. Bird, and P. Devanbu. THEX: Mining metapatterns from java. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 122–125, May 2010. doi: 10.1109/MSR.2010.5463349.

L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10): 23–29, October 2000. ISSN 0018-9162. doi: 10.1109/2.876288.

Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A Large-scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM*, 60(10):91–100, September 2017. ISSN 0001-0782. doi: 10.1145/3126905. URL `http://doi.acm.org/10.1145/3126905`.

M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor. An Empirical Study on the Usage of the Swift Programming Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 634–638, March 2016. doi: 10.1109/SANER.2016.66.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806598. URL `http://doi.acm.org/10.1145/1806596.1806598`.

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL `http://dl.acm.org/citation.cfm?id=2032497.2032503`.

Harry J. Saal and Zvi Weiss. Some Properties of APL Programs. In *Proceedings of Seventh International Conference on APL*, APL '75, pages 292–297, New York, NY, USA, 1975. ACM. doi: 10.1145/800117.803819. URL `http://doi.acm.org/10.1145/800117.803819`.

Harry J. Saal and Zvi Weiss. An empirical study of APL programs. *Computer Languages*, 2 (3):47–59, January 1977. ISSN 0096-0551. doi: 10.1016/0096-0551(77)90007-8. URL `http://www.sciencedirect.com/science/article/pii/0096055177900078`.

A Salvadori, J. Gordon, and C. Capstick. Static Profile of COBOL Programs. *SIGPLAN Not.*, 10(8):20–33, August 1975. ISSN 0362-1340. doi: 10.1145/956028.956031. URL `http://doi.acm.org/10.1145/956028.956031`.

Z. Shen, Z. Li, and P. C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990. ISSN 1045-9219. doi: 10.1109/71.80162.

E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010283.

Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. Understanding and Automatically Preventing Injection Attacks on Node.js. *Microsoft Research*, January 2017. URL `https://www.microsoft.com/en-us/research/publication/understanding-automatically-preventing-injection-attacks-node-js/`.

Andreas Stuchlik and Stefan Hanenberg. Static vs. Dynamic Type Systems: An Empirical Study About the Relationship Between Type Casts and Development Time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4. doi: 10.1145/2047849.2047861. URL `http://doi.acm.org/10.1145/2047849.2047861`.

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345, November 2010. doi: 10.1109/APSEC.2010.46.

N. M. Tiwari, G. Upadhyaya, H. A. Nguyen, and H. Rajan. Candoia: A Platform for Building and Sharing Mining Software Repositories Tools as Apps. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 53–63, May 2017. doi: 10.1109/MSR.2017.56.

Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 760–771, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884849. URL `http://doi.acm.org/10.1145/2884781.2884849`.

Jurgen Vinju and James R. Cordy. How to make a bridge between transformation and analysis technologies? In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transfor-*

mation Techniques in Software Engineering, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2006/426`.

Johnni Winther. Guarded Type Promotion: Eliminating Redundant Casts in Java. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, pages 6:1–6:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0893-9. doi: 10.1145/2076674.2076680. URL `http://doi.acm.org/10.1145/2076674.2076680`.

Baijun Wu and Sheng Chen. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.*, 1(OOPSLA):105:1–105:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133929. URL `http://doi.acm.org/10.1145/3133929`.

Baijun Wu, John Peter Campora III, and Sheng Chen. Learning User Friendly Type-error Messages. *Proc. ACM Program. Lang.*, 1(OOPSLA):106:1–106:29, October 2017. ISSN 2475-1421. doi: 10.1145/3133930. URL `http://doi.acm.org/10.1145/3133930`.