Università
della
Svizzera
italiana

Faculty
of Informatics

Dissertation Proposal

November 2, 2018

When and How JAVA Developers Give Up Static Type Safety

Luis Mastrangelo

Abstract

The main goal of a static type system is to prevent certain kind of errors from happening at run-time. A type system is formulated as a set of constraints that gives any expression or term in a program a well-defined type. Yet mainstream programming languages are endowed with type systems that provide the means to circumvent their constraints through the *unsafe intrinsics* and *casting* mechanisms.

We want to understand how and when developers circumvent these constraints. This knowledge can be: a) a recommendation for current and future language designers to make informed decisions b) a reference for tool builders, *e.g.*, by providing more precise or new refactoring analyses, c) a guide for researchers to test new language features, or to carry out controlled programming experiments, and d) a guide for developers for better practices.

We plan to empirically study how these two mechanisms — unsafe intrinsics and casting — are used by JAVA developers to circumvent the static type system. We have devised (for a subset of unsafe intrinsics) and we are devising (for casting) usage patterns, recurrent programming idioms to solve a specific issue. We believe that having usage patterns can help us to better categorize use cases and thus understand how those features are used.

Research Advisor
Prof. Matthias Hauswirth

Research Co-advisor
Prof. Nathaniel Nystrom

Internal Committee Members
Prof. Antonio Carzaniga, Prof. Gabriele Bavota

External Committee Members
Prof. Jan Vitek, Prof. Hridesh Rajan

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. Antonio Carzaniga, Università della Svizzera Italiana, Switzerland

---

Prof. Gabriele Bavota, Università della Svizzera Italiana, Switzerland

---

Prof. Jan Vitek, Northeastern University & Czech Technical University

---

Prof. Hridesh Rajan, Iowa State University

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

---

Prof. Olaf Schenk, Ph.D. Program Director, Università della Svizzera Italiana, Switzerland

Contents

Chapter 1

Introduction

In programming language design, the main goal of a *static* type system is to prevent certain kind of errors from happening at run-time. A type system is formulated as a set of constraints that gives any expression or term in a program a well-defined type. As Pierce [2002] states: "A type system can be regarded as calculating a kind of *static* approximation to the run-time behaviors of the terms in a program." These constraints are enforced by the *type-checker* either when compiling or linking the program. Thus, any program not satisfying the constraints stated within a type system is simply rejected by the type-checker.

Nevertheless, often the static approximation provided by a type system is not precise enough. Being static, the analysis done by the type-checker needs to be conservative: It is better to reject programs that are valid, but whose validity cannot be ensured by the type-checker, rather than accept some invalid programs. However, there are situations when the developer has more information about the program that is too complex to explain in terms of typing constraints. To that end, programming languages often provide *mechanisms* that make the typing constraints less strict to permit more programs to be valid, at the expense of causing more errors at run-time. These mechanisms are essentially two: *Unsafe Intrinsics* and *Casting*.

Unsafe Intrinsics. Unsafe intrinsics is the ability to perform certain operations *without* being checked by the compiler. They are *unsafe* because any misuse made by the programmer can compromise the entire system, *e.g.*, corrupting data structures without notice, or crashing the run-time system. Unsafe intrinsics can be seen in safe languages, *e.g.*, JAVA, C#, RUST, or HASKELL. Foreign Function Interface (*FFI*), *i.e.*, calling native code from within a safe environment is unsafe. It is so because the run-time system cannot guarantee anything about the native code. In addition to FFI, some safe languages offer so-called *unsafe* blocks, *i.e.*, making unsafe operations within the language itself, *e.g.*, C#[1] and RUST[2] . Other languages provide an API to perform unsafe operations, *e.g.*, HASKELL[3] and JAVA. But in the case of JAVA,

[1] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code

[2] https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html

[3] http://hackage.haskell.org/package/base-4.11.1.0/docs/System-IO-Unsafe.html

the API to make unsafe operations, `sun.misc.Unsafe`, is unsupported[4] and undocumented. It was originally intended for internal use within the JDK, but as we shall see later on, it is used outside the JDK as well.

Casting. Programming languages with subtyping such as JAVA or C++ provide a mechanism to *view* an expression as a different type as it was defined. This mechanism is often called *casting* and takes the form `(T)t`. Casting can be in two directions: *upcast* and *downcast*. An upcast conversion happens when converting from a reference type S to a reference type T, provided that T is a *supertype* of S. An upcast does not require any explicit casting operation nor compiler check. However, as we shall see later on, there are situations where an upcast requires an explicit casting operation. On the other hand, a downcast happens when converting from a reference type S to a reference type T, provided that T is a *subtype* of S. Unlike upcasts, downcasts require a run-time check to verify that the conversion is indeed valid. This implies that downcasts provide the means to bypass the static type system. By avoiding the type system, downcasts can pose potential threats, because it is like the developer saying to the compiler: *"Trust me here, I know what I'm doing"*. Being a escape-hatch to the type system, a cast is often seen as a design flaw or code smell [Tufano et al., 2015] in an object-oriented system.

1.1 Research Question

If static type systems aim to prevent certain kind of errors from happening at run-time, yet they provide the means to circumvent their constraints, why exactly does one need to do so? Are these mechanisms actually used in real-world code? If yes, then how so? This triggers our **main research question**:

MRQ

For what purpose do developers circumvent static type systems?

We have confidence that this knowledge can be: a) a reference for current and future language designers to make informed decisions about programming languages, *e.g.*, the adoption of *Variable Handles* in JAVA 9 [Lea, 2014], or the addition of *Smart Casts* in KOTLIN,[5] b) a reference for tool builders, *e.g.*, by providing more precise or new refactoring analyses, c) a guide for researchers to test new language features, *e.g.*, Winther [2011] or to carry out controlled experiments about programming, *e.g.*, Stuchlik and Hanenberg [2011] and d) a guide for developers for best or better practices.

1.2 Plan

To answer our question above, we plan to empirically study how the two aforementioned mechanisms — unsafe intrinsics and casting — are used by developers. Since any kind of

[4] `http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html`

[5] `https://kotlinlang.org/docs/reference/typecasts.html#smart-casts`

language study must be language-specific, our plan is to focus on JAVA given its wide usage and relevance for both research and industry.[6] Moreover, we focus on the JAVA Unsafe API to study unsafe intrinsics, given than the Java Native Interface already has been studied in Tan et al. [2006]; Tan and Croft [2008]; Kondoh and Onodera [2008]; Sun and Tan [2014]; Li and Tan [2009]. In Chapter 2 we give a review of the literature in empirical studies of programming languages features. Sections 2.3.1 and 2.3.2 review the *state-of-the-art* of the different aspects related to the two proposed studies.

To better drive our *main research question*, we propose to answer the following set of sub-questions. To answer these research sub-questions, we have already devised (for the Unsafe API) and we are devising (for casting) *usage patterns*. Usage patterns are *recurrent programming idioms* used by developers to solve a specific issue. We believe that having usage patterns can help us to better categorize use cases and thus understand how these mechanisms are used. These patterns can provide an insight into how the language is being used by developers in real-world applications. Overall these sub-questions will help us to answer our MRQ:

Unsafe API.

*URQ*1 : **To what extent does the Unsafe API impact common application code?** We want to understand to what extent code actually uses Unsafe or depends on it.

*URQ*2 : **How and when are Unsafe features used?** We want to investigate what functionality third-party libraries require from Unsafe. This could point out ways in which the JAVA language and/or the JVM can be evolved to provide the same functionality, but in a safer way.

These questions have been already answered in our previous published study on the Unsafe API in JAVA [Mastrangelo et al., 2015]. Chapter 3 presents a summary of this study.

Casting.

*CRQ*1 : **How frequently is casting used in common application code?** We want to understand to what extent application code actually uses casting operations.

*CRQ*2 : **How and when casts are used?** If casts are actually used in application code, we want to know how and when developers need to escape the type system.

*CRQ*3 : **How recurrent are the patterns for which casts are used?** In addition to understand how and when casts are used, we want to measure how often developers need to resort to certain idioms to solve a particular problem.

Finally, in Chapter 4 we present our plan for the *casting* study, showing the results we have so far.

[6] `https://www.tiobe.com/tiobe-index/`

Chapter 2

Literature Review

Understanding how developers use language features and APIs is a broad topic. There is plenty of research in the computer science literature about empirical studies of programs which involves multiple *dimensions* directly related to our plan. Over the last decades, researchers always have been interested in understanding what kind of programs developers write. The motivation behind these studies is quite broad, and has been shifted to the needs of researchers, together with the evolution of computer science itself.

For instance, to measure the advantages between compilation and interpretation in BASIC, Hammond [1977] studied a representative dataset of programs. 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. Adding to Knuth's work, Shen et al. [1990] conducted 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. Miller et al. [1990, 1995]; Forrester and Miller [2000] studied the reliability of programs using *fuzz* testing. Dieckmann and Hölzle [1999] studied the memory allocating behavior in the SPECjvm98 benchmarks.[1] The importance of conducting empirical studies of programs gave rise to the International Conference on Mining Software Repositories[2] in 2004.

When conducting empirical studies about programs, multiple dimensions are involved. The first one is *What to analyze?* Benchmarks and corpora are used as a source of programs to analyze. Another aspect is how to select good candidates projects from a large-base software repository. This is presented in §2.1. After the selection of programs to analyze is set, comes the question *how to analyze them?* An overview of what tools are available to extract information from software repositories is given in §2.2. With this infrastructure, *what questions do researchers ask?* In §2.3, we give an overview of large-scale empirical studies that show what kind of questions researchers ask. This chapter ends by presenting the related work more specific to the Unsafe API and Casting in §2.3.1 and §2.3.2 respectively.

[1] `https://www.spec.org/jvm98/`

[2] `http://www.msrconf.org/`

2.1 Benchmarks and Corpora

Benchmarks are crucial to properly evaluate and measure product development. This is key for both research and industry. One popular benchmark suite for JAVA is the DaCapo Benchmark [Blackburn et al., 2006]. This suite has been already cited in more than thousand publications, showing how important is to have reliable benchmark suites. The SPECjvm2008[3] (Java Virtual Machine Benchmark) and SPECjbb2000[4] (Java Business Benchmark) are another popular JAVA benchmark suite.

Another suite has been developed by Tempero et al. [2010]. They provide a corpus of curated open source systems to facilitate empirical studies on source code. On top of Qualitas Corpus, Dietrich et al. [2017b] provide an executable corpus of JAVA programs. This allows any researcher to experiment with both static and dynamic analysis.

For any benchmark or corpus to be useful and reliable, it must faithfully represent real world code. For instance, DaCapo applications were selected to be diverse real applications and ease of use, but they "excluded GUI applications since they are difficult to benchmark systematically." Along these lines, Allamanis and Sutton [2013] go one step further and provide a large-scale (14,807) curated corpus of open source JAVA projects.

With the advent of cloud computing, several source code management (SCM) hosting services have emerged, *e.g.*, *GitHub*, *GitLab*, *Bitbucket*, and *SourceForge*. These services allow the developer to work with different SCMs, *e.g.*, Git, Mercurial, Subversion to host their open source projects. These projects are usually taken as a representation of real-world applications. Thus, while not curated corpora, these hosting services are commonly used to conduct empirical studies.

Another dimension to consider when analyzing large codebases, is how relevant the repositories are. Lopes et al. [2017] conducted 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 considering which projects to analyze when mining software repositories.

Baxter et al. [1998] propose a clone detection algorithm using Abstract Syntax Trees, while Rieger and Ducasse propose a visual detection for clones. Yuan and Guo [2011]; Chen et al. instead propose Count Matrix-based approach to detect code clones.

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, number of contributors, total code churn, and number of commits.

2.2 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

[3] https://www.spec.org/jvm2008/

[4] https://www.spec.org/jbb2000/

but challenging tasks. The most common being downloading, storing, parsing, analyzing and properly extracting information from different kinds of artifacts. In this scenario, there are several tools that allows a researcher or developer to query information about software repositories.

Urma and Mycroft [2012] evaluated seven source code query languages[5]: *Java Tools Language* [Cohen and Maman], *Browse-By-Query*[6], *SOUL* [De Roover et al., 2011], *JQuery* [Volder, 2006], *.QL* [de Moor et al., 2007], *Jackpot*[7], and *PMD*[8]. They have implemented — whenever possible — four use cases using the tools mentioned above. They concluded that only *SOUL* and *.QL* have the minimal features to implement all their use cases.

Dyer et al. [2013a,b] built *Boa*, both a domain-specific language and an online platform[9]. It is used to query software repositories on two popular hosting services, *GitHub* and *SourceForge*. The same authors of *Boa* conducted a study on how new JAVA features, *e.g.*, *Assertions*, *Enhanced-For Loop*, *Extends Wildcard*, were adopted by developers over time [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*[10] 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 — before named *.QL* —, an object-oriented domain-specific language to query recursive data structures Avgustinov et al. [2016].

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[11] was unavailable for testing.

Bajracharya et al. [2009] provide a tool to query large code bases by extracting the source code into a relational model. Sourcegraph[12] is a tool that allows regular expression and diff searches. It integrates with source repositories to ease navigate software projects.

Posnett et al. [2010] have extended ASM [Bruneton et al., 2002; Kuleshov, 2007] to detect meta-patterns, *i.e.*, purely structural patterns of object-oriented interaction. 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.

[5] https://wiki.openjdk.java.net/display/Compiler/Java+Corpus+Tools
[6] http://browsebyquery.sourceforge.net/
[7] http://wiki.netbeans.org/Jackpot
[8] https://pmd.github.io/
[9] http://boa.cs.iastate.edu/
[10] https://lgtm.com/
[11] http://www.spencer-t.racing/datasets
[12] https://sourcegraph.com

[2009] built *Rascal*, a DSL that aims to bring them together by querying the AST of a program.

As its name suggests, JavaParser[13] is a parser for JAVA. The main issue with JavaParser is the lack to do symbol resolution integrated with the project dependencies.

2.3 Large-scale Codebase Empirical Studies

In the same direction as our plan, Callaú et al. [2013] performed an empirical study to assess how much the dynamic and reflective features of SMALLTALK are actually used in practice. Analogously, Richards et al. [2010, 2011]; Wei et al. [2016] conducted 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 projects by popularity, measured by how many times was *starred* in *GitHub*. We argue that it is more important to analyze projects that are *representative*, not *popular*.

Gorla et al. [2014] mined a large set of Android applications, clustering applications by their description topics and identifying outliers in each cluster with respect to their API usage. Grechanik et al. [2010] also mined large scale software repositories to obtain several statistics on how source code is actually written.

For JAVA, Dietrich et al. [2017a] conducted a study about how programmers use contracts in *Maven Central*[14]. Dietrich et al. [2014] have studied how API changes impact JAVA programs. They have used the Qualitas Corpus [Tempero et al., 2010] mentioned above for their study.

Tufano et al. [2015, 2017] studied when code smells are introduced in source code. Palomba et al. [2015] contribute a dataset of five types of code smells together with a systematic procedure for validating code smell datasets. Palomba et al. [2013] propose to detect code smells using change history information.

Nagappan et al. [2015] conducted a study on how the `goto` statement is used in C. They used *GitHub* as a data source for C programs. They concluded that `goto` statements are most used for *handling errors* and *cleaning up resources*.

Static vs. Dynamic Analysis. Given the dynamic nature of JAVASCRIPT, most of the studies mentioned above for JAVASCRIPT perform dynamic analysis. However, Callaú et al. [2013] uses static analysis to study a dynamically checked language. For JAVA, most empirical studies use static analysis. This is due the fact of the availability of input data. Finding valid input data for test cases is not a trivial task, even less to make it scale. For JAVASCRIPT,

[13] `http://javaparser.org/`

[14] `http://central.sonatype.org/`

having a big corpus of web-sites generating valid input data makes more feasible to implement dynamic analysis.

Exceptions

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

Kechagia and Spinellis [2014] analyzed how undocumented and unchecked exceptions cause most of the exceptions in Android applications.

Programming Language Features

Programming language design has been always a hot topic in computer science literature. It has been extensively studied in the past decades. There is a trend in incorporating programming features into mainstream object-oriented languages, *e.g.*, lambdas in JAVA 8[15], C++11[16] and C# 3.0[17]; or parametric polymorphism, *i.e.*, generics, in JAVA 5.[18,19] For instance, JAVA generics were designed to extend JAVA's type system to allow "a type or method to operate on objects of various types while providing compile-time type safety" [Gosling et al.]. However, it was later shown [Amin and Tate, 2016] that compile-time type safety was not fully achieved.

Mazinanian et al. [2017] and Uesbeck et al. [2016] studied how developers use lambdas in JAVA and C++ respectively. 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 does 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 found that in most cases there is an alternative usage that improves performance.

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.

2.3.1 Unsafe Intrinsics in JAVA

Oracle provides the `sun.misc.Unsafe` class for low-level programming, *e.g.*, synchronization primitives, direct memory access methods, array manipulation and memory usage. Although the `sun.misc.Unsafe` class is not officially documented, it is being used in both industrial

[15] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27
[16] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf
[17] https://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic7
[18] https://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html
[19] http://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf

applications and research projects [Korland et al., 2010; Pukall et al.; Gligoric et al., 2011] outside the JDK, compromising the safety of the JAVA ecosystem.

Oracle software engineer Paul Sandoz performed an informal analysis of Maven artifacts and usages in Grepcode [Sandoz, 2015] and conducted a unscientific user survey to study how *Unsafe* is used [Sandoz, 2014]. The survey consists of 7 questions[20] that help to understand what pieces of `sun.misc.Unsafe` should be mainstreamed. In our work [Mastrangelo et al., 2015] we extend Sandoz' work by performing a comprehensive study of the *Maven Central* software repository to analyze how and when `sun.misc.Unsafe` is being used. This study is summarized in Chapter 3.

Tan et al. [2006] propose a safe variant of JNI. Tan and Croft [2008]; Kondoh and Onodera [2008] conducted an empirical security study to describe a taxonomy to classify bugs when using JNI. Sun and Tan [2014] develop a method to isolate native components in Android applications. Li and Tan [2009] analyze the discrepancy between how exceptions are handled in native code and JAVA.

2.3.2 Casting

Casting operations in JAVA[21] allows the developer to view a reference at a different type as it was declared. The related `instanceof` operator[22] tests whether a reference could be cast to a different type without throwing `ClassCastException`.

Winther [2011] has implemented a path sensitive analysis that allows the developer to avoid casting once a guarded `instanceof` is provided. He proposes four cast categorizations according to their run-time type safety: *Guarded Casts*, *Semi-Guarded Casts*, *Unguarded Casts*, and *Safe Casts*. We plan to refine this categorization to answer our *CRQ*2 (*How and when casts are used?*). This is described in Chapter 4.

Tsantalis et al. [2008] present an Eclipse plug-in that identifies type-checking bad smells, a "variation of an algorithm that should be executed, depending on the value of an attribute". They provide refactoring analysis to remove the detected smells by introducing inheritance and polymorphism. This refactoring will introduce casts to select the right type of the object.

Livshits [2006]; Livshits et al. [2005] "describes an approach to call graph construction for JAVA programs in the presence of reflection." He has devised some common usage patterns for reflection. Most of the patterns use casts. We plan to categorize all cast usages, not only where reflection is used.

Landman et al. [2017] have analyzed the relevance of static analysis tools with respect to reflection. They conducted an empirical study to check how often the reflection API is used in real-world code. They have devised reflection AST patterns, which often involve the use of casts. Finally, they argue that controlled programming experiments on subjects need to be correlated with real-world use cases, *e.g.*, *GitHub* or *Maven Central*.

[20] http://www.infoq.com/news/2014/02/Unsafe-Survey
[21] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.16
[22] https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2

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 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 this 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.

Chapter 3

The JAVA Unsafe API in the Wild

We have analyzed *74GB* of compiled JAVA code, spread over 86,479 JAVA archives, to determine how JAVA's unsafe capabilities are used in real-world libraries and applications. We found that 25% of JAVA bytecode archives depend on unsafe third-party JAVA code, and thus JAVA's safety guarantees cannot be trusted. We identify 14 different usage patterns of JAVA's unsafe capabilities, and we provide supporting evidence for why real-world code needs these capabilities. Our long-term goal is to provide a foundation for the design of new language features to regain safety in JAVA.

We have already published our work on how developers use the `sun.misc.Unsafe` API. For a detailed description of the methodology used to find patterns and the patterns we found please refer to Mastrangelo et al. [2015]. Here we answer *URQ*1 in §3.1, followed by how the patterns we found could be implemented in a safer way §3.2 in response to *URQ*2.

3.1 Is Unsafe Used?

To answer *URQ*1 (*To what extent does the Unsafe API impact common application code?*) we need to determine whether and how Unsafe is actually used in real-world third-party JAVA libraries, and to what degree real-world applications directly and indirectly depend on such unsafe 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 software repository.

Artifacts. In Maven, an artifact is the output of the build procedure of a project. Artifacts are usually *.jar* files, which archive compiled JAVA bytecode stored in *.class* files.

Bytecode Analysis. We use a bytecode analysis library to search for method call sites and field accesses of the `sun.misc.Unsafe` class.

Dependency Analysis. We define the impact of an artifact as how many artifacts depend on it, either directly or indirectly. This helps us to define the impact of artifacts that use `sun.misc.Unsafe`, and thus the impact `sun.misc.Unsafe` has on real-world code overall.

Our analysis found 48,490 uses of `sun.misc.Unsafe` — 48,139 call sites and 351 field accesses — distributed over 817 different artifacts. This initial result shows that Unsafe is indeed used in third-party code.

We use the dependency information to determine the impact of the artifacts that use `sun.misc.Unsafe`. We rank all artifacts according to their impact (the number of artifacts that directly or indirectly depend on them). High-impact artifacts are important; a safety violation in them can affect any artifact that directly or indirectly depends on them. We find that while overall about 1% of artifacts directly use Unsafe, for the top-ranked 1000 artifacts, 3% directly use Unsafe. Thus, Unsafe usage is particularly prevalent in high-impact artifacts, artifacts that can affect many other artifacts.

Moreover, we found that 21,297 artifacts (47% of the 47,127 artifacts with dependency information, or 25% of the 86,479 artifacts we downloaded) directly or indirectly depend on `sun.misc.Unsafe`. Excluding language artifacts, numbers do not change much: Instead of 21,297 artifacts, we found 19,173 artifacts, 41% of the artifacts with dependency information, or 22% of artifacts downloaded. Thus, `sun.misc.Unsafe` usage in third-party code indeed impacts a large fraction of projects.

3.2 What is the Unsafe API Used for?

In response to *URQ*2 (*How and when are Unsafe features used?*), many of the patterns we found indicate that *Unsafe* is used to achieve better performance or to implement functionality not otherwise available in the JAVA language or standard library.

However, many of the patterns described can be implemented using APIs already provided in the JAVA standard library. In addition, there are several existing proposals to improve the situation with *Unsafe* already under development within the JAVA community. Oracle software engineer Paul Sandoz [2014] performed a survey on the OpenJDK mailing list to study how Unsafe is used[1] and describes several of these proposals.

A summary of the patterns with existing and proposed alternatives to *Unsafe* is shown in Table 3.1. The table consists of the following columns: The ***Pattern*** column indicates the name of the pattern. The next three columns indicate whether the pattern could be implemented either as a language feature (***Lang***), virtual machine extension (***VM***), or library extension (***Lib***). The ***Ref*** column indicates that the pattern can be implemented using reflection. A bullet (•) indicates that an alternative exists in the JAVA language or API. A check mark (✓) indicates that there is a proposed alternative for JAVA.

Many JAVA APIs already exist that provide functionality similar to *Unsafe*. Indeed, these APIs are often implemented using *Unsafe* under the hood, but they are designed to be used safely. They maintain invariants or perform runtime checks to ensure that their use of *Unsafe* is safe. Because of this overhead, using *Unsafe* directly should in principle provide better performance at the cost of safety.

For example, the *java.util.concurrent* package provides classes for safely performing atomic operations on fields and array elements, as well as several synchronizer classes. These classes

[1] `http://www.infoq.com/news/2014/02/Unsafe-Survey`

Table 3.1: Patterns and their alternatives. A bullet (•) indicates that an alternative exists in the JAVA language or API. A check mark (✓) indicates that there is a proposed alternative for JAVA.

#	Pattern	Lang	VM	Lib	Ref
1	Allocate an Object without Invoking a Constructor	✓			
2	Process Byte Arrays in Block		✓		
3	Atomic Operations			•	
4	Strongly Consistent Shared Variables			✓	
5	Park/Unpark Threads			•	
6	Update Final Fields				•
7	Non-Lexically-Scoped Monitors	✓			
8	Serialization/Deserialization	✓		•	•
9	Foreign Data Access and Object Marshaling	✓		•	
10	Throw Checked Exceptions without Being Declared	✓			
11	Get the Size of an Object or an Array	✓		✓	
12	Large Arrays and Off-Heap Data Structures	✓		✓	
13	Get Memory Page Size	✓		✓	
14	Load Class without Security Checks	✓		✓	

can be used instead of *Unsafe* to implement atomic operations or strongly consistent shared variables. The standard library class *java.util.concurrent.locks.LockSupport* provides *park* and *unpark* methods to be used for implementing locks. These methods are just thin wrappers around the `sun.misc.Unsafe` methods of the same name and could be used to implement the park pattern. JAVA already supports serialization of objects using the *java.lang.Serializable* and *java.io.ObjectOutputStream* API. The now-deleted JEP 187 Serialization 2.0 proposal[2] [3] addresses some of the issues with JAVA serialization.

Because volatile variable accesses compile to code that issues memory fences, strongly consistent variables can be implemented by accessing volatile variables. However, the fences generated for volatile variables may be stronger (and therefore less performant) than are needed for a given application. Indeed, the *Unsafe Put Ordered* and *Fence* methods were likely introduced to improve performance versus volatile variables. The accepted proposal JEP 193 (Enhanced Volatiles [Lea, 2014]) introduces *variable handles*, which allow atomic operations on fields and array elements.

Many of the patterns can be implemented using the reflection API, albeit with lower performance than with *Unsafe* [Korland et al., 2010]. For example, reflection can be used for accessing object fields to implement serialization. Similarly, reflection can be used in combination with *java.nio.ByteBuffer* and related classes for data marshaling. The reflection API can also be used to write to final fields. However, this feature of the reflection API makes

[2] http://mail.openjdk.java.net/pipermail/core-libs-dev/2014-January/024589.html

[3] http://web.archive.org/web/20140702193924/http://openjdk.java.net/jeps/187

sense only during deserialization or during object construction and may have unpredictable behavior in other cases.

Writing a final field through reflection may not ensure the write becomes visible to other threads that might have cached the final field, and it may not work correctly at all if the VM performs compiler optimizations such as constant propagation on final fields.

Many patterns use *Unsafe* to use memory more efficiently. Using `structs` or packed objects can reduce memory overhead by eliminating object headers and other per-object overhead. JAVA has no native support for `structs`, but they can be implemented with byte buffers or with JNI.[4]

The Arrays 2.0 proposal [Rose, 2012] and the value types proposal [Rose et al., 2014] address the large arrays pattern. Project Sumatra [OpenJDK, 2013] proposes features for accessing GPUs and other accelerators, one of the use cases for foreign data access. Related proposals include JEP 191 [Nutter, 2014], which proposes a new foreign function interface for JAVA, and Project Panama [Rose, 2014], which supports native data access from the JVM.

A *sizeof* feature could be introduced into the language or into the standard library. A use case for this feature includes cache management implementations. A higher-level alternative might be to provide an API for memory usage tracking in the JVM. A page size method could be added to the standard library, perhaps in the *java.nio* package, which already includes *MappedByteBuffer* to access memory-mapped storage.

Other patterns may require JAVA language changes. For instance, the language could be changed to not require methods to declare the exceptions they throw, obviating the need for *Unsafe* in this case. Indeed, there is a long-running debate[5] about the software-engineering benefits of checked exceptions. C#, for instance, does not require that exceptions be declared in method signatures at all. One alternative not requiring a language change is to use JAVA generics instead. Because of type erasure, a checked exception can be coerced unsafely into an unchecked exception and thrown.

Changing the language to support allocation without constructors or non-lexically-scoped monitors is feasible. However, implementation of these features must be done carefully to ensure object invariants are properly maintained. In particular, supporting arbitrary unconstructed objects can require type system changes to prevent usage of the object before initialization [Qi and Myers, 2009]. Limiting the scope of this feature to support deserialization only may be a good compromise and has been suggested in the JEP 187 Serialization 2.0 proposal.

Since *Unsafe* is often used simply for performance reasons, virtual machine optimizations can reduce the need for *Unsafe*. For example, the JVM's runtime compiler can be extended with optimizations for vectorizing byte array accesses, eliminating the motivation to use *Unsafe* to process byte arrays. Many patterns use *Unsafe* to use memory more efficiently. This could be ameliorated with lower GC overhead. There are proposals for this, for instance JEP 189 Shenandoah: Low Pause GC [Christine H. Flood, 2014].

[4] `http://www.oracle.com/technetwork/java/jvmls2013sciam-2013525.pdf`

[5] `http://www.ibm.com/developerworks/library/j-jtp05254/`

Chapter 4

Casting Operations in the Wild

Casting operations provide the means to escape the static type system. *But do they pose a problem for developers?* Several studies [Kechagia and Spinellis, 2014; Coelho et al., 2015; Zhitnitsky, 2016] show that `ClassCastException` is in top 10 of exceptions being thrown when analysing stack traces. To illustrate the sort of problem developers have when applying casting conversions, we performed a simple search for commits including the term `ClassCastException` on *GitHub*. The search returns about 150K results.[1] We have included here a few source code results as an example[2]

Forgotten Guard. The following listing[3] shows a cast that throws `ClassCastException` because the developer forgot to include a guard. In this case, the developer fixed the error by introducing a guard on the cast with `instanceof`.

```
@@ -41,6 +41,8 @@ public SCMTypeColumn() {
   }
       public String getScmType(@SuppressWarnings("rawtypes") Job job) {
+        if(!(job instanceof AbstractProject<?, ?>))
+            return "";
       AbstractProject<?, ?> project = (AbstractProject<?, ?>) job;
       return project.getScm().getDescriptor().getDisplayName();
   }
```

Wrong Cast Target. In the next example[4] the `CustomFileFilter` is an inner static class inside `JCustomFileFilter`. Notice the cast happens inside an `equals` method, where this idiom is well known. But the developer has used the outer — wrong — class to cast to.

```
@@ -156,7 +156,7 @@ public boolean equals(Object obj) {
  if (getClass() != obj.getClass()) {
      return false;
  }
- final JCustomFileChooser other = (JCustomFileChooser) obj;
+ final CustomFileFilter other = (CustomFileFilter) obj;
```

[1] `https://github.com/search?l=Java&q=ClassCastException&type=Commits`

[2] To easily spot what the developer has changed to fix the `ClassCastException`, we present each source code excerpt using the Git commit *diff* as reported by *GitHub*.

[3] `https://github.com/jenkinsci/extra-columns-plugin/commit/02d10bd1fcbb2e656da9b1b4ec54208b0cc1cbb2`

[4] `https://github.com/GoldenGnu/jeveassets/commit/5f4750bc8cfa7eed8ad01efd8add2cd2cc9bd831`

```
if (!Objects.equals(this.extensions, other.extensions)) {
    return false;
}
```

Generic Type Inference Mismatch. In the following listing,[5] the *dynamic* property "peer.p2p.pingInterval" (lines 5 and 6) has type int. To fix the error, the developer only changed the type of the literal 5: from long to int.

```
1 @@ -281,7 +281,7 @@ private void startTimers() {
2         } catch (Throwable t) {
3             logger.error("Unhandled exception", t);
4         }
5 -    }, 2, config.getProperty("peer.p2p.pingInterval", 5L), TimeUnit.SECONDS);
6 +    }, 2, config.getProperty("peer.p2p.pingInterval", 5), TimeUnit.SECONDS);
7 }
```

Looking at the definition of the getProperty method below,[6] it obtains a dynamic property given a property name. If it finds a value, return it. Otherwise, returns the default value (second argument). But the return type of getProperty is a generic type inferred by the type of the default value, in this case, long. The ClassCastException is then thrown in line 5, when casting java.lang.Integer to java.lang.Long. To then fix the bug, the developer changed the type of the literal: from long to int.

```
1 public <T> T getProperty(String propName, T defaultValue) {
2     if (!config.hasPath(propName)) return defaultValue;
3     String string = config.getString(propName);
4     if (string.trim().isEmpty()) return defaultValue;
5     return (T) config.getAnyRef(propName);
6 }
```

This indicates that casts represents a source of errors for developers. We present here our partial results for the cast study. First we give an overview of the study in §4.1, while §4.2 gives an estimation of how often a cast operator is used. Finally, §4.3 introduces the methodology we plan to use to discover cast usage patterns.

4.1 Overview of our Study

We propose to answer the following question: *How and when do developers need to escape the type system?* The cast operator in JAVA provides the means to view a reference at a different type as it was declared. Upcasts conversions are done automatically by the compiler. In the case of downcasts, a check is inserted at run-time to verify that the conversion is sound, thus escaping the type system. *Why is so?* Therefore, we believe we should care about how the casting operations are used in the wild. Specifically, we want to answer the following research questions:

[5] https://github.com/ethereum/ethereumj/commit/224e65b9b4ddcb46198a6f8faf69edc65d34d382

[6] https://github.com/ethereum/ethereumj/blob/224e65b9b4ddcb46198a6f8faf69edc65d34d382/ethereumj-core/src/main/java/org/ethereum/config/SystemProperties.java#L312

*CRQ*1 : **How frequently is casting used in common application code?** We want to understand to what extent application code actually uses casting operations.

*CRQ*2 : **How and when casts are used?** If casts are actually used in application code, we want to know how and why developers need to escape the type system.

*CRQ*3 : **How recurrent are the patterns for which casts are used?** In addition to understand how and why casts are used, we want to measure how often developers need to resort to certain idioms to solve a particular problem.

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

Source Code Analysis. We have implemented our study using the QL query language: "a declarative, object-oriented logic programming language for querying complex, potentially recursive data structures encoded in a relational data model" [Avgustinov et al., 2016]. QL allows us to analyze programs at the source code level by abstracting the code sources into a Datalog model. Besides providing structural data for programs, *i.e.*, ASTs, QL has the ability to query static types and perform data-flow analysis. To run our QL queries, we have used the service provided by Semmle.[7]

Projects. As a code base representative of the "real world", we have chosen open-source projects hosted in *GitHub*, the world-most popular source code management repository. So far, we have analyzed 24 JAVA projects in *lgtm*. We plan to scale up our analysis to the whole *lgtm* project database.

Usage Pattern Detection. After all cast instances are found, we analyze this information to discover usage patterns. QL allows us to automatically categorize cast use cases into patterns. This methodology is described in section 4.3.

Our list of patterns is not exhaustive. Due to the nature of the cast operator, some casts were uncategorized as they would need a whole program analysis, *e.g.*, including libraries in the analysis.

4.2 Is the Cast Operator used?

To answer *CRQ*1 (*How frequently is casting used in common application code?*) we want to know how many cast instances are used in a given project. To this end, we gather the following statistics using QL. We show them here to give an estimation of the size of the code base being analized. As mentioned above, these results are preliminary. We plan to scale up our analysis to the whole *lgtm* project database.

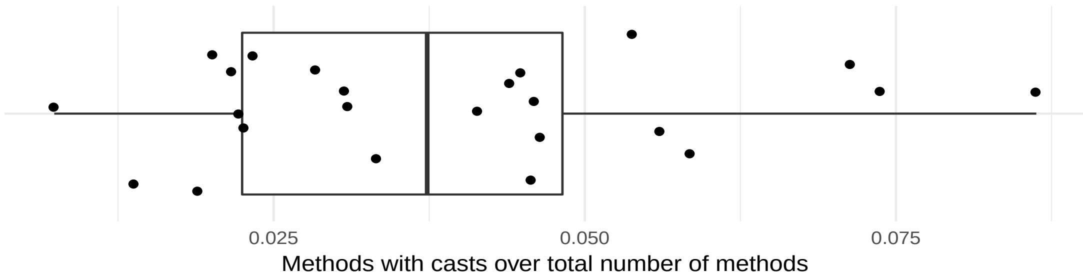
[7] `https://lgtm.com/`

Description	Value
Number of Projects	24
Number of LOC	1,439,913
Number of Methods	121,665
Number of Methods *w/*Cast	6,091
Number of Exprs	4,324,652
Number of Casts	8,627

The *Number of Methods* and *Number of Methods w/Cast* values includes only methods with a body, *i.e.*, not abstract, nor native. The *Number of Exprs* value show how many expressions there are in the ASTs of all source code analyzed. Finally, the *Number of Casts* value indicates how many cast expressions (subtype of `Expr` as defined by QL) were found.

For our study, we are interested in both upcasts and downcasts. Thus, we *exclude* primitive conversions in our study (§5.1.2, §5.1.3, §5.1.4, and §5.1.13 from the JAVA Language Specification[8]). The *Number of Casts* value shown above include only reference conversions. Primitive conversions are always safe (in terms of throwing `ClassCastException`. A primitive conversion happens when both the type of the expression to be casted to and the type to cast to are primitive types. Note that with this definition, we include in our study *boxed* types. Since boxed types are reference types (and therefore not necessarily safe) we want to include them in our analysis.

We want to know how many cast instances there are across projects. Thus, we have computed the ratio between methods containing at least a cast over total number of methods — with implementation — in a given project. The following chart shows this ratio for all analyzed projects:



All projects have less than 10% of methods with at least a cast. Overall, around a 3.92% of methods contain at least one cast operation. This means there is a low density of casts. Given the fact that generics were introduced JAVA 5, this can explain this low density.

Nevertheless, casts are still used. We want to understand why there are casts instances (*CRQ*2) and how often the use cases that leads to casts are used (*CRQ*3). The following sections give an answer to these questions.

4.3 Finding Casts Usage Patterns

To answer both research questions *CRQ*2 (*How and when casts are used?*) and *CRQ*3 (*How recurrent are the patterns for which casts are used?*) we have used the QL query language

[8]https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

within the *lgtm* service to look for cast instances. As mentioned in section 4.2, QL treats primitive conversions as casts. Thus, a preliminary step is to exclude them as cast instances. The following QL query shows how to retrieve all relevant cast expressions:

```
import java
from CastExpr ce where not (
ce.getExpr().getType() instanceof PrimitiveType and
ce.getTypeExpr().getType() instanceof PrimitiveType
) select ce
```

Listing 4.1: QL query to retrieve all relevant cast expressions.

Figure 4.1 depicts our methodology. We have used this initial result as a starting point for our analysis. Afterwards, we select a random sample for manual inspection. We manually inspected the mentioned casts trying to understand why and how they were used.

By manually inspecting several casts instances, we observe that certain characteristics appear often, *e.g.*, a cast in a overridden method, or a cast guarded by an `instanceof`. We then *tag* cast instances based on these observations. We implement a QL predicate that detects them and proceed to refine our query with this new tag predicate. After a new tag is added, the query is run again to iterate over the new results.

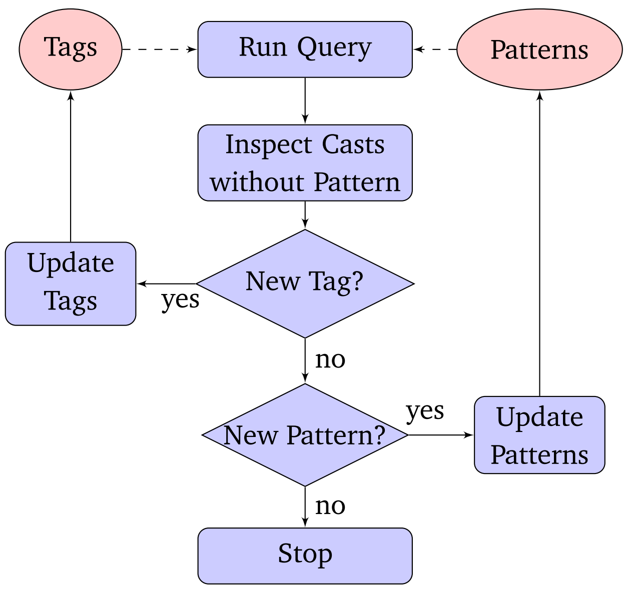


Figure 4.1: Process to discover cast tags and patterns.

Whenever we detect that those tags appear often, we further inspect the source code to check that is indeed a pattern. We have formalized the structure of each pattern as a QL predicate based on those tags. Similarly with tags, after a new pattern is added, the query is run again to inspect the casts without pattern. To sum up, our methodology iterates over the results until no *more* patterns can be detected. The final QL query is available online.[9]

Manual Categorization of Patterns

Some code patterns might be too difficult to express in terms of QL queries. This situation arises when the knowledge to determine the pattern is outside the source code, *e.g.*, in configuration files or library call sites. Thus, in those cases we can only acknowledge that a pattern exists, but not how recurrent it is.

[9] `https://gitlab.com/acuarica/java-cast-queries/blob/master/obs.ql`

Chapter 5

Conclusions

In this proposal we have presented our research plan. We have devised common usage patterns for the JAVA Unsafe API. We discussed several current and future alternatives to improve the JAVA language. This work has been published in [Mastrangelo et al., 2015]. On the other hand, we plan to complement our Unsafe API study with our casting study. We are devising common usage patterns that involve the casting operator. Having a taxonomy of usage patterns — for both the Unsafe API and casting — can shed light on how JAVA developers circumvent the static type system's constraints.

Bibliography

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-2936-1 978-1-4799-0345-0. doi: 10.1109/MSR.2013.6624029.

Nada Amin and Ross Tate. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984004.

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.

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.

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.

S. Bajracharya, J. Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Tools and Evaluation 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure*, pages 1–4, May 2009. doi: 10.1109/SUITE.2009.5070010.

I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No.*

98CB36272), pages 368–377, Bethesda, MD, USA, 1998. IEEE Comput. Soc. ISBN 978-0-8186-8779-2. doi: 10.1109/ICSM.1998.738528.

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.

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.

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.

Xiliang Chen, Alice Yuchen Wang, and Ewan Tempero. A Replication and Reproduction of Code Clone Detection Studies. page 10.

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.

Roman Kennke Christine H. Flood. JEP 189: Shenandoah: An Ultra-Low-Pause-Time Garbage Collector. 2014.

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.

Tal Cohen and Itay Maman. JTL – the Java Tools Language. page 20.

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.

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.

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.

Oege de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, September 2007. doi: 10.1109/SCAM.2007.31.

Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 71–80, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093168.

Sylvia Dieckmann and Urs Hölzle. A study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In Rachid Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 92–115. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48743-2.

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, 2017a. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-035-4. doi: 10.4230/LIPIcs.ECOOP.2017.9.

Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. XCorpus – An executable Corpus of Java Programs. *The Journal of Object Technology*, 16(4):1:1, 2017b. ISSN 1660-1769. doi: 10.5381/jot.2017.16.4.a1.

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.

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.

Justin E Forrester and Barton P Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. page 10, 2000.

Milos Gligoric, Darko Marinov, and Sam Kamin. CoDeSe: Fast Deserialization via Code Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 298–308, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. doi: 10.1145/2001420.2001456.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking App Behavior Against App Descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568276.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® Language Specification. page 670.

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.

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.

Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An Empirical Investigation into a Large-scale Java Open Source Code Repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0039-1. doi: 10.1145/1852786.1852801.

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.

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.

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.

Maria Kechagia and Diomidis Spinellis. Undocumented and Unchecked: Exceptions That Spell Trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 312–315, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597089.

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.

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.

Goh Kondoh and Tamiya Onodera. Finding Bugs in Java Native Interface Programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 109–118, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390645.

Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with Java STM. January 2010.

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.

Doug Lea. JEP 193: Enhanced Volatiles. 2014.

Siliang Li and Gang Tan. Finding Bugs in Exceptional Situations of JNI Programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653716.

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

Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 139–160. Springer, Berlin, Heidelberg, November 2005. ISBN 978-3-540-29735-2 978-3-540-32247-4. doi: 10.1007/11575467_11.

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.

Magnus Madsen and Esben Andreasen. String Analysis for Dynamic Field Access. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Albert Cohen, editors, *Compiler Construction*, volume 8409, pages 197–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54806-2 978-3-642-54807-9. doi: 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.

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.

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.

Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990. ISSN 00010782. doi: 10.1145/96267.96279.

Barton P. Miller, David Koski, Cjin Pheow, Lee Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

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.

Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An Empirical Study of Goto in C Code from GitHub Repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 404–414, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786834.

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.

Charles Oliver Nutter. JEP 191: Foreign Function Interface. 2014.

OpenJDK. Project Sumatra. 2013.

F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Landfill: An Open Dataset of Code Smells with Public Evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485, May 2015. doi: 10.1109/MSR.2015.69.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, Silicon Valley, CA, USA, November 2013. IEEE. ISBN 978-1-4799-0215-6. doi: 10.1109/ASE.2013.6693086.

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.

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.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 978-0-262-16209-8.

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.

M Pukall, C Kaestner, W Cazzola, S Goetz, A Grebhahn, and R Schroeter. Flexible Dynamic Software Updates of Java Applications: Tool Support and Case Study. page 39.

Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480890.

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.

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.

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.

Matthias Rieger and Stephane Ducasse. Visual Detection of Duplicated Code. page 6.

John Rose, Brian Goetz, and Guy Steele. State of the Values. 2014.

John R. Rose. Arrays 2.0. 2012.

John R. Rose. The isthmus in the VM. 2014.

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.

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.

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.

Paul Sandoz. Safety Not Guaranteed: Sun.misc.Unsafe and the quest for safe alternatives. 2014. Oracle Inc. [Online; accessed 29-January-2015].

Paul Sandoz. Personal communication. 2015.

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.

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.

Mengtao Sun and Gang Tan. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 165–176, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2972-9. doi: 10.1145/2627393.2627396.

Gang Tan and Jason Croft. An Empirical Security Study of the Native Code in the JDK. /paper/An-Empirical-Security-Study-of-the-Native-Code-in-Tan-Croft/4c3a84729bd09db6a90a862846bb29e937ec2ced, 2008.

Gang Tan, Srimat Chakradhar, Raghunathan Srivaths, and Ravi Daniel Wang. Safe Java native interface. In *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

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. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, April 2008. doi: 10.1109/CSMR.2008.4493342.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, November 2017. ISSN 0098-5589, 1939-3520. doi: 10.1109/TSE.2017.2653105.

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.

Raoul-Gabriel Urma and Alan Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools - PLATEAU '12*, page 35, Tucson, Arizona, USA, 2012. ACM Press. ISBN 978-1-4503-1631-6. doi: 10.1145/2414721.2414728.

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.

Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In *In Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006*, pages 88–102. Springer, 2006.

Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. Empirical study of the dynamic behavior of JavaScript objects. *Software: Practice and Experience*, 46(7):867–889, July 2016. ISSN 1097-024X. doi: 10.1002/spe.2334.

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.

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.

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.

Yang Yuan and Yao Guo. CMCD: Count matrix based code clone detection. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 250–257, December 2011. doi: 10.1109/APSEC.2011.13.

Alex Zhitnitsky. The Top 10 Exception Types in Production Java Applications - Based on 1B Events. https://blog.takipi.com/the-top-10-exceptions-types-in-production-java-applications-based-on-1b-events/, June 2016.