Università
della
Svizzera
italiana

Faculty
of Informatics

Dissertation Proposal

January 16, 2018

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

Contents **iii**

1 Introduction **1**
1.1 Research Question 2
1.2 Proposal Outline 3

2 Understanding How JAVA Language Features Are Used **5**

3 Reflection Patterns **7**

4 Literature Review **9**
4.1 Existing Code Patterns 9
4.2 Compilers Writers 15
4.3 Benchmarks and Corpuses 15
4.4 Large-scale Codebase Empirical Studies 16
4.5 Controlled Experiments on Subjects 17
4.6 Code Patterns Discovery 17
4.7 Tools for Mining Software Repositories 18
4.8 Selecting Good Representatives 18

5 The Unsafe API **19**

6 Casts **21**
6.1 Abstract 21
6.2 Introduction 21
6.3 Related Work 22
6.4 Casts 23
6.5 Study Overview 24
6.6 Are they *casts* operator used? 25
6.7 Finding *casts* Usage Patterns 26
6.8 Preliminary Considerations 27
6.9 Complex Analysis 29

6.10 Preliminary . . . 30
6.11 Casts Usage Patterns . . . 31
6.12 Discussion . . . 31
6.13 Related Work . . . 31
6.14 Conclusions . . . 32
6.15 Latex . . . 32

7 Casts Discovery **35**
7.1 Select All Expressions . . . 35
7.2 All casts . . . 35

8 Casts Detection **37**
8.1 Lookup by ID (135 casts) . . . 37
8.2 Family polymorphism (56 casts + possibly 25 more [need to check better]) . . . 38
8.3 Typecase (55 instanceof, 65 casts) . . . 38
8.4 Factory method (26 casts, including 24 redundant) . . . 38
8.5 equals (6 instanceof, 18 casts [12 getClass]) . . . 38
8.6 search or filter by type (9 instanceof, 11 casts) . . . 39
8.7 container object (16 casts) . . . 39
8.8 testing (13 instanceof, 3 casts) . . . 39
8.9 null (11 casts) . . . 39
8.10 query result (11 casts) . . . 40
8.11 Payload (10 casts) . . . 40
8.12 lookup by type tag (9 casts) . . . 40
8.13 Argument check (6 instanceof, 3 casts) . . . 40
8.14 Reflection field or invoke (1 instanceof, 7 casts) . . . 40
8.15 Stash (8 casts) . . . 41
8.16 Object in collection (8 casts) . . . 41
8.17 covariant field of supertype (8 casts) . . . 41
8.18 Return Type Test/instanceof (5 instanceof, 3 casts) . . . 41
8.19 type parameter (7 casts) . . . 42
8.20 newInstance (1 instanceof, 5 casts) . . . 42
8.21 Redundant cast (6 casts) . . . 42
8.22 add type parameters (6 casts) . . . 42
8.23 remove type parameter (5 casts) . . . 43
8.24 readObject (4 casts) . . . 43
8.25 exception for rethrow (2 instanceof, 2 casts) . . . 43
8.26 Covariant return (3 casts) . . . 43
8.27 result check (2 instanceof, 1 cast) . . . 43
8.28 Clone (2 casts) . . . 43
8.29 Throwable.getCause (2 casts) . . . 43
8.30 Library method returning Object (2 casts) . . . 43

8.31 method argument of type Object (1 cast) 43
8.32 Heterogeneous collections (1 cast) 44
8.33 URL.openConnection (1 cast) 44
8.34 Result of binary operation (1 cast) 44
8.35 Type parameter resolution (1 cast) 44
8.36 Global flag (1 cast) 44
8.37 Singleton 44

9 Exceptions **47**

Bibliography **49**

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*[7], *GitLab*[8] or *Bitbucket*[9], and package managers repositories, *e.g.*, *Maven Central*[10] or *npm*[11]. 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: Per Feature Research Questions

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

[7] `https://github.com/`
[8] `https://gitlab.com/`
[9] `https://bitbucket.org/`
[10] `http:/central.sonatype.org/`
[11] `https://www.npmjs.com/`

1.2 Proposal Outline

The rest of this proposal is organized as follows: Chapter 4 gives a review of the literature in the *state-of-the-art* of the different aspects related to our goal. More specifically, Chapter 4.1 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 6, 3, 9 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

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 5.

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

Chapter 3

Reflection Patterns

This list of patterns are more of semantic patterns.

When reflection and metaprogramming can be used.

Related Work

Chapter 4

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 §4.2 we present empirical studies regarding compilers writers. How benchmarks and corpuses relate to this kind of studies is presented in §4.3. §4.4 gives an overview of other large-scale studies either in JAVA or in other languages. Related to our cast study, in §4.5 we show studies on how static type systems impact on programmers productivity. Code Patterns discovery is presented in §4.6. Finally, §4.7 gives an overview of what tools are available to extract information from a software repository, while §4.8 of how to select good candidates projects.

Meyerovich and Rabkin [2013]

4.1 Existing Code Patterns

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");
```

Table 4.1: Existing 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]	

```
                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

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.7 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.

4.8 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`

Chapter 5

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 guarantees for a large fraction of today's JAVA code. Unbeknownst to many application developers, the Java runtime includes a "backdoor" that allows expert library and framework developers to circumvent Java's safety guarantees. This backdoor is there by design, and is well known to experts, as it enables them to write high-performance "systems-level" 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`.

Chapter 6

Casts

6.1 Abstract

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 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.

6.2 Introduction

In programming language design, the goal of a type system is to prevent certain kind of errors at runtime. Thus, a type system is formulated as a collections of constraints that gives any expression in the program a well defined type. Type systems can be characterized in many different ways. The most common being when it is either statically or dynamically checked (usually by the compiler or interpreter).

In the context of object-oriented languages, there is usually a subtype mechanism that allows the interoperability of two different, but related types. In the particular case of JAVA (*OO* language with static type system), the cast expression[1] and the `instanceof` opera-

[1] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.16`

tor[2] provide a bridge between compile-time and runtime checking. This is due most to the subtyping mechanism found in most of these kind of languages.

But, there is a constant struggle between the advocates of these two categories. The ones for static type system claim that it help them to detect errors in advance. In the contrary, the ones for dynamic type system claim that the verbosity of a static system slows down the development progress; and any error detected by a static type system should be catched easily by a well defined test suite.

Unfortunately, there is no clear response to this dilemma. There are several studies that try to answer this question. Harlin et. al Harlin et al. [2017] test whether the use of a static type system improves development time. Stuchlik and Hanenberg Stuchlik and Hanenberg [2011] have done an empirical study about the relationship between type casts and development time. To properly assess these kind of studies, it is needed to understand what kind of casts are written, and more importantly, the rationale behind them.

Moreover, sometimes a cast indicates a design flaw in an object-oriented system.

RQ1 Can we detect when a cast is a sign of a flaw in an object-oriented design?

RQ2 Can we improve class design by studying the use of casts?

This paper tries to answer these questions. We have analyzed and studied a large JAVA repository looking for cast and related operators to see how and why are they used. We come up with cast patterns that provide the rationale behind them.

The rest of this paper is organized as follows. Section 6.4 presents an overview of casting in JAVA. Section 6.5 discusses our research questions and introduces our study. Section 6.6 presents an overview of how casts are used. Section 6.7 describes our methodology for finding casts usage patterns. Sections 6.11 and 6.12 introduce and discuss the patterns we found. Section 6.13 presents related work, and Section 6.14 concludes the paper.

GET https://api.github.com/repos/zweifisch/ob—http/languages

6.3 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.

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2

6.4 Casts

A *cast* in JAVA serves the purpose of convert two related types. As defined in the JAVA specification[3], there are several kinds of conversions. In this context we are interested in conversion of classes.

Listing 6.1 shows how the cast operator is used to change the type of an object. In this case, the target of the cast expression is the variable o (line 2), which is defined as Object. Therefore, in order to use it properly, a cast is needed.

```
1 Object o = "foo";
2 String s = (String)o;
```

Listing 6.1: Variable o is defined as Object, then casted to String.

Whenever a cast fails at runtime, a `ClassCastException` [4] is thrown. Listing 6.2 shows an example where a `ClassCastException` is thrown at runtime. In this example the exception is thrown because it is not possible to conversion from Integer to String.

```
1 Object x = new Integer(0);
2 System.out.println((String)x);
```

Listing 6.2: Incompatible types throwing `ClassCastException` at runtime.

As with any exception, the `ClassCastException` can be catched to detect whenever a cast failed. This is shown in listing 6.3.

```
1 try {
2   Object x = new Integer(0);
3   System.out.println((String)x);
4 } catch (ClassCastException e) {
5   System.out.println("");
6 }
```

Listing 6.3: Catching `ClassCastException`

Sometimes it is not desired to catch an exception to test whether a cast would fail otherwise. Thus, in addition to the cast operator, the `instanceof` operator tests whether an expression can be casted properly. Listing 6.4 shows a usage of the `instanceof` operator together with a cast expression.

```
if (x instanceof Foo) {
  ((Foo)x).doFoo();
}
```

Listing 6.4: Use of `instanceof` operator to test whether a reference is of certain type.

[3] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html`
[4] `https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html`

An alternative to using the `instanceof` operator is keeping track of the types at the application level, as shown in listing 6.5. This kind of cast is called *semi guarded* casts Winther [2011].

```
if (x.isFoo()) {
  ((Foo)x).doFoo();
}
```

Listing 6.5: Keep track of the actual types instead of `instanceof`.

Doing an *upcast* is trivial and does not require an explicit casting.

6.5 Study Overview

We believe we should care about how the casting operations are used in the wild if we want to properly support empirical studies related to static type systems. Therefore, we want to answer the following questions:

*Q*1 : **Are casting operations used in common application code?**

We want to understand to what extent third-party code actually uses casting operations.

*Q*2 : **Which features of are used?**

As provides many features, we want to understand which ones are actually used, and which ones can be ignored.

*Q*3 : **Why are features used?**

We want to investigate what functionality third-party libraries require from. This could point out ways in which the JAVA language and/or the JVM need to be evolved to provide the same functionality, but in a safer way.

To answer the above questions, we need to determine whether and how casting operations are actually used in real-world third-party JAVA libraries. To achieve our goal, several elements are needed.

Code Repository. As a code base representative of the "real world", we have chosen the *Maven Central*[5] software repository. The rationale behind this decision is that a large number of well-known JAVA projects deploy to *Maven Central* using Apache Maven[6]. Besides code written in JAVA, projects written in are also deployed to *Maven Central* using the Scala Build Tool (sbt)[7]. Moreover, *Maven Central* is the largest JAVA repository[8], and it

[5] `http://central.sonatype.org/`
[6] `http://maven.apache.org/`
[7] `http://www.scala-sbt.org/`
[8] `http://www.modulecounts.com/`

contains projects from the most popular source code management repositories, like *GitHub*[9] and *SourceForge*[10].

Artifacts. In Maven terminology, an artifact is the output of the build procedure of a project. An artifact can be any type of file, ranging from a pdf to a zip file. However, artifacts are usually jar files, which archive compiled JAVA bytecode stored in class files.

Bytecode Analysis. We examine these kinds of artifacts to analyze how they use casting operations. We use a bytecode analysis library to search for method call sites and field accesses of the `sun.misc.Unsafe` class.

Usage Pattern Detection. After all call sites and field accesses are found, we analyze this information to discover usage patterns. It is common that an artifact exhibits more than one pattern. Our list of patterns is not exhaustive. We have manually investigated the source code of the 100 highest-impact artifacts using `sun.misc.Unsafe` to understand why and how they are using it.

6.6 Are they *casts* operator used?

Statistics under the Maven repository. These stats were collected using the Maven Bytecode Dataset.

Description	Value
'.jar's size	88GB
Number of '.jar'	134,156
Number of '.jar' w/ classes	114,495
Number of classes	24,109,857
Number of methods	222,492,323
Number of bytecode instructions	4,421,391,470
Number of 'checkcast' instructions	47,622,853
Number of 'instanceof' instructions	8,411,639
Number of methods w/ 'checkcast'	27,019,431
Number of methods w/ 'instanceof'	5,267,707

Notice that around a 12% of methods contain a 'checkcast' instruction. Which means that it is used a lot.

But there are way less 'instanceof' instructions than 'checkcast'. What does it mean? A lot of 'checkcast's are unguarded.

```
--- Size ---
Total uncompressed size: 176,925 MB
--- Structural ---
Number of classes: 24,116,635
```

[9] `https://github.com/`
[10] `http://sourceforge.net/`

```
Number of methods: 222,525,678
Number of call sites: 661,713,609
Number of field uses: 334,462,791
Number of constants: 133,020,244
--- Instructions ---
Number of zeroOpCount: 833,070,650
Number of iincCount: 12,052,811
Number of multiANewArrayCount: 70,688
Number of intOpCount: 98,592,545
Number of jumpCount: 223,854,453
Number of varCount: 1,227,756,300
Number of invokeDynamicCount: 1,481,910
Number of lookupSwitchCount: 1,044,018
Number of tableSwitchCount: 1,377,260
--- Casts ---
Number of CHECKCAST: 47,947,250
Number of INSTANCEOF: 8,505,668
Number of ClassCastException: 114,049
Methods w/ CHECKCAST: 27,033,672
Methods w/ INSTANCEOF: 5,270,791
--- Error ---
Files not found: 150
```

So, yes, cast are used.

6.7 Finding *casts* Usage Patterns

One more thing: anything about Scala-specific cast patterns? You clearly need to add counts, examples, explanations, reasons, consequences (in terms of the above questions). Also, the patterns you have so far are (probably) straightforward to detect (instruction sits in method X, or operates on type Y). I'd say you'll need to look deeper (with some program analysis) to find more interesting patterns that consist of multiple instructions.

We have analyzed 88GB of jar files under the Maven Central Repository. We have used the last version of each artifact in the Maven Repository. This a representative of the artifact itself.

Then we have used ASM **?**

The **Bytecode** column refer to either an cast related instruction or exception. These are the cast related bytecodes:

checkcast as specified by: [11]

[11] `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html\#jvms-6.5.checkcast`

instanceof as specified by: [12]
ClassCastException as specified by: [13]
The following two columns indicates how many bytecode where found in:

- **local**

My local machine. This machine contains a **partial** download of a current snapshot of Maven Central. Re-download all the artifacts is in progress.

- **fermat**

fermat.inf.usi.ch machine. This machine contains an old snapshot of Maven Central (2015)

We carry out our analysis at the bytecode level on the Maven Repository. Since we are not interested in the artifacts evolution, for our analysis we used the last version of each artifact. In total we have analysed **88GB** of compressed '.jar' files.

6.8 Preliminary Considerations

For the bytecode analysis, we need to take into consideration certain code is being compiled. This is why we need to take the following preliminary considerations.

6.8.1 Simple cast

```
Object o = "Ciao";
return (String)o;
```

```
0: ldc           #2                  // String Ciao
2: astore_0
3: aload_0
4: checkcast     #3                  // class java/lang/String
7: areturn
```

6.8.2 Generics vs. Non-generics

The following two Java snippets get compiled to the same bytecode instructions as showed below. Notice that the two snippets only differ in the use of Generics.

```
ArrayList l = new ArrayList();
l.add("Ciao");
return (String)l.get(0);
```

[12] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html\#jvms-6.5.instanceof

[13] https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html

```
ArrayList<String> l = new ArrayList<String>();
l.add("Ciao");
return l.get(0);
```

```
 0: new           #2        // class java/util/ArrayList
 3: dup
 4: invokespecial #3        // Method java/util/ArrayList."<init>":()V
 7: astore_0
 8: aload_0
 9: ldc           #4        // String Ciao
11: invokevirtual #5        // Method java/util/ArrayList.add:(Ljava/lang/Object;)Z
14: pop
15: aload_0
16: iconst_0
17: invokevirtual #6        // Method java/util/ArrayList.get:(I)Ljava/lang/Object;
20: checkcast     #7        // class java/lang/String
23: areturn
```

6.8.3 Upcast

The following snippet shows how even in the presence of a cast in the source code, no actual ‘checkcast‘ is emitted.

```
return (Object)"Foo";
```

```
0: ldc           #2                  // String Ciao
2: areturn
```

6.8.4 Conditional Operator

Using the conditional operator produces the following bytecode. [MavenDS](https://bitbucket.org/acuarica/mavends)

[JNIF](https://bitbucket.org/acuarica/jnif)

Queries

To retrieve the stats showed above, we have used SQL queries against the bytecode database. Each individual query is aimed to answer a precise question. The following list presents all the SQL queries used to retrieve the stats, and its respective answer (after the ‘;‘).

1. [How many checkcast instructions?](sql/checkcast-count.out)
2. [‘checkcast‘ most used arguments](sql/checkcast-most-used-args.out)

3. [‘checkcast‘ most used targets](sql/checkcast-most-used-target.out)
4. [How many classes?](sql/class-count.out)
5. [How many bytecode instructions?](sql/code-count.out)
6. [How many ‘equals‘ methods?](sql/equals-method-count.out)
7. [How many ‘equals‘ methods with ‘checkcast‘?](sql/equals-method-w-checkcast-count.out)
8. [How many ‘equals‘ methods with ‘instanceof‘?](sql/equals-method-w-instanceof-count.out)
9. [How many ‘instanceof‘ instructions?](sql/instanceof-count.out)
10. [‘instanceof‘ most used arguments](sql/instanceof-most-used-args.out)
11. [‘instanceof‘ most used targets](sql/instanceof-most-used-target.out)
12. [How many ‘.jar‘ files?](sql/jar-count.out)
13. [How many ‘.jar‘ files with classes?](sql/jar-w-classes-count.out)
14. [How many methods?](sql/method-count.out)
15. [How many methods with ‘checkcast‘ instruction?](sql/method-w-checkcast-count.out)
16. [How many methods with ‘instanceof‘ instruction?](sql/method-w-instanceof-count.out)
17. [How many methods with signature?](sql/methods-w-signature.out)

6.9 Complex Analysis

Now the following problem comes: How to extract code patterns? The database itself is not enough, and it faces scalability problems.

The idea would be to use method slicing, both backward and forward. In this way we can see how the casting are being used.

After the slicing, we could implement some sort of method equivalence to detect different patterns.

6.10 Preliminary

I started by downloading github projects. I grabbed all Java projects with more than 10,000 stars. This was 35 projects. They range in size from 992 lines of code to 588,302. I don't think this approach is necessarily representative since most of these projects seem to be libraries or frameworks (hence many stars), but I had to start somewhere.

I then searched for casts and instanceofs in the projects. I ignored primitive casts. I found 33788 casts, 14828 instanceof. Nb. we should also look at calls to getClass since these are sometimes used instead of instanceof (particularly often in equals()).

I then started to go through the source by hand, inspecting each cast. For each cast (instanceof), I put a comment trying to classify the cast into some sort of pattern. Most are easily classifiable, others require inspecting other code to see the type hierarchy. I then looked at all the commented casts again and tried to lump them together into more general patterns. I only managed to inspect 12 of the smaller projects (including one with 0 casts, one with just 1, and one with just 2). The largest project I looked at had 149 casts. The remaining projects have from 115 to 11,617 casts (spring-framework). My approach clearly doesn't scale, but I wanted to see what I could do manually. All in all, I looked at 481 casts and 106 instanceofs.

First thing to note in general. Most casts don't have an associated instanceof. This is because of, shall we say, a lack of defensive programming. I found this surprising. It seems a lot of code (particularly Android GUI code) is constructed on top of frameworks that return interface types (or even Object) a lot and cast to application-specific types without checking, because presumably, the programmer knows best.

Here are the patterns I found, in order of usages. The family polymorphism pattern is the most dubious, since it requires looking at the class hierarchy in more detail than I did). I think some of these patterns could be restated, cleaned up, merged, split, etc.

—

Now, what to make of this? First, I'm not claiming these are all the patterns or that these patterns are the right patterns. But, I think we should ask ourselves if doing a static analysis (either on bytecode or source) will find most of these patterns, and if so what kind of analysis is needed. Bytecode analysis won't find, say, the redundant cast pattern or some of the patterns involving generics, because these compile into a no-op. I think some of these patterns require application-specific knowledge that any static analysis would have difficulty finding.

Most of the patterns are very local: you just have to look at the line of code containing the cast or a few lines before it to identify the pattern. The main thing is to know where the value being cast is coming from. Most of the time, you don't even have to look at the class hierarchy, but for some patterns (e.g., family polymorphism), you do have to know what is the static type of the object being cast and what is its relationship to the cast type? For some of the patterns (e.g., stash), it might be useful to find matching calls: for instance, one method calls setTag and another calls getTag, casting to the type of the object that set stored

by setTag.

Several projects use application-specific type tags rather instanceof. Sometimes, type tests are buried in other methods (e.g., the code calls a method that does an instanceof and returns boolean (see the type test pattern), then uses the boolean result to check that a cast will succeed.

I don't have a good sense yet for how many of these patterns are the result of language deficiencies. Certainly typecase can be replaced with a visitor pattern (or pattern matching in a better language). The family polymorphism pattern requires either type parameters or (better) abstract types. Scala was designed to address this. lookup by ID requires some sort of typed heterogeneous collections (like an HList), which is difficult even in Scala or Haskell.

—

Since manual inspection is very slow, I think we need a way to speed up the inspection process. I looked at only about 150 casts per hour. For the projects I downloaded, at this rate, it would take about 320 hours to look at them all, i.e., 40 hours a week for 8 weeks. Clearly we need to be faster, either by sampling or by scaling up the inspection process (crowd sourcing?).

We should be more careful about the choice of projects to inspect. Popularity (github stars) isn't exactly representative. It was just easy to do the search.

Here are some more questions I had while looking at this:

- Are casts local? In a given project, are casts limited to just a few classes or are they widespread?
- How many casts are dominated by an instanceof in the same method (or in another method)?
- How many typecase are "real" in that there's actually more than one alternative?
- How often does it happen that an unguarded cast cannot possibly fail in any execution (for instance when there's really only one class implementing an interface and therefore a cast (to the class) must succeed because there are no objects of any other class). I'm not sure if this is the right why to ask the question.

6.11 Casts Usage Patterns

6.12 Discussion

Here we discuss.

6.13 Related Work

Relwork.

6.14 Conclusions

asdf

6.15 Latex

- Guarded Type Promotion – Eliminating Redundant Casts in Java~**?**

Study of type casts in several project. Quite similar to what we want to do. Focus on Guarded Type casts.

- Contracts in the Wild: A Study of Java Programs~**?**

Investigate 25 fix contract patterns. Section 2.3: Come up with new Contract Patterns.

- Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study~**?**.

They also have done a study on Casts. But only for a small curated sets of projects. They analyze the relevance of static analysis tools w.r.t reflection. We want to study Reflection in the Wild. Empirical Studies on subjects need to be correlated with real world use cases, e.g. Maven Repository.

- Static vs. Dynamic Type Systems: An Empirical Study About the Relationship between Type Casts and Development Time~**?**

Studied the type casts in relation of development time. Group study. We want to Study Casts in the Wild.

- An empirical study of the influence of static type systems on the usability of undocumented software

? Similar to Challenges ...

- Impact of Using a Static-Type System in Computer Programming~**?**

Test whether the use of a Static-Type System improves productivity. Productivity in this case is measured by development time. Two languages, a statically and dynamically-typed. Two programming tasks, Code a program from scratch and Debug a faulty program. Two program kinds, Simple program and Encryption program. A static-type system does not impact coding a program form scratch. Nevertheless, a static-type system does make software productivity improve when debugging a program.

- Empirical Study of Usage and Performance of Java Collections~**?**.

Mining GitHub corpus to study the use of collections, and how these usages can be improved.
Mining metapatterns in Java~**?**
Adoption of Java Generics3~**?**

6.15.1 Exception Handling

Android~**?**

How developer use exception handling in java~**?**
Libraries java exception~**?**
bdd~**?**
java generics championed~**?**
code smell~**?**

6.15.2 Evidence Languages

Similar to our work related to **Unsafe ?**

Chapter 7

Casts Discovery

7.1 Select All Expressions

```
import java

from Expr e
select e
```

7.2 All casts

Snippet to select all cast expressions.

```
import java

from CastExpr ce
select ce
```

Chapter 8

Casts Detection

8.1 Lookup by ID (135 casts)

Lookup an object by ID or tag or name and cast result (used often in Android code).
`getAttribute` returns `Object`.

```
AuthState authState = (AuthState) context.getAttribute(ClientContext.TARGET_AUTH_
```

```
AuthState authState = (AuthState) field.get(obj);
```

```
import java

/* Expression 'e' assumes that 'v' could be of type 't'. */
predicate isLookup(Expr e, FieldAccess fa) {
  exists (CastExpr ce | ce = e |
    exists (MethodAccess ma | ma = ce.getExpr() |
      not ma.getMethod().isStatic() and not ma.getMethod().isVarargs() and ma.get
      ma.getMethod().getNumberOfParameters() = 1 and
      ma.getMethod().getParameterType(0).getTypeDescriptor() = "Ljava/lang/String
      ma.getMethod().getReturnType().getTypeDescriptor() = "Ljava/lang/Object;" a
      ma.getArgument(0).getType().getTypeDescriptor() = "Ljava/lang/String;" and
      ma.getArgument(0) = fa and
      fa.getField().isFinal() and fa.getField().isStatic() and //fa.getField().is
      fa.getField().getType().getTypeDescriptor() = "Ljava/lang/String;" // Doubl
    )
  )
}

from Expr e, FieldAccess fa
where isLookup(e, fa)
select e, "Expression is " + e + " " + fa.getField().pp()
```

```
//+ " " + fa.getField().getAnAssignedValue()
```

This is known to the application, but only at runtime. Type-safe runtime dictionary. Is it worth to change the API?

8.2 Family polymorphism (56 casts + possibly 25 more [need to check better])

8.2.1 Description

Two or more mutually dependent classes are subtyped, but fields or method parameters in the base class cannot be overridden in the subtype to use the derived types. Also includes casting to "internal" classes. Also includes casting "context" objects to a subtype. Usually unchecked (16 instanceofs classified as typecase or argument check are related to the cast in this pattern, so maybe should be reclassified). Includes also some "quasi reflection" calls to the java annotation processing API.

8.3 Typecase (55 instanceof, 65 casts)

instanceof + cast on known subtypes of the static type. Often there's just one case and the default case (i.e., instanceof fails) does a no-op or reports an error. 11 of the casts here are checked against application-specific type tags rather than instanceof. The one case typecase is possibly the same as family polymorphism.

8.4 Factory method (26 casts, including 24 redundant)

Cast factory method result to subtype (special case of family polymorphism) Usually Logger.getLogger.

8.5 equals (6 instanceof, 18 casts [12 getClass])

instanceof (or getClass) + cast in equals to check if argument has same type as receiver.

```
@Override
public boolean equals(@NullableDecl Object object) {
  if (object instanceof StringConverter) {
    StringConverter that = (StringConverter) object;
    return sourceFormat.equals(that.sourceFormat) && targetFormat.equals(that.tar
  }
  return false;
}
```

```
import java

predicate isEquals(Method m) {
  m.getName() = "equals" and m.getNumberOfParameters() = 1 and not m.isAbstract()
  m.getParameterType(0).getTypeDescriptor() = "Ljava/lang/Object;" and not m.getP
  m.getReturnType().getTypeDescriptor() = "Z"
}

from CastExpr ce, Method m
where ce.getEnclosingCallable() = m and isEquals(m)
select m
```

8.6 search or filter by type (9 instanceof, 11 casts)

search or filter a collection by inspecting the types (and often other properties) of the objects in the collection. Note the collection could be an ad-hoc linked list too.

8.7 container object (16 casts)

the container or parent of an object in some composite should be a particular type, cast to it

8.8 testing (13 instanceof, 3 casts)

instanceof in a test (did a method under test create the right object?), or uses getClass, then might cast to access fields

8.9 null (11 casts)

Cast to null to resolve method overloading ambiguity

```
onSuccess(statusCode, headers, (String) null);
```

```
import java

from CastExpr ce, NullLiteral nl
where ce.getExpr() = nl
select ce
```

8.10 query result (11 casts)

Cast a query result (either SQL query or XPath or application-specific)

8.11 Payload (10 casts)

Cast access to message payload (usually Object) 6 or 10 instances that use a type tag to check the message type and cast to the right pattern – maybe these cases should be considered typecase.

```
case FAILURE_MESSAGE:
    response = (Object[]) message.obj;
    if (response != null && response.length >= 4) {
        onFailure((Integer) response[0], (Header[]) response[1], (byte[]) respons
    } else {
        AsyncHttpClient.log.e(LOG_TAG, "FAILURE_MESSAGE didn't got enough params'
    }
    break;
```

8.12 lookup by type tag (9 casts)

Lookup in a collection using a application-specific type tag or a java.lang.Class

8.13 Argument check (6 instanceof, 3 casts)

Check that method argument has expected type (subtype of declared type) typically in overridden methods.

8.14 Reflection field or invoke (1 instanceof, 7 casts)

Cast result of field access or method invocation using reflection.

```
public static void endEntityViaReflection(HttpEntity entity) {
    if (entity instanceof HttpEntityWrapper) {
        try {
            Field f = null;
            Field[] fields = HttpEntityWrapper.class.getDeclaredFields();
            for (Field ff : fields) {
                if (ff.getName().equals("wrappedEntity")) {
                    f = ff;
                    break;
```

```
                }
            }
            if (f != null) {
                f.setAccessible(true);
                HttpEntity wrapped = (HttpEntity) f.get(entity);
                if (wrapped != null) {
                    wrapped.consumeContent();
                }
            }
        } catch (Throwable t) {
            log.e(LOG_TAG, "wrappedEntity consume", t);
        }
    }
}
```

8.15 Stash (8 casts)

Cast access to field of type Object used to stash a value (typically a tag value in a GUI object, or a message payload)

8.16 Object in collection (8 casts)

Cast when accessing an object from a unparameterized collection object or a collection instantiated on Object rather than a more precise type.

Includes one overly complicated use of Java 8 streams.

8.17 covariant field of supertype (8 casts)

cast field of supertype which has less-specific type (same as family polymorphism?). Often unchecked cast to a subinterface with a presumed type.

8.18 Return Type Test/instanceof (5 instanceof, 3 casts)

typically just a method wrapping an instanceof

```
private static boolean a(Exception e) {
  return e instanceof RuntimeException;
}
```

```
import java

from InstanceOfExpr ie, ReturnStmt rs
where rs.getResult() = ie
select rs, ie
```

8.19 type parameter (7 casts)

Unchecked casts to a method type parameter (essentially cast to whatever the caller expects to be returned). Unchecked casts to class type parameter (simulating a self type). Casting to T[].

8.20 newInstance (1 instanceof, 5 casts)

cast result of Class or Array.newInstance

8.21 Redundant cast (6 casts)

This is a cast that should always succeed based on the static type. Some of these seem to be because some of the types changed during a refactoring and the cast was not removed. Others seem to be for documentation purposes or just paranoia.

```
final Result<List<Data>> result2 = JSON.parseObject("{\"data\":[]}", new TypeRefe
assertNotNull(result2.data);
assertTrue(result2.data instanceof List);

import java

from InstanceOfExpr ioe, RefType t, RefType ct
where t = ioe.getExpr().getType()
  and ct = ioe.getTypeName().getType()
  and ct = t.getASupertype+()
select ioe, "There is no need to test whether an instance of $@ is also an instan
  t, t.getName(),
  ct, ct.getName()
```

8.22 add type parameters (6 casts)

add type parameters to an un-parameterized collection or wildcard collection

8.23 remove type parameter (5 casts)

remove a type parameter from a collection (or java.lang.Class) or to replace parameter with wildcard

8.24 readObject (4 casts)

cast result of readObject()

8.25 exception for rethrow (2 instanceof, 2 casts)

instanceof + cast an exception to RuntimeException or Error to rethrow in handler

8.26 Covariant return (3 casts)

Cast the result of a super call in an overridden method with covariant return (see also family polymorphism)

8.27 result check (2 instanceof, 1 cast)

check result of a call has the right type

8.28 Clone (2 casts)

cast result of clone()

8.29 Throwable.getCause (2 casts)

Throwable.getCause has type Throwable, cast to Exception

8.30 Library method returning Object (2 casts)

Cast because some library method returns Object (e.g., the version object in Apache JDO). Similar to stash?

8.31 method argument of type Object (1 cast)

overridden method takes an Object not something more specific

8.32 Heterogeneous collections (1 cast)

Accessing a collection that holds values of different types (usually a Collection<Object> or a Map<K, Object>).

Easily confused with object in collection so need to revisit usages of both

8.33 URL.openConnection (1 cast)

The method is declared to return URLConnection but can return a more specific type based on the URL string. Cast to that. Should generalize this pattern.

8.34 Result of binary operation (1 cast)

Cast result of binary operation to subtype.

8.35 Type parameter resolution (1 cast)

Use reflection to get class object for a type parameter, then cast to Class<T>.

8.36 Global flag (1 cast)

Cast to a known demo subclass when running in demo mode. This should be some sort of typecase I guess, but we check a global boolean flag (or a method in a configuration object) rather than a type tag or an instanceof.

8.37 Singleton

Unguarded pattern

```
public void add(String key, String value) {
    if (key != null && value != null) {
        Object params = urlParamsWithObjects.get(key);
        if (params == null) {
            // Backward compatible, which will result in "k=v1&k=v2&k=v3"
            params = new HashSet<String>();
            this.put(key, params);
        }
        if (params instanceof List) {
            ((List<Object>) params).add(value);
        } else if (params instanceof Set) {
```

```
                ((Set<Object>) params).add(value);
            }
        }
}
```

Chapter 9

Exceptions

Here we talk about exception, maybe?

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.

Leo A. Meyerovich and Ariel S. Rabkin. Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509515. URL http://doi.acm.org/10.1145/2509136.2509515.

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, *Transformation 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.