When and How Java Developers Give Up Static Type Safety

Subtitle: Reinventing the World

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Luis Mastrangelo

under the supervision of
Prof. Matthias Hauswirth and Prof. Nathaniel Nystrom

March 2019

Dissertation Committee

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

Dissertation accepted on First March 2019

Research Advisor	Co-Advisor
Prof. Matthias Hauswirth	**Prof. Nathaniel Nystrom**
Ph.D. Program Director	Ph.D. Program Director
Prof. Walter Binder	**Prof. Olaf Schenk**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luis Mastrangelo
Lugano, First March 2019

To my beloved

Someone said . . .

Someone

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.

Acknowledgements

acknowledgements

Contents

Figures

Tables

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, 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

[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`

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

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 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

[5] https://kotlinlang.org/docs/reference/typecasts.html#smart-casts
[6] https://www.tiobe.com/tiobe-index/

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.

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.

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

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

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.

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

[3] `https://www.spec.org/jvm2008/`
[4] `https://www.spec.org/jbb2000/`

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 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

[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/

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. [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

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

[11] `http://www.spencer-t.racing/datasets`

[12] `https://sourcegraph.com`

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

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, 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

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

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 applications and research projects [Korland

[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`

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.

[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`

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*.

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

The JAVA Virtual Machine (JVM) executes JAVA bytecode and provides other services for programs written in many programming languages, including JAVA, SCALA, and CLOJURE. The JVM was designed to provide strong safety guarantees. However, many widely used JVM implementations expose an API that allows the developer to access low-level, unsafe features of the JVM and underlying hardware, features that are unavailable in safe JAVA bytecode. This API is provided through an undocumented[1] class, `sun.misc.Unsafe`, in the JAVA reference implementation produced by Oracle.

Other virtual machines provide similar functionality. For example, the C# language provides an `unsafe` construct on the .NET platform,[2] and RACKET provides unsafe operations.[3]

The operations `sun.misc.Unsafe` provides can be dangerous, as they allow developers to circumvent the safety guarantees provided by the JAVA language and the JVM. If misused, the consequences can be resource leaks, deadlocks, data corruption, and even JVM crashes.[4] [5] [6] [7] [8]

We believe that `sun.misc.Unsafe` was introduced to provide better performance and more capabilities to the writers of the JAVA runtime library. However, `sun.misc.Unsafe` is increasingly being used in third-party frameworks and libraries. Application developers who rely on JAVA's safety guarantees have to

[1] `http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html`
[2] `https://msdn.microsoft.com/en-us/en-en/library/chfa2zb8(v=vs.90).aspx`
[3] `http://docs.racket-lang.org/reference/unsafe.html`
[4] `https://groups.google.com/d/msg/elasticsearch/Nh-kXI5J6Ek/WXIZKhhGVHkJ`
[5] `https://github.com/EsotericSoftware/kryo/issues/219`
[6] `https://github.com/dain/snappy/issues/24`
[7] `https://netbeans.org/bugzilla/show_bug.cgi?id=229655`
[8] `https://netbeans.org/bugzilla/show_bug.cgi?id=244914`

trust the implementers of the language runtime environment (including the core runtime libraries). Thus the use of `sun.misc.Unsafe` in the runtime libraries is no more risky than the use of an unsafe language to implement the JVM. However, the fact that more and more "normal" libraries are using `sun.misc.Unsafe` means that application developers have to trust a growing community of third-party JAVA library developers to not inadvertently tamper with the fragile internal state of the JVM.

Given that the benefits of safe languages are well known, and the risks of unsafe languages are obvious, why exactly does one need unsafe features in third-party libraries? Are those features used in real-world code? If yes, how are they used, and what are they used for?

We studied a large repository of JAVA code, *Maven Central*, to answer these questions. We have analyzed 74 GB 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 strong foundation to make informed decisions in the future evolution of the JAVA language and virtual machine, and for the design of new language features to regain safety in JAVA.

We have already published our work on how developers use the Unsafe API in JAVA [Mastrangelo et al., 2015]. In this thesis we outline the risks of using the *Unsafe* API in §3.1. Then we answer *URQ*1 in §3.2. To answer *URQ*2, first we introduce our methodology and the patterns we found in §3.3 and §3.4 respectively, to then present how the patterns we found could be implemented in a safer way in §3.5.

3.1 The Risks of Compromising Safety

We outline the risks of *Unsafe* by illustrating how the improper use of *Unsafe* violates JAVA's safety guarantees.

In JAVA, the unsafe capabilities are provided as instance methods of class `sun.misc.Unsafe`. Access to the class has been made less than straightforward. Class `sun.misc.Unsafe` is final, and its constructor is not public. Thus, creating an instance requires some tricks. For example, one can invoke the private constructor via reflection. This is not the only way to get hold of an unsafe object, but it is the most portable.

```
1 Constructor<Unsafe> c = Unsafe.class.getDeclaredConstructor();
2 c.setAccessible(true);
3 Unsafe unsafe = c.newInstance();
```

Listing 3.1. Instantiating an Unsafe object

Given the unsafe object, one can now simply invoke any of its methods to directly perform unsafe operations.

Violating Type Safety

In JAVA, variables are strongly typed. For example, it is impossible to store an int value inside a variable of a reference type. *Unsafe* can violate that guarantee: it can be used to store a value of any type in a field or array element.

```
1 class C {
2   private Object f = new Object();
3 }
4 long fieldOffset = unsafe.objectFieldOffset(
5         C.class.getDeclaredField("f") );
6 C o = new C();
7 unsafe.putInt(o, fieldOffset, 1234567890);      // f now points to nirvana
```

Listing 3.2. `sun.misc.Unsafe` can violate type safety

Crashing the Virtual Machine

A quick way to crash the VM is to free memory that is in a protected address range, for example by calling `freeMemory` as follows.

```
1 unsafe.freeMemory(1);
```

Listing 3.3. `sun.misc.Unsafe` can crash the VM

In JAVA, the normal behavior of a method to deal with such situations is to throw an exception. Being unsafe, instead of throwing an exception, this invocation of `freeMemory` crashes the VM.

Violating Method Contracts

In JAVA, a method that does not declare an exception cannot throw any checked exceptions. *Unsafe* can violate that contract: it can be used to throw a checked exception that the surrounding method does not declare or catch.

```
1 void m() {
2   unsafe.throwException(new Exception());
3 }
```

Listing 3.4. `sun.misc.Unsafe` can violate a method contract

Uninitialized Objects

JAVA guarantees that an object allocation also initializes the object by running its constructor. *Unsafe* can violate that guarantee: it can be used to allocate an object without ever running its constructor. This can lead to objects in states that the objects' classes would not seem to admit.

```
1 class C {
2   private int f;
3   public C() { f = 5; }
4   public int getF() { return f; }
5 }
6
7 C c = (C)unsafe.allocateInstance(C.class);
8 assert c.getF()==5; // violated
```

Listing 3.5. `sun.misc.Unsafe` can lead to uninitialized objects

Monitor Deadlock

JAVA provides synchronized methods and synchronized blocks. These constructs guarantee that monitors entered at the beginning of a section of code are exited at the end. *Unsafe* can violate that contract: it can be used to asymmetrically enter or exit a monitor, and that asymmetry might be not immediately obvious.

```
1 void m() {
2   unsafe.monitorEnter(o);
3   if (c) return;
4   unsafe.monitorExit(o);
5 }
```

Listing 3.6. `sun.misc.Unsafe` can lead to monitor deadlocks

The above examples are just the most straightforward violations of JAVA's safety guarantees. The `sun.misc.Unsafe` class provides a multitude of methods that can be used to violate most guarantees JAVA provides.

To sum it up: *Unsafe* is dangerous. But should anybody care? In the next sections we present a study to determine whether and how *Unsafe* is used in real-world third-party JAVA libraries, and to what degree real-world applications directly and indirectly depend on it.

3.2 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.

Which Features of Unsafe Are Actually Used?

Figures 3.1 and 3.2 show all instance methods and static fields of `sun.misc.Unsafe`. For each member we show how many call sites or field accesses we found across the artifacts. The class provides 120 public instance methods and 20 public fields (version 1.8 update 40). The figure only shows 93 methods because the 18 methods in the *Heap Get* and *Heap Put* groups, and *staticFieldBase* are overloaded, and

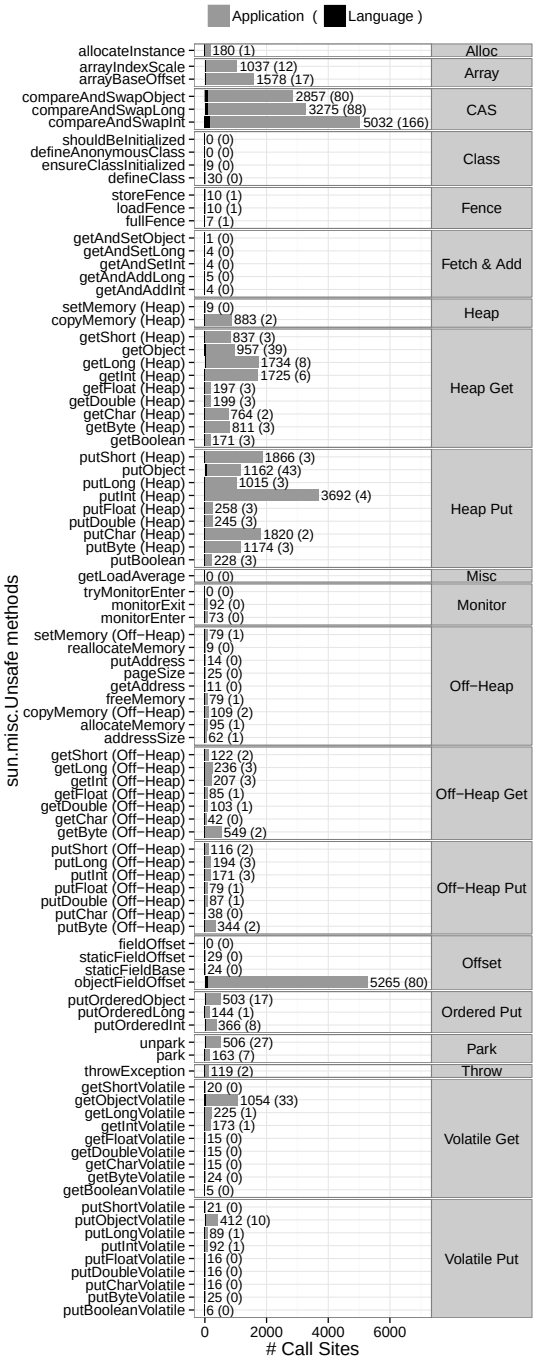


Figure 3.1. `sun.misc.Unsafe` method usage on Maven Central

we combine overloaded methods into one bar.

We show two columns, *Application* and *Language*. The *Language* column

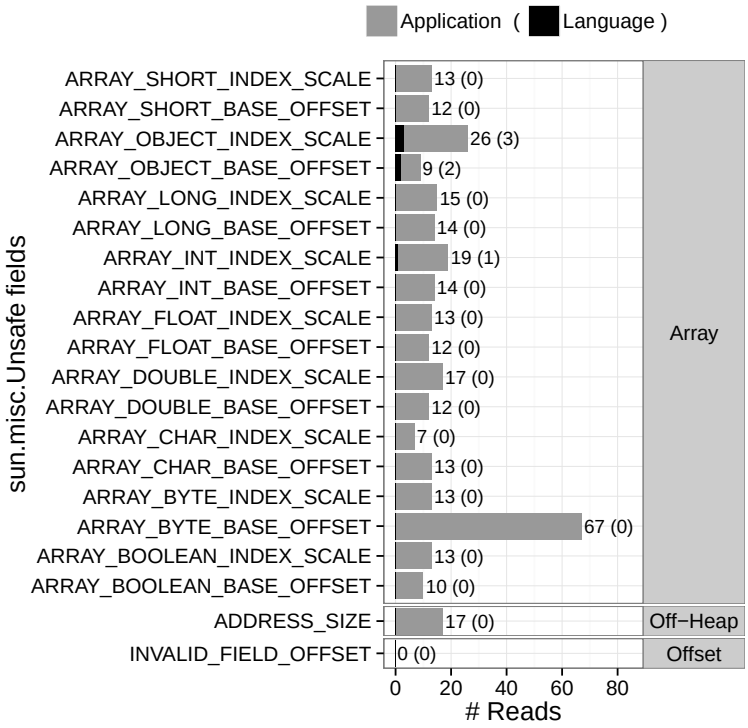


Figure 3.2. `sun.misc.Unsafe` field usage on Maven Central

corresponds to language implementation artifacts while the *Application* column corresponds to the rest of the artifacts.

We categorized the members into groups, based on the functionality they provide:

- The *Alloc* group contains only the *allocateInstance* method, which allows the developer to allocate a JAVA object without executing a constructor. This method is used 181 times: 180 in *Application* and 1 in *Language*.
- The *Array* group contains methods and fields for computing relative addresses of array elements. The fields were added as a simpler and potentially faster alternative in a more recent version of *Unsafe*. The value of all fields in this group are constants initialized with the result of a call to either *arrayBaseOffset* or *arrayIndexScale* in the *Array* group. The figures show that the majority of sites still invoke the methods instead of accessing the corresponding constant fields.
- The *CAS* group contains methods to atomically compare-and-swap a JAVA variable. These operations are implemented using processor-specific atomic

instructions. For instance, on *x86* architectures, *compareAndSwapInt* is implemented using the `CMPXCHG` machine instruction. Figure 3.1 shows that these methods represent the most heavily used feature of *Unsafe*.

- Methods of the *Class* group are used to dynamically load and check JAVA classes. They are rarely used, with *defineClass* being used the most.
- The methods of the *Fence* group provide memory fences to ensure loads and stores are visible to other threads. These methods are implemented using processor-specific instructions. These methods were introduced only recently in JAVA 8, which explains their limited use in our data set. We expect that their use will increase over time and that other operations, such as those in the *Ordered Put*, or *Volatile Put* groups will decrease as programmers use the lower-level fence operations.
- The *Fetch & Add* group, like the *CAS* group, allows the programmer to atomically update a JAVA variable. This group of methods was also added recently in JAVA 8. We expect their use to increase as programmers replace some calls to methods in the *CAS* group with the new functionality.
- The *Heap* group methods are used to directly access memory in the JAVA heap. The *Heap Get* and *Heap Put* groups allow the developer to load and store a Java variable. These groups are among the most frequently used ones in *Unsafe*.
- The *Misc* group contains the method *getLoadAverage*, to get the load average in the operating system run queue assigned to the available processors. It is not used.
- The *Monitor* group contains methods to explicitly manage JAVA monitors. The *tryMonitorEnter* method is never used.
- The *Off-Heap* group provides access to unmanaged memory, enabling explicit memory management. Similarly to the *Heap Get* and *Heap Put* groups, the *Off-Heap Get* and *Off-Heap Put* groups allow the developer to load and store values in Off-Heap memory. The usage of these methods is non-negligible, with *getByte* and *putByte* dominating the rest. The value of the *ADDRESS_SIZE* field is the result of the method *addressSize()*.
- Methods of the *Offset* group are used to compute the location of fields within JAVA objects. The offsets are used in calls to many other `sun.misc.Unsafe` methods, for instance those in the *Heap Get*, *Heap Put*, and the *CAS* groups.

The method *objectFieldOffset* is the most called method in `sun.misc.Unsafe` due to its result being used by many other `sun.misc.Unsafe` methods. The *fieldOffset* method is deprecated, and indeed, we found no uses. The *INVALID_FIELD_OFFSET* field indicates an invalid field offset; it is never used because code using *objectFieldOffset* is not written in a defensive style.

- The *Ordered Put* group has methods to store to a JAVA variable without emitting any memory barrier but guaranteeing no reordering across the store.

- The *park* and *unpark* methods are contained in the *Park* group. With them, it is possible to block and unblock a thread's execution.

- The *throwException* method is contained in the *Throw* group, and allows one to throw checked exceptions without declaring them in the `throws` clause.

- Finally, the *Volatile Get* and *Volatile Put* groups allow the developer to store a value in a JAVA variable with `volatile` semantics.

It is interesting to note that despite our large corpus of code, there are several Unsafe methods that are never actually called. If *Unsafe* is to be used in third-party code, then it might make sense to extract those methods into a separate class to be only used from within the runtime library.

3.3 Finding **sun.misc.Unsafe** Usage Patterns

We examined the artifacts in the Maven Central software repository to identify usage patterns for *Unsafe*. This section describes our methodology for identifying these patterns.

Our first step is to visualize how an artifact uses *Unsafe*. To this end, we count the *Unsafe* call sites and field usages per class in each artifact. Figures 3.3 and 3.4 show two examples of call sites usages for *com.lmax*:*disruptor* and *org.scala-lang*:*scala-library* respectively. Each row shows a fully qualified class name and their usage of `sun.misc.Unsafe`.

After determining the call sites and field usage per artifact, we tried to find a way to group artifacts by how they use `sun.misc.Unsafe`. The first issue is to determine which method calls work together to achieve a goal. These calls might all be located within a single class, be spread across different classes within a

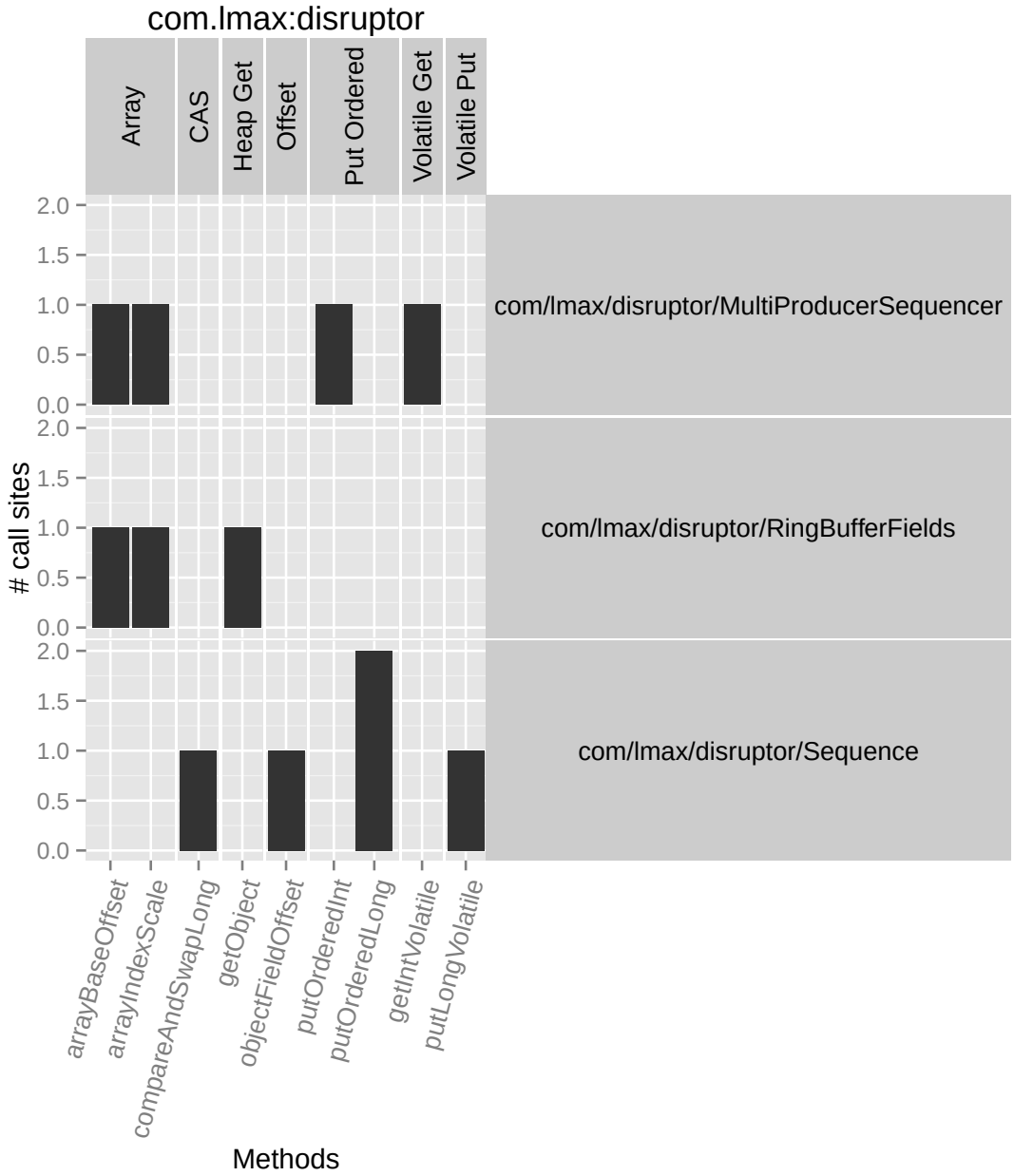


Figure 3.3. com.lmax:disruptor call sites

package, or be spread across different packages within the whole artifact. After trying different combinations, we decided to group together calls occurring within a single class and its inner classes.

We cluster classes and their inner classes by *Unsafe* method usage using a dendrogram. Because a dendrogram can result in different clusters depending on

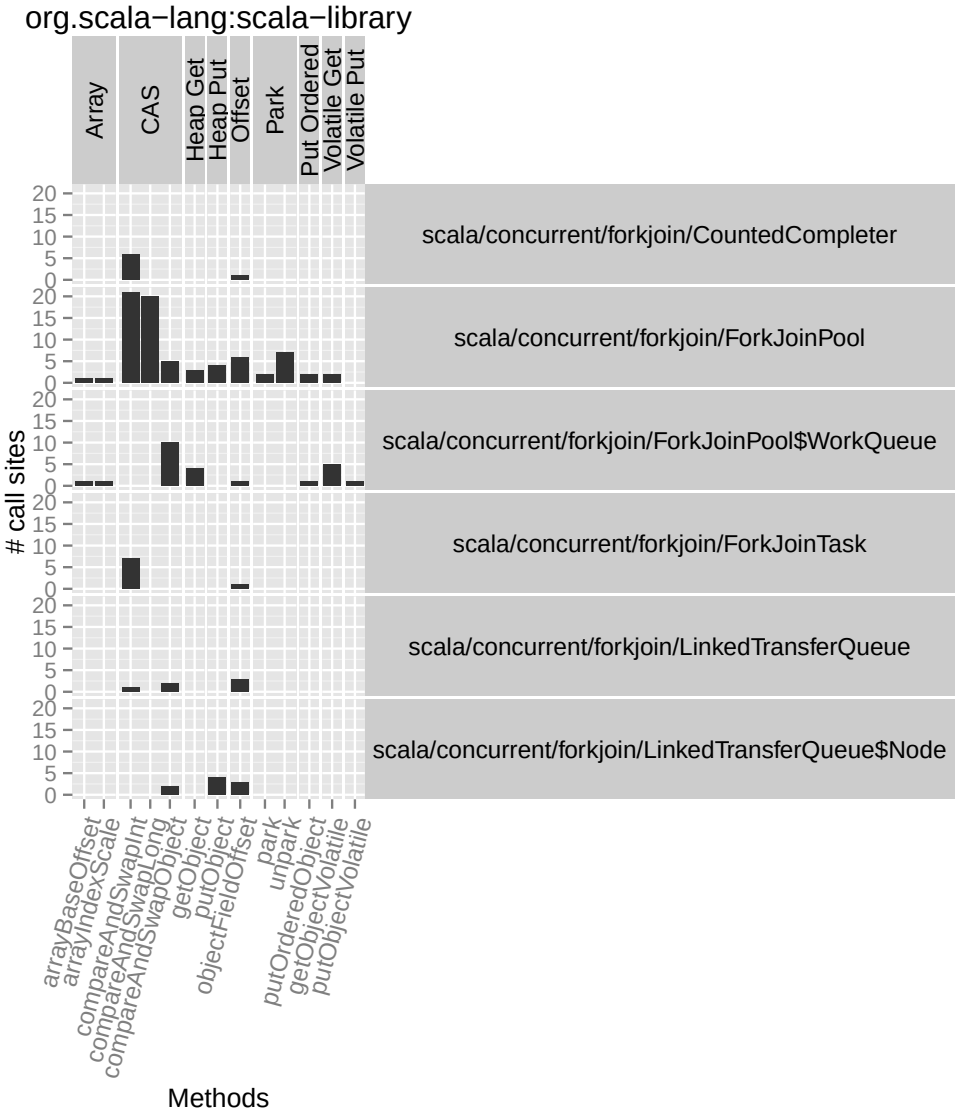


Figure 3.4. org.scala-lang:scala-library call sites

at which height the dendrogram is cut, we experimented with various clusterings until settling on 31 clusters. An example of a cluster and its dendrogram is shown in Figure 3.5. In the figure we can see classes using methods of the *Off-Heap*, *Off-Heap Get*, and *Off-Heap Put* groups to implement large arrays.

Once we had a clustering of the artifacts by method usage, we manually inspected a sample of artifacts in each cluster to identify patterns. Some artifacts contained more than one pattern. For instance the cluster in Figure 3.5 contains classes that use *Unsafe* to implement large off-heap arrays, but also contains calls

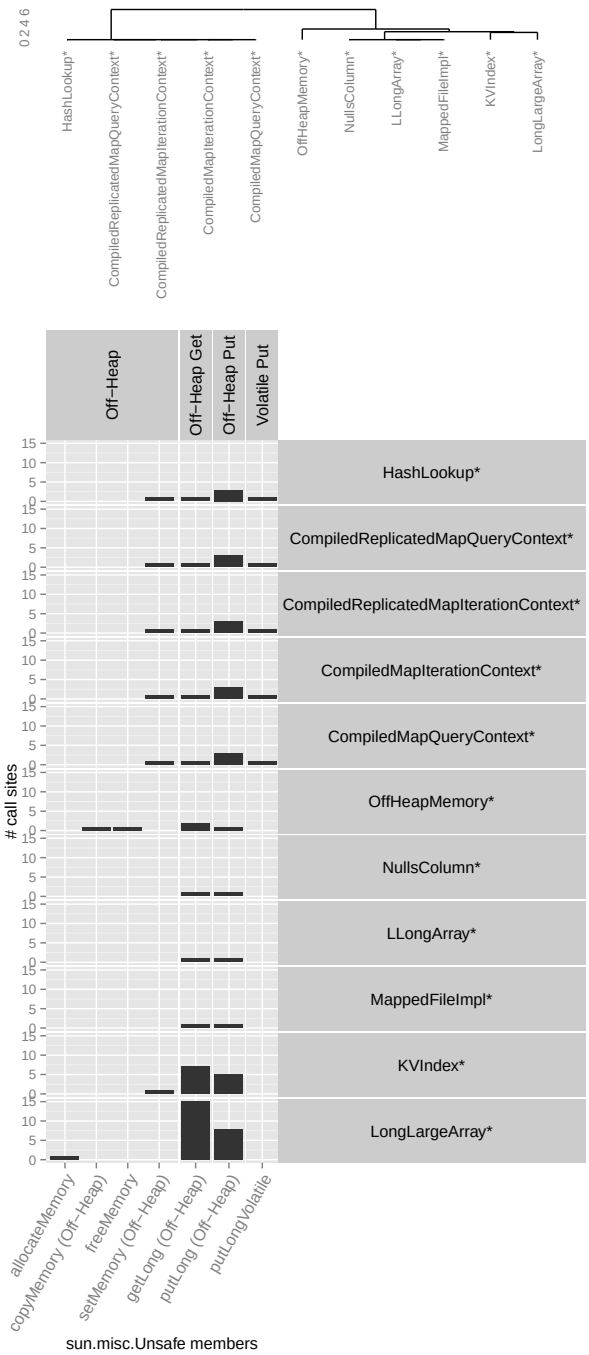


Figure 3.5. Classes using off-heap large arrays

to methods of the *Put Volatile* group used to implement strongly shared consistent variables. We tagged each artifact manually inspected with the set of patterns

that it exhibits.

3.4 Usage Patterns of **sun.misc.Unsafe**

This section presents the patterns we have found during our study. We present them sorted by how many artifacts depend on them, as computed from the Maven dependency graph described in Section 3.2.

A summary of the patterns is shown in Table 3.1. The ***Pattern*** column indicates the name of the pattern. ***Found in*** indicates the number of artifacts in *Maven Central* that contain the pattern. ***Used by*** indicates the number of artifacts that transitively depend on the artifacts with the pattern. ***Most used artifacts*** presents the three most used artifacts containing the pattern, that is, the artifact with the most other artifacts that transitively depend upon it. Artifacts are shown using their Maven identifier, *i.e.* ⟨*groupId*⟩:⟨*artifactId*⟩.

We present each pattern using the following template.

Description. What is the purpose of the pattern? What does it do?
Rationale. What problem is the pattern trying to solve? In what contexts is it used?
Implementation. How is the pattern implemented using `sun.misc.Unsafe`?
Issues. Issues to consider when using the pattern and problems discussed in the Stack Overflow database.

3.4.1 Allocate an Object without Invoking a Constructor

Description. With this pattern an object can be allocated on the heap without executing its constructor.
Rationale. This pattern is useful for creating mock objects for testing and in deserializing serialized objects.
Implementation. The *allocateInstance* method takes as parameter a *java.lang.Class* object, and returns a new instance of that class. Unlike allocating an object directly, or through the reflection API, the object's constructor is not invoked.
Issues. If the constructor is not invoked, the object might be left uninitialized and its invariants might not hold. Users of *allocateInstance* must take care to properly initialize the object before it is used by other code. This is often done in conjunction with other methods of *Unsafe*, for instance those in the *Heap Put* group, or by using the Java reflection API.

Table 3.1. Patterns and their occurrences in the Maven Central repository.

	Pattern	Found In	Used by	Most used artifacts
1	Allocate an Object without Invoking a Constructor	88	14794	*org.springframework:spring-core*, *org.objenesis:objenesis*, *org.mockito:mockito-all*
2	Process Byte Arrays in Block	44	12274	*com.google.guava:guava*, *com.google.gwt:gwt-dev*, *net.jpountz.lz4:lz4*
3	Atomic Operations	84	10259	*org.scala-lang:scala-library*, *org.apache.hadoop:hadoop-hdfs*, *org.glassfish.grizzly:grizzly-framework*
4	Strongly Consistent Shared Variables	198	9795	*org.scala-lang:scala-library*, *org.jruby:jruby-core*, *com.hazelcast:hazelcast-all*
5	Park/Unpark Threads	62	7330	*org.scala-lang:scala-library*, *org.codehaus.jsr166-mirror:jsr166y*, *com.netflix.servo:servo-internal*
6	Update Final Fields	11	7281	*org.codehaus.groovy:groovy-all*, *org.jodd:jodd-core*, *com.lmax:disruptor*
7	Non-Lexically-Scoped Monitors	14	7015	*org.jboss.modules:jboss-modules*, *org.apache.cassandra:cassandra-all*, *org.gridgain:gridgain-core*
8	Serialization/Deserialization	32	5689	*com.hazelcast:hazelcast-all*, *com.esotericsoftware.kryo:kryo*, *com.thoughtworks.xstream:xstream*
9	Foreign Data Access and Object Marshaling	8	3690	*eu.stratosphere:stratosphere-core*, *com.github.jnr:jffi*, *org.python:jython*
10	Throw Checked Exceptions without Being Declared	59	3566	*io.netty:netty-all*, *net.openhft:lang*, *ai.h2o:h2o-core*
11	Get the Size of an Object or an Array	4	3003	*net.sf.ehcache:ehcache*, *com.github.jbellis:jamm*, *org.openjdk.jol:jol-core*
12	Large Arrays and Off-Heap Data Structures	12	487	*org.neo4j:neo4j-primitive-collections*, *com.orientechnologies:orientdb-core*, *org.mapdb:mapdb*
13	Get Memory Page Size	11	359	*org.apache.hadoop:hadoop-common*, *net.openhft:lang*, *org.xerial.larray:larray-mmap*
14	Load Class without Security Checks	21	294	*org.elasticsearch:elasticsearch*, *org.apache.geronimo.ext.openejb:openejb-core*, *net.openhft:lang*

3.4.2 Process Byte Arrays in Block

Description. When processing the elements of a byte array, better performance can be achieved by processing the elements 8 bytes at a time, treating it as a long array, rather than one byte at a time.

Rationale. The pattern is used for fast byte array processing, for instance, when comparing two byte arrays lexicographically.
Implementation. The *arrayBaseOffset* method is invoked to get the base offset of the byte array. Then *getLong* is used to fetch and process 8 bytes of the array at a time.
Issues. The pattern assumes that bytes in an array are stored contiguously. This may not be true for some VMs, *e.g.* those implementing large arrays using discontinuous arrays or arraylets Siebert [2000]; Bacon et al. [2003]. Users of the pattern should be aware of the endianness of the underlying hardware. In one Stack Overflow discussion, this pattern is discouraged since it is non-portable and, on many JVMs, results in slower code.[9]

3.4.3 Atomic Operations

Description. To implement non-blocking concurrent data structures and synchronization primitives, hardware-specific atomic operations provided by `sun.misc.Unsafe` are used.
Rationale. Non-blocking algorithms often scale better than algorithms that use locking.
Implementation. To get the offset of a JAVA variable either *objectFieldOffset* or *arrayBaseOffset*/*arrayIndexScale* can be used. With this offset, the methods from the *CAS* or *Fetch & Add* groups are used to perform atomic operations on the variable. Other methods of *Unsafe* are often used in the implementation of concurrent data structures, including *Volatile Get/Put*, *Ordered Put*, and *Fence* methods.
Issues. Non-blocking algorithms can be difficult to implement correctly. Programmers must understand the Java memory model and how the *Unsafe* methods interact with the memory model.

3.4.4 Strongly Consistent Shared Variables

Description. Because of Java's weak memory model, when implementing concurrent code, it is often necessary to ensure that writes to a shared variable by one thread become visible to other threads, or to prevent reordering of loads and stores. Volatile variables can be used for this purpose, but `sun.misc.Unsafe` can be used instead with better performance. Additionally, because Java does not allow array elements to be declared volatile, there is no possibility other than to

[9] `http://stackoverflow.com/questions/12226123`

use *Unsafe* to ensure visibility of array stores. The methods of the *Ordered Put* groups and the *Volatile Get/Put* groups can be used for these purposes. In addition, the *Fence* methods were introduced in Java 8 expressly to provide greater flexibility for this use case.
Rationale. This pattern is useful for implementing concurrent algorithms or shared variables in concurrent settings. For instance, JRuby uses a *fullFence* to ensure visibility of writes to object fields.
Implementation. To ensure a write is visible to another thread, *Volatile Put* methods or *Ordered Put* methods can be used, even on non-volatile variables. Alternatively, a *storeFence* or *fullFence* can be used. *Volatile Get* methods ensure other loads and stores are not reordered across the load. A *loadFence* could also be used before a read of a shared variable.
Issues. Fences can replace volatile variables in some situations, offering better performance. Most of the uses of the pattern use the *Ordered Put* and *Volatile Put* methods. Since they were added to Java only recently, there are currently few instances of the pattern that use the *Fence* methods.

3.4.5 Park/Unpark Threads

Description. The *park* and *unpark* methods are used to block and unblock threads and are useful for implementing locks and other blocking synchronization constructs.
Rationale. The alternative to parking a thread is to busy-wait, which uses CPU resources and does not allow other threads to proceed.
Implementation. The *park* method blocks the current thread while *unpark* unblocks a thread given as an argument.
Issues. Users of *park* must be careful to avoid deadlock.

3.4.6 Update Final Fields

Description. This pattern is used to update a final field.
Rationale. Although it is possible to use reflection to implement the same behavior, updating a final field is easier and more efficient using `sun.misc.Unsafe`. Some applications update final fields when cloning objects or when deserializing objects.
Implementation. The *objectFieldOffset* methods and one of the *Put* methods work in conjunction to directly modify the memory where a final field resides.
Issues. There are numerous security and safety issues with modifying final fields. The update should be done only on newly created objects (perhaps also using

allocateInstance to avoid invoking the constructor) before the object becomes visible to other threads. The Java Language Specification (Section 17.5.3) Gosling et al. [2013] recommends that final fields not be read until all updates are complete. In addition, the language permits compiler optimizations with final fields that can prevent updates to the field from being observed. Since final fields can be cached by other threads, one instance of the pattern uses *putObjectVolatile* to update the field rather than simply *putObject*. Using this method ensures that any cached copy in other threads is invalidated.

3.4.7 Non-Lexically-Scoped Monitors

Description. In this pattern, monitors are explicitly acquired and released without using `synchronized` blocks.
Rationale. The pattern is used in some situations to avoid deadlock, releasing a monitor temporarily, then reacquiring it.
Implementation. One usage of the pattern is to temporarily release monitor locks acquired in client code (e.g., through a synchronized block or method) and then to reenter the monitor before returning to the client. The *monitorExit* method is used to exit the synchronized block. Because monitors are reentrant, the pattern uses the method *Thread.holdsLock* to implement a loop that repeatedly exits the monitor until the lock is no longer held. When reentering the monitor, *monitorEnter* is called the same number of times as *monitorExit* was called to release the lock.
Issues. Care must be taken to balance calls to *monitorEnter* and *monitorExit*, or else the lock might not be released or an `IllegalMonitorStateException` might be thrown.

3.4.8 Serialization/Deserialization

Description. In this pattern, `sun.misc.Unsafe` is used to persist and subsequently load objects to and from secondary memory dynamically. Serialization in JAVA is so important that it has a *Serializable* interface to automatically serialize objects that implement it. Although this kind of serialization is easy to use, it does not offer good performance and is inflexible. It is possible to implement serialization using the reflection API. This is also expensive in terms of performance. Therefore, fast serialization frameworks often use *Unsafe* to get and set fields of objects. Some of these projects use reflection to check if `sun.misc.Unsafe` is available, falling back on a slower implementation if not.

Rationale. De/serialization requires reading and writing fields to save and restore objects. Some of these fields may be final or private.
Implementation. Methods of *Heap Get* and *Heap Put* are used to read and write fields and array elements. Deserialization may use *allocateInstance* to create objects without invoking the constructor.
Issues. Using *Unsafe* for serialization and deserialization has many of the same issues as using *Unsafe* for updating final fields (Section 3.4.6) and for creating objects without invoking a constructor (Section 3.4.1). Objects must not escape before being completely deserialized. Type safety can be violated by using methods of the *Heap Put* group. In addition, care must be taken when deserializing some data structures. For instance, data structures that use *System.identityHashCode* or *Object.hashCode* may need to rehash objects on deserialization because the deserialized object might have a different hash code than the original serialized object.

3.4.9 Foreign Data Access and Object Marshaling

Description. In this pattern `sun.misc.Unsafe` is used to share data between Java code and code written in another language, usually C or C++.
Rationale. This pattern is needed to efficiently pass data, especially structures and arrays, back and forth between Java and native code. Using this pattern can be more efficient than using native methods and JNI.
Implementation. The methods of the *Off-Heap* group are used to access memory off the Java heap. Often a buffer is allocated using *allocateMemory*, which is then passed to the other language using JNI. Alternatively, the native code can allocate a buffer in a JNI method. The *Off-Heap Get* and *Off-Heap Put* methods are used to access the buffer.
Issues. Use of *Unsafe* here is inherently not type-safe. Care must be taken especially with native pointers, which are represented as `long` values in Java code.

3.4.10 Throw Checked Exceptions without Being Declared

Description. This pattern allows the programmer to throw checked exceptions without being declared in the method's `throws` clause.
Rationale. In testing and mocking frameworks, the pattern is used to circumvent declaring the exception to be thrown, which is often unknown. It is used in the Java Fork/Join framework to save the generic exception of a thread to be rethrown later.
Implementation. The pattern is implemented using the *throwException* method.

Issues. This method can violate Java's subtyping relation, because it is not expected for a method that does not declare an exception to actually throw it. At run time, this can manifest as an uncaught exception.

3.4.11 Get the Size of an Object or an Array

Description. This pattern uses `sun.misc.Unsafe` to estimate the size of an object or an array in memory.

Rationale. The object size can be useful for making manual memory management decisions. For instance, when implementing a cache, object sizes can be used to implement code to limit the cache size.

Implementation. To compute the size of an array, add *arrayBaseOffset* and *arrayIndexScale* (for the given array base type) times the array length. For objects, use *objectFieldOffset* to compute the offset of the last instance field. In both cases, a VM-dependent fudge factor is added to account for the object header and for object alignment and padding.

Issues. Object size is very implementation dependent. Accounting for the object header and alignment requires adding VM-dependent constants for these parameters.

3.4.12 Large Arrays and Off-Heap Data Structures

Description. This pattern uses off-heap memory to create large arrays or data structures with manual memory management.

Rationale. Java's arrays are indexed by `int` and are thus limited to 2^{31} elements. Using *Unsafe*, larger buffers can be allocated outside the heap.

Implementation. A block of memory is allocated with *allocateMemory* and then accessed using *Off-Heap Get* and *Off-Heap Put* methods. The block is freed with *freeMemory*.

Issues. This pattern has all the issues of manual memory management: memory leaks, dangling pointers, double free, etc. One issue, mentioned on Stack Overflow, is that the memory returned by *allocateMemory* is uninitialized and may contain garbage.[10] Therefore, care must be taken to initialize allocated memory before use. The *Unsafe* method *setMemory* can be used for this purpose.

[10] `http://stackoverflow.com/questions/16723244`

3.4.13 Get Memory Page Size

Description. `sun.misc.Unsafe` is used to determine the size of a page in memory.
Rationale. The page size is needed to allocate buffers or access memory by page. A common use case is to round up a buffer size, typically a *java.nio.ByteBuffer*, to the nearest page size. Hadoop uses the page size to track memory usage of cache files mapped directly into memory using *java.nio.MappedByteBuffer*. Another use is to process a buffer page-by-page. Some native libraries require or recommend allocating buffers on page-size boundaries.[11]
Implementation. Call *pageSize*.
Issues. Some platforms on which the JVM runs do not have virtual memory, so requesting the page size is non-portable.

3.4.14 Load Class without Security Checks

Description. `sun.misc.Unsafe` is used to load a class from an array containing its bytecode. Unlike with the *ClassLoader* API, security checks are not performed.
Rationale. This pattern is useful for implementing lambdas, dynamic class generation, and dynamic class rewriting. It is also useful in application frameworks that do not interact well with user-defined class loaders.
Implementation. The pattern is implemented using the *defineClass* method, which takes a byte array containing the bytecode of the class to load.
Issues. The pattern violates the Java security model. Untrusted code could be introduced into the same protection domain as trusted code.

3.5 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] per-

[11] `http://stackoverflow.com/questions/19047584`

Table 3.2. 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	✓		✓	

formed a survey on the OpenJDK mailing list to study how Unsafe is used[12] and describes several of these proposals.

A summary of the patterns with existing and proposed alternatives to *Unsafe* is shown in Table 3.2. 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 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 syn-

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

chronizer classes. These classes 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[13] [14] 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 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.[15]

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]

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

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

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

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[16] 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].

3.6 Conclusions

`sun.misc.Unsafe` is an API that was designed for limited use in system-level runtime library code. The *Unsafe* API is powerful, but dangerous. The improper

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

use of *Unsafe* undermines JAVA's safety guarantees. We studied to what degree *Unsafe* usage has spread into third-party libraries, to what degree such third-party usage of *Unsafe* can impact existing Java code, and which *Unsafe* API features such third-party libraries actually use. We studied the questions and discussions developers have about *Unsafe*, and we identified common usage patterns. We thereby provided a basis for evolving the *Unsafe* API, the JAVA language, and the JVM by eliminating unused or abused unsafe features, and by providing safer alternatives for features that are used in meaningful ways. We hope this will help to make *Unsafe* safer.

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 and issues including the term `ClassCastException` on *GitHub* within projects marked as using the JAVA language. The search returns about 171K[1] and 73K[2] results respectively at the time of writing. At first glance, these results indicate that indeed `ClassCastException` represents a source for problems to developers. We have included here a few commit results as an example.[3] This footnote (3) should be integrated into the text.

Forgotten Guard. The following listing[4] shows a cast applied to the variable `job` (in line 6) 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`.

```
1 @@ -41,6 +41,8 @@ public SCMTypeColumn() {
2     }
3         public String getScmType(@SuppressWarnings("rawtypes") Job job) {
4 +         if(!(job instanceof AbstractProject<?, ?>))
5 +             return "";
6         AbstractProject<?, ?> project = (AbstractProject<?, ?>) job;
7         return project.getScm().getDescriptor().getDisplayName();
8     }
```

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

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

[3] 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*.

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

Wrong Cast Target. In the next example[5] the `CustomFileFilter` is an inner static class inside `JCustomFileFilter`. Notice that 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;
  if (!Objects.equals(this.extensions, other.extensions)) {
      return false;
  }
```

Generic Type Inference Mismatch. In the following listing,[6] 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`.

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

Looking at the definition of the `getProperty` method below,[7] 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`.

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

Compiler Bug. One issue[8] shows bad things happen when abusing the type system. A bug in the javac compiler[9] was causing JVM's `checkcast` instructions to be skipped. This bug was fixed in JDK 9, breaking Mockito answer strategies.

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

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

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

[8] https://github.com/mockito/mockito/issues/357

[9] https://bugs.openjdk.java.net/browse/JDK-8058199

This indicates that type casts represent a source of errors for developers. Therefore we want to understand why developers need to use type casts. To this end, 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. Nevertheless, as we shall see later, in some situations a developer is forced to insert upcasts. 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:

*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. In §4.1 we first give an estimation of how often the cast operator is used in real-world applications to answer *CRQ*1. In §4.2 we introduce the methodology we have used to devise cast usage patterns. Then, §4.3 presents the cast usage patterns we have devised to answer both *CRQ*2 and *CRQ*3.

4.1 Is the Cast Operator used?

To answer *CRQ*1 (*How frequently is casting used in common application code?*), several elements are needed.

Source Code Analysis. We have gathered cast usage 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.[10]

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. To answer *CRQ*1, we have analyzed 195 JAVA projects in *lgtm*.

Ultimately, 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. The query to gather this statistics is available online.[11]

Number of Projects	195
Number of LOC	19,264,590
Number of Methods	1,492,490
Number of Methods *w/*Cast	99,018
Number of Expressions	57,292,018
Number of Cast Expressions	21,491

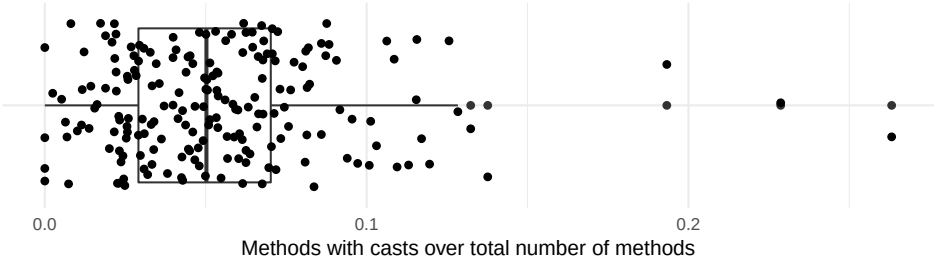
The *Number of Methods* and *Number of Methods w/Cast* values includes only methods with a body, *i.e.*, not abstract, nor native. The *Number of Expressions* value show how many expressions there are in the ASTs of all source code analyzed. Finally, the *Number of Cast Expressions* value indicates how 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[12]). 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

[10] `https://lgtm.com/`
[11] `https://gitlab.com/acuarica/java-cast-queries/blob/master/ql/stats.ql`
[12] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html`

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 5.47% 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.

4.2 Finding Casts Usage Patterns

Similarly to §4.1, 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?*) several elements are needed.

Source Code Analysis. Given the complexity of finding cast usage patterns, we have opted to manually analyze code snippets to devise cast usage patterns. We used the QL query language within the *lgtm* service to fetch all cast operator instances.

Projects. As for the project selection, we have used the Semmle project database. We argue this database provides a close approximation to "real world" applications, since teams that want their code to be analyzed push their projects onto lgtm.com to run queries against them. This filters out uninteresting projects, *e.g.*, student projects. There are also popular projects, *e.g.*, Apache Maven,[13] Neo4j,[14] Hibernate,[15] that were pulled in by Semmle itself as demo projects.

At the time of writing, there is a total of 7.559 projects in the Semmle project database, with a total 10,193,435 casts. There are 215 projects for which we could not get the source code to be analyzed. In total, these 215 projects repre-

[13] `https://lgtm.com/projects/g/apache/maven`
[14] `https://lgtm.com/projects/g/neo4j/neo4j/`
[15] `https://lgtm.com/projects/g/hibernate/hibernate-orm/`

sents 1,162,583 casts. Moreover, there are also 516 projects that does not contain any cast. Therefore the total cast population to be analyzed consists of 9,030,852 casts spread in 6,840 projects.

We assume that all cast instances can be categorized in several patterns. Since the amount of cast instances is quite large, it is not feasible to *manually* analyze all of them. Therefore we have opted to perform random sampling to get a subset of cast instances to manually analyze. Now comes the question: *What is an appropriate sample size?* We want to pick a sample size such that the probability of missing the least frequent pattern is extremely low. This kind of question is exactly answered by the **Hypergeometric Distribution**.

The Hypergeometric Distribution is a discrete probability distribution used with a finite population of ~~size~~ N ~~of~~ subjects with two distinct features. It is used to calculate the probability of obtaining k successes of getting one feature — provided that there are K subjects with that feature in the population — in n draws.

Returning to our problem of finding an appropriate sample size, we can model our question as follows: The two distinct features are either finding a cast of the least frequent pattern or not. Therefore we want to know what is the probability of not finding this pattern, *i.e.*, $k = 0$. Our finite population consists of $N = 9,030,852$ cast instances. Then we assume that a pattern is irrelevant if it represents less than 0.1% of the population, thus $K = 9,000$ cast instances. Plugging-in these parameters using the Hypergeometric Distribution formula,[16] with different sample sizes, we found that with a sample size of $n = 5000$ the probability is 0.0068, *i.e.*, less than 0.1%, of not finding the least frequent pattern.

Put your raw results (the file with the 5000 cast instances and your tags) into some repo and say here that it's available there.

4.3 Cast Usage Patterns

Using the methodology described in the above section, we have devised 32 casts usage patterns.[17] In this section we present the cast usage patterns we found. The manual categorization can be found online.[18] To ease the patterns presentation, we have clustered them in 10 groups according to their purpose. Table 4.1

[16] The reader can use any available online Hypergeometric Distribution calculator to plug-in these parameters, *e.g.*, `https://keisan.casio.com/exec/system/1180573201`

[17] We have excluded casts that represents primitive numeric type conversions, as they do not represent any pattern. However, during our manual analysis we found 238 primitive conversions. Moreover, we found 206 links that were not accessible during our analysis.

[18] `https://gitlab.com/acuarica/phd-thesis/blob/master/analysis/casts-5000.csv`

Table 4.1. Cast Usage Patterns and their Groups.

#	Cast Pattern	Cast Pattern Description	Group
1	PatternMatching	Cast guarded with an `instanceof` operator.	Guarded
2	TypeTag	Cast guarded by an application-specific condition.	
3	Equals	A cast used in the implementation of the well-known `equals` method.	
4	GetByClassLiteral	Similar to TypeTag, but guarded by a `Class` literal.	
5	Family	A cast applied in a family of classes.	Creational
6	Factory	A cast used to convert a newly created objects.	
7	KnownLibraryMethod	When the client of an API knows the exact return type of a method invocation.	
8	Tag	A library provides a field to stash user-specific values. Cast to it.	
9	Deserialization	A cast used to convert newly created objects in deserialization.	
10	StackSymbol	A cast to an heterogenous stack.	
11	CreateByClassLiteral	Cast an object created depending on a class literal.	
12	LookupById	A cast to an heterogenous collection element.	Tuples
13	StaticResource	A cast to a value known at compile-time.	
14	ObjectAsArray	A cast to a constant array slot used as a field of an object.	
15	SelectOverload	A cast to disambiguate between overloaded methods.	Member Resolution
16	AccessPrivateField	A cast to access a private field in a superclass.	
17	Clone	A cast to the well-known method `clone`.	Variance
18	CovariantReturn	A generalization of Clone, A cast when the return type of a method is covariant.	
19	RemoveTypeParameter	Remove type parameter in a generic type to permit covariant generics.	
20	ImplicitIntersectionType	A cast to implicitly use an intersection type.	Implicit Types
21	ImplicitUnionType	A cast to implicitly use a union type.	
22	SoleSubclassImplementation	A cast to the only subclass implementation.	Structural
23	RecursiveGeneric	A generic cast to `this`.	
24	NewDynamicInstance	Cast the result of the `newInstance` method in the `Class`, `Constructor`, or `Array` classes.	Reflection
25	ReflectiveAccessibility	Cast the result of the `Method::invoke`, or `Field::get`.	
26	UncheckedCast	An unchecked cast.	Unchecked
27	GenericArray	A cast to create a generic array.	
28	RemoveWildcard	A cast used to remove the wildcard in a generic type.	
29	Literal	A conversion between numeric types at compile-time.	Code Smell
30	RawTypes	A cast used instead of the declared generic type.	
31	Redundant	A cast that is not necessary for compilation.	
32	VariableLessSpecificType	A cast to a variable that could be declared to be more specific.	

presents each pattern and the groups they belong to. Moreover, we are interested in the scope of the cast instance, *i.e.*, *does it appear in application source code, test code, or generated code?* Figure 4.1 shows our patterns and their occurrences sort by frequency. The column on the right correspond to the group the pattern belongs to.

Each pattern is described using the following template:

- **Description.** Tells what is this pattern about. It gives a general overview of the structure of the pattern.

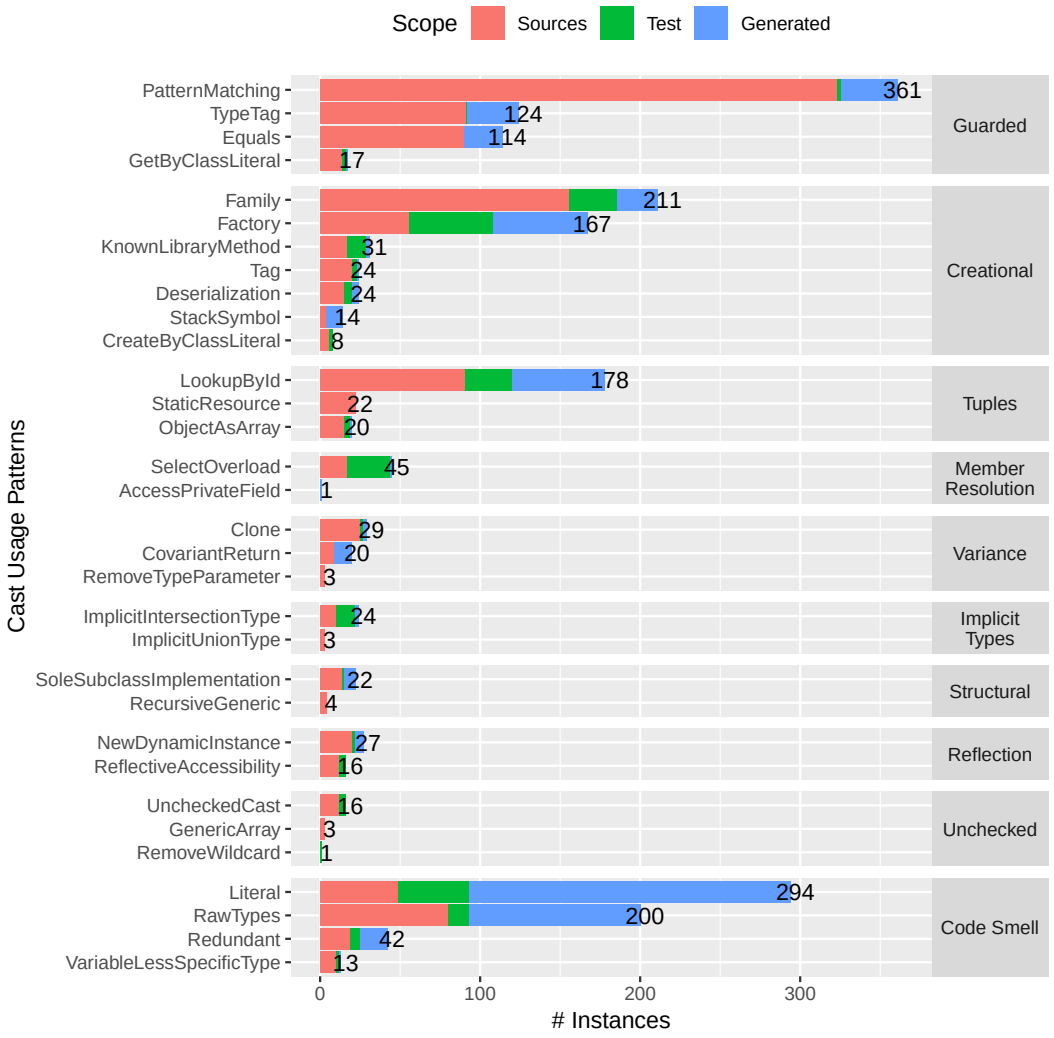


Figure 4.1. Cast Patterns Occurrences

- **Instances.** Gives one or more concrete examples found in real code.[19] For each instance presented here, we provide the link to the source code repository in *lgtm*. We provide the link in case the reader wants to do further inspection of the snippet presented.[20]

[19]Please notice that the snippets presented here were slightly modified for formatting purposes. Moreover, to facilitate some snippet presentations, we remove irrelevant code and replace it with the comment `// [...]`.

[20]Instead of presenting *lgtm* long URLs, we have used the URL shortening service `https://bitly.com/` for an easier reading.

- **Detection.** Describes briefly how this pattern was detected in terms of the tags introduced in the previous section.
- **Discussion.** Presents suggestions, flaws, or comments about the pattern.
- **Related Patterns.** How the pattern being described relates to other patterns?

GUARDED Patterns

The patterns in this group are guarded casts.

4.3.1 PatternMatching

Description. This pattern is composed of a guard (`instanceof`) followed by a cast on known subtypes of the static type. Often there is just one case and the default case, *i.e.*, `instanceof` fails, does a no-op or reports an error. Another common approach is to have several cases, usually one *per* subtype.
Instances. The following listing shows an example of the PATTERNMATCHING pattern.[21] In this example, there is only a single case. After the cast is done, a specific operation is performed on the casted type, *i.e.*, invoking the `checkBlock` method.

```
1 Item item = helmet.getItem();
2 if (item instanceof ItemImaginationGlasses)
3     return ((ItemImaginationGlasses)item).checkBlock(what, helmet, this);
```

In the next case,[22] more than one type test are performed. Each cast corresponds to a type test.

```
1 public static String getStringFromObject(Object object) {
2     if (object instanceof Item) {
3         return getStringFromStack(new ItemStack((Item) object));
4     } else if (object instanceof Block) {
5         return getStringFromStack(new ItemStack((Block) object));
```

[21] http://bit.ly/2FzYYHq
[22] http://bit.ly/2HnNwB7

```
6       } else if (object instanceof ItemStack) {
7           return getStringFromStack((ItemStack) object);
8       } else if (object instanceof String) {
9           return (String) object;
10      } else if (object instanceof List) {
11          return getStringFromStack((ItemStack) ((List) object).get(0));
12      } else return "";
13  }
```

There are situations when the type test is applied to more than one variable. In the following example[23] a double type test is performed on the parameters o1 and o2.

```
1   public static boolean nullSafeEquals(Object o1, Object o2) {
2       if (o1 == o2) {
3           return true;
4       }
5       if (o1 == null || o2 == null) {
6           return false;
7       }
8       if (o1.equals(o2)) {
9           return true;
10      }
11      if (o1.getClass().isArray() && o2.getClass().isArray()) {
12          if (o1 instanceof Object[] && o2 instanceof Object[]) {
13              return Arrays.equals((Object[]) o1, (Object[]) o2);
14          }
15          if (o1 instanceof boolean[] && o2 instanceof boolean[]) {
16              return Arrays.equals((boolean[]) o1, (boolean[]) o2);
17          }
18          if (o1 instanceof byte[] && o2 instanceof byte[]) {
19              return Arrays.equals((byte[]) o1, (byte[]) o2);
20          }
21          if (o1 instanceof char[] && o2 instanceof char[]) {
22              return Arrays.equals((char[]) o1, (char[]) o2);
23          }
24          if (o1 instanceof double[] && o2 instanceof double[]) {
25              return Arrays.equals((double[]) o1, (double[]) o2);
```

[23] http://bit.ly/2FDN9Rd

```
26          }
27          if (o1 instanceof float[] && o2 instanceof float[]) {
28              return Arrays.equals((float[]) o1, (float[]) o2);
29          }
30          if (o1 instanceof int[] && o2 instanceof int[]) {
31              return Arrays.equals((int[]) o1, (int[]) o2);
32          }
33          if (o1 instanceof long[] && o2 instanceof long[]) {
34              return Arrays.equals((long[]) o1, (long[]) o2);
35          }
36          if (o1 instanceof short[] && o2 instanceof short[]) {
37              return Arrays.equals((short[]) o1, (short[]) o2);
38          }
39      }
40      return false;
41  }
```

Detection. To detect this pattern, we look for a cast guarded by an `instanceof` expression. The operand expression to both the cast and the `instanceof` needs to be the same. In the case the operand expression is a variable, we also check that there is no assignment between the `instanceof` guard and the cast to that variable.

In some situations the operand expression is a method invocation. We make sure that the value returned is the same for both the `instanceof` and the cast, *i.e.*, it is a pure method.

Discussion. The PATTERNMATCHING pattern consists of testing the runtime type of a variable against several related types. It is a technique that allows a developer to take different actions according to the runtime type of an object. Depending on the — runtime — type of an object, different cases, usually one for each type will follow.

The PATTERNMATCHING pattern can be seen as an *ad-hoc* alternative to pattern matching. This construct can be seen in several other languages, *e.g.*, SCALA, C#, and HASKELL. For instance, in SCALA the pattern matching construct is achieved using the `match` keyword. In this example,[24] a different action is taken according to the runtime type of the parameter `notification` (line 9).

[24] Adapted from `https://docs.scala-lang.org/tour/pattern-matching.html`

```scala
abstract class Notification
case class Email(sender: String, title: String, body: String)
    extends Notification
case class SMS(caller: String, message: String)
    extends Notification
case class VoiceRecording(contactName: String, link: String)
    extends Notification

def showNotification(notification: Notification): String = {
    notification match {
        case Email(email, title, _) =>
        s"You got an email from $email with title: $title"
        case SMS(number, message) =>
        s"You got an SMS from $number! Message: $message"
        case VoiceRecording(name, link) =>
        s"Voice Recording from $name! Click the link: $link"
    }
    }
    val someSms = SMS("12345", "Are you there?")
    val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

    // prints You got an SMS from 12345! Message: Are you there?
    println(showNotification(someSms))

    // Voice Recording from Tom! Click the link: voicerecording.org/id/123
    println(showNotification(someVoiceRecording))
```

As a workaround, alternatives to the PATTERNMATCHING pattern can be the visitor pattern or polymorphism. But in some cases, the chain of `instanceofs` is of boxed types. Thus no polymorphism can be used. There is an ongoing proposal[25] to add pattern matching to the JAVA language.

Related Patterns. The EQUALS pattern can be seen as a special case for the PATTERNMATCHING pattern, because often the parameter to the `equals` method is guarded by an `instanceof` expression.

[25] `http://openjdk.java.net/jeps/305`

4.3.2 TypeTag

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

A cast guarded by a test on a field from the same object instead of using `instanceof`.

Instances. The following example[26] shows an instance of the TypeTag pattern. The cast type of the parameter `reference` is determined by the value of the parameter `referenceType`.

```
1  private int getReferenceIndex(int referenceType, Reference reference) {
2      switch (referenceType) {
3          case ReferenceType.FIELD:
4              return fieldSection.getItemIndex((FieldRefKey) reference);
5          case ReferenceType.METHOD:
6              return methodSection.getItemIndex((MethodRefKey) reference);
7          case ReferenceType.STRING:
8              return stringSection.getItemIndex((StringRef) reference);
9          case ReferenceType.TYPE:
10             return typeSection.getItemIndex((TypeRef) reference);
11         case ReferenceType.METHOD_PROTO:
12             return protoSection.getItemIndex((ProtoRefKey) reference);
13         default:
14             throw new ExceptionWithContext(
15                 "Unknown reference type: %d",  referenceType);
16     }
17 }
```

In the next case[27] a type test is performed — through a method call — before actually applying the cast to the variable `props`. Note that the type test is using the `instanceof` operator (line 8).

```
1  @Override
2  public CTSolidColorFillProperties getSolidFill() {
3      return isSetSolidFill() ? (CTSolidColorFillProperties)props : null;
4  }
5
```

[26] http://bit.ly/2Ho8bVL
[27] http://bit.ly/2FW5SXU

```
@Override
public boolean isSetSolidFill() {
    return (props instanceof CTSolidColorFillProperties);
}
```

Detection. The detection of this pattern is similar to the PATTERNMATCHING detection, but instead of looking for an `instanceof` guarded cast, we look for an application-specific guard.

Discussion.

Related Patterns.

4.3.3 Equals

Description. This pattern is a common pattern to implement the `equals` method (declared in `java.lang.Object`). A cast expression is guarded by either an `instanceof` test or a `getClass` comparison (usually to the same target type as the cast); in an `equals`[28] method implementation. This is done to check if the argument has same type as the receiver (`this` argument).

Notice that a cast in an `equals` method is needed because it receives an `Object` as a parameter.

Instances. The following listing[29] shows an example of the pattern. In this case, `instanceof` is used to guard for the same type as the receiver.

```
@Override
public boolean equals(Object obj) {
    if ( this == obj ) {
        return true;
    }
    if ( (obj instanceof Difference) ) {
        Difference that = (Difference) obj;
        return actualFirst == that.actualFirst
          && expectedFirst == that.expectedFirst
          && actualSecond == that.actualSecond
          && expectedSecond == that.expectedSecond
          && key.equals( that.key );
    }
```

[28] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

[29] http://bit.ly/2vJw94J

```
14      return false;
15  }
```

Alternatively, the following listing[30] shows another example of the EQUALS pattern. But in this case, a `getClass` comparison is used to guard for the same type as the receiver.

```
1   @Override
2   public boolean equals( Object o ) {
3       if ( this == o ) return true;
4       if ( o == null || getClass() != o.getClass() )
5           return false;
6
7       ValuePath that = (ValuePath) o;
8       return nodes.equals(that.nodes) &&
9           relationships.equals(that.relationships);
10  }
```

In some situations, the type casted to is not same as the enclosing class. Instead, the type casted to is the super class of the enclosing class. The following example[31] shows this scenario. This usually happens when the Google AutoValue library[32] is used. ~~The~~ AutoValue is a code generator for value classes.

```
1   @AutoValue
2   abstract class ListsItem implements Parcelable {
3       // [...]
4   }
5
6   abstract class $AutoValue_ListsItem extends ListsItem {
7       @Override
8       public boolean equals(Object o) {
9         if (o == this) {
10          return true;
11        }
12        if (o instanceof ListsItem) {
13          ListsItem that = (ListsItem) o;
```

[30] http://bit.ly/2vKP0MW
[31] http://bit.ly/2HmHMYE
[32] https://github.com/google/auto/tree/master/value

```
14          return (this.id == that.id())
15                  && (this.name.equals(that.name()))
16                  && (this.itemCount == that.itemCount());
17        }
18        return false;
19      }
20  }
```

Detection. The detection query looks for a cast expression inside an `equals` method implementation. Moreover, the cast needs to be guarded by either an `instanceof` test or a `getClass` comparison. And the type being casted to needs to be either the same as the enclosing class or a superclass of it.
Discussion. The pattern for an `equals` method implementation is well-known.

We found out that, with respect to cast, most `equals` methods are implemented with the same structure. Maybe avoid boilerplate code by providing code generation, like in HASKELL (with `deriving`).

Vaziri et al. [2007] propose a declarative approach to avoid boilerplate code when implementing both the `equals` and `hashCode` methods. They manually analyzed several applications, and found there are many issues while implementing `equals()` and `hashCode()` methods. It would be interesting to check whether these issues happen in real application code.

There is an exploratory document[33] by Brian Goetz — JAVA Language Architect — addressing these issues from a more general perspective. It is definitely a starting point towards improving the JAVA language.
Related Patterns. This pattern can be seen as a special instance of the PATTERN-MATCHING pattern.

4.3.4 GetByClassLiteral

Description.
Instances.
Detection.
Discussion.
Related Patterns.

CREATIONAL Patterns

[33] `http://cr.openjdk.java.net/~briangoetz/amber/datum.html`

These patterns creates to how they are created.

In these intro paragraphs, can you also enumerate the patterns by name, so readers see what's coming?

4.3.5 Family

Description. Family polymorphism.
Instances. The following example[34] shows an instance of the FAMILY pattern.

```
1  public interface StepInterface extends VariableSpace, HasLogChannelInterface {
2    // [...]
3    public void stopRunning( StepMetaInterface stepMetaInterface,
4            StepDataInterface stepDataInterface ) throws KettleException;
5  }
6
7  public class DynamicSQLRow extends BaseStep implements StepInterface {
8    private DynamicSQLRowMeta meta;
9    private DynamicSQLRowData data;
10   // [...]
11   public void stopRunning( StepMetaInterface smi,
12         StepDataInterface sdi ) throws KettleException {
13     meta = (DynamicSQLRowMeta) smi;
14     data = (DynamicSQLRowData) sdi;
15     // [...]
16   }
17 }
```

Detection.
Discussion. Ernst [2001]
Related Patterns.

4.3.6 Factory

Description. Creates an object based on some arguments either to the method call or constructor. Since the arguments are known at compile-time, cast to the specific type.

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

[34] http://bit.ly/2FN59J8

The method is declared to return URLConnection but can return a more specific type based on the URL string. Cast to that. We should generalize this pattern.
Instances.
35
36

```
KeyPairGenerator pairGen = KeyPairGenerator.getInstance("RSA");
pairGen.initialize(1024);
KeyPair keyPair = pairGen.generateKeyPair();
RSAKey rsaJWK1 = new RSAKey.Builder((RSAPublicKey) keyPair.getPublic())
        .privateKey((RSAPrivateKey) keyPair.getPrivate())
        .keyID("1")
        .build();
keyPair = pairGen.generateKeyPair();
RSAKey rsaJWK2 = new RSAKey.Builder((RSAPublicKey) keyPair.getPublic())
        .privateKey((RSAPrivateKey) keyPair.getPrivate())
        .keyID("2")
        .build();
```

Detection.
Discussion.
Related Patterns.

4.3.7 KnownLibraryMethod

Description. There are cases when a method's return type is less specific than the actual return type value. This is usually to hide implementation details. Nevertheless, sometimes it is convenient for the developer to work directly on the actual return type. This pattern is used to cast from the method's return type to the *known* actual return type.
Instances.

4.3.8 Tag

Description. Used
Instances.

[35] http://bit.ly/2HvRlUX
[36] https://docs.oracle.com/javase/8/docs/api/java/security/KeyPair.html#getPrivate()

Detection.
Discussion.
Related Patterns. Related to VARIABLELESSSPECIFICTYPE.

4.3.9 Deserialization

Description. Used to deserialize an object.
Instances.

```
1 Object readObject = new ObjectInputStream(unserialize).readObject();
2 assertNotNull(readObject);
3 return type.cast(readObject);
```

Listing 4.1. Instance of the pattern (`http://bit.ly/2KOpj3A`)

Detection.

```
1 DeserializationCastPattern() {
2     this instanceof ReadObjectCastTag
3 }
```

4.3.10 StackSymbol

Description.
Instances.
Detection.
Discussion.
Related Patterns.

4.3.11 CreateByClassLiteral

Description.
Instances.
Detection.
Discussion.
Related Patterns.

TUPLES Patterns

Tuples patterns.

4.3.12 LookupById

Description. This pattern is used to extract stashed values from a generic container.

Lookup an object by ID, tag or name and cast the result (it is used often in Android code). It accesses a collection that holds values of different types (usually implemented as `Collection<Object>` or as `Map<K, Object>`).

Instances.

In the example shown in listing, the `getAttribute` method returns `Object`. The variable `context` is of type `BasicHttpContext`, which is implemented with `HashMap`.

```
1 AuthState authState =
2         (AuthState) context.getAttribute(ClientContext.TARGET_AUTH_STATE);
```

Listing 4.2. Example of the pattern.

Discussion.

This pattern suggests an heterogeneous dictionary. Given our manual inspection, we believe that all dictionary keys and resulting types are known at compile-time, *i.e.*, by the programmer. But in any case a cast is needed given the restriction of the type system. As a complementary analysis, it would be interesting to check whether all call sites to `getAttribute` receives a constant (`final static` field).

Notice that this pattern is not guarded by an `instanceof`. However, the cast involved does not fail at runtime. This means that the source of the cast is known to the programmer. This raises the following questions:

- *What kind of analysis is needed to detect the source of the cast?*
- *Is worth to have it?*
- *Is better to change API?*
- *How other — statically typed — languages support this kind of idiom?*
- *Could generative programming a.k.a. templates solve this problem?*

4.3.13 StaticResource

Description.

A cast to a method access to `findViewById` or a method that reads a static resource.

This is a pattern seen when using the Android platform.

Discussion.

These casts could be solved by using code generation, or partial classes as in C#.

4.3.14 ObjectAsArray

Description. In this pattern an array is used as an untyped object. A cast is applied to a constant array slot, *e.g.*, `(String) array[1]`.

Instances. The following example[37] shows an instance of the ObjectAsArray pattern. A cast is performed to a constant array slot: `(BitSet)currentState[3]` in line 11. Nevertheless, the `currentState` parameter is accessed in lines 7 and 15 with constant indices, denoting an untyped object. Looking further, in the `else` branch (line 17) is where the array is being created matching the usage mentioned above.

```
1  public Object[] replacingDetachedState(Detachable pc, Object[] currentState) {
2      if ((flags&FLAG_RESETTING_DETACHED_STATE) != 0) {
3          return null;
4      } else if ((flags&FLAG_RETRIEVING_DETACHED_STATE) != 0) {
5          // Retrieving the detached state from the detached object
6          // Don't need the id or version since they can't change
7          BitSet theLoadedFields = (BitSet)currentState[2];
8          for (int i = 0; i < this.loadedFields.length; i++) {
9              this.loadedFields[i] = theLoadedFields.get(i);
10         }
11         BitSet theModifiedFields = (BitSet)currentState[3];
12         for (int i = 0; i < dirtyFields.length; i++) {
13             dirtyFields[i] = theModifiedFields.get(i);
14         }
15         setVersion(currentState[1]);
16         return currentState;
17     } else {
18         // Updating the detached state in the detached object with our state
19         Object[] state = new Object[4];
20         state[0] = myID;
21         state[1] = getVersion(myPC);
22         // Loaded fields
23         BitSet loadedState = new BitSet();
```

[37] http://bit.ly/2S1L5Zf

```
24          for (int i = 0; i < loadedFields.length; i++) {
25              if (loadedFields[i]) {
26                  loadedState.set(i);
27              } else {
28                  loadedState.clear(i);
29              }
30          }
31          state[2] = loadedState;
32          // Modified fields
33          BitSet modifiedState = new BitSet();
34          for (int i = 0; i < dirtyFields.length; i++) {
35              if (dirtyFields[i]) {
36                  modifiedState.set(i);
37              } else {
38                  modifiedState.clear(i);
39              }
40          }
41          state[3] = modifiedState;
42          return state;
43      }
44  }
```

Detection.
Discussion. This pattern usually suggests an abuse of the type system.
Related Patterns.

Member Resolution Patterns

4.3.15 SelectOverload

Description. This pattern is used to select the appropriate version of an overloaded method[38] where two or more of its implementations differ *only* in some argument type.

A cast to `null` is often used to select against different versions of a method, *i.e.*, to resolve method overloading ambiguity. Whenever a `null` value needs to be an argument of an a cast is needed to select the appropriate implementation.

[38] Using ad-hoc polymorphism Strachey [2000]

This is because the type of `null` has the special type *null*[39] which can be treated as any reference type. In this case, the compiler cannot determine which method implementation to select.

Instances. The following listingshows an example of pattern. In this example, there are three versions of the `onSuccess` method, The cast `(String) null` is used to select the appropriate version (line 7), based on the third parameter. Overloaded methods that differ only in their argument type (the third one).

Another use case is to select the appropriate the right argument when calling a method with variable arguments.

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

```
1 public void onSuccess(
2       int statusCode, Header[] headers, JSONObject response) {...}
3
4 public void onSuccess(
5       int statusCode, Header[] headers, JSONArray response) {...}
6
7 public void onSuccess(
8       int statusCode, Header[] headers, String responseString) {...}
```

In the following example[40] `actual.data()` returns `Long`.

```
1 private void eq(long expected, IntegerReply actual) {
2       assertEquals(expected, (long) actual.data());
3 }
4
5 public static void assertEquals(Object expected, Object actual) { }
6 public static void assertEquals(long expected, long actual) { }
```

Detection. Listing 4.3 shows how to detect this pattern. This pattern shows up when a cast is directly applied to the `null` constant.

```
1 import java
2
3 from CastExpr ce, NullLiteral nl
4 where ce.getExpr() = nl
```

[39] https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.1
[40] http://bit.ly/2FC9Llb

```
5 select ce
```

Listing 4.3. Detection of the pattern.

Discussion. Casting the `null` constant seems rather artificial. This pattern shows either a lack of expressiveness in JAVA or a bad API design. Several other languages support default parameters, *e.g.*, SCALA, C# and C++. Adding default parameters might be a partial solution.

4.3.16 AccessPrivateField

Description.
Instances.
Detection.
Discussion.
Related Patterns.

VARIANCE Patterns

4.3.17 Clone

Description. A cast to a `clone` method.
Instances.

4.3.18 CovariantReturn

Description.
Instances.
- CovariantReturn

```
1 EditorView editorView = ((CayenneModelerFrame) Application.
2         getInstance().getFrameController().getView())
3         .getView();
```

Detection.
Discussion.
Related Patterns.
CovariantReturn

4.3.19 RemoveTypeParameter

Description.
Instances.
Detection.
Discussion.
Related Patterns.

Implicit Types Patterns

4.3.20 ImplicitIntersectionType

Description. Cast a reference v of type — class or interface — T to an interface type I whether T does not implement I. The cast succeeds at runtime because all possible runtime types of v actually implement the interface I. For instance, in `(Comparable)(Number)4`, `Number` does not implement the `Comparable` interface, but class `Integer` does.
Instances.

```
1 final Comparable max = (Comparable) properties.getMaxValue();
```

Listing 4.4. From `http://bit.ly/2FQOt4v`

4.3.21 ImplicitUnionType

Description.
Instances.
Detection.
Discussion.
Related Patterns.

Structural Patterns

4.3.22 SoleSubclassImplementation

Description.

Instances. The following example[41]

```
1 public final EncodedElement.Builder addAllThrown( Iterable<? extends Type> elements) {
2     this.thrown.addAll(elements);
3     return (EncodedElement.Builder) this;
4 }
```

Detection.
Discussion.
Related Patterns.

4.3.23 RecursiveGeneric

Description.
Instances.
Detection.
Discussion.
Related Patterns.

Reflection Patterns

4.3.24 NewDynamicInstance

Description. Dynamically creates an object or array by means of reflection. The `newInstance` method family declared in the `Class`,[42] `Array`[43, 44] and `Constructor`[45] classes creates an object or array dynamically by means of reflection, *i.e.*, the type of object being created is not known at compile-time. This pattern consists of casting the result of these methods to the appropriate target type.

[41] http://bit.ly/2S4BoJs
[42] https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#newInstance--
[43] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html#newInstance-java.lang.Class-int-
[44] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html#newInstance-java.lang.Class-int...-
[45] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Constructor.html#newInstance-java.lang.Object...-

Instances. The following example[46] shows a cast to the `Class.newInstance()` method.

```
1 logger = (AuditLogger) Class.forName(className).newInstance();
```

The following example[47] shows how to dynamically create an array, using the `Array` class.

```
1 return list.toArray( (T[]) Array.newInstance( componentType, list.size()));
```

Whenever a constructor other than the default constructor is needed, the `newInstance` method declared in the `Constructor` class should be used to select the appropriate constructor, as shown in the following example. [48]

```
1 return (Exception) Class
2                         .forName(className)
3                         .getConstructor(String.class)
4                         .newInstance(message);
```

The following example[49] shows a guarded instance of the NEWDYNAMICINSTANCE pattern. This seems rather unusual, as this pattern is not guarded.

```
1  private static List<String> getMapperMethodNames(final Class clazz) {
2      try {
3          if (clazz != null) {
4              Object obj = clazz.newInstance();
5              if (obj instanceof BaseMethodMapper) {
6                  return ((BaseMethodMapper) obj).getAllFunctionNames();
7              }
8          }
9      } catch (Exception e) {
10         e.printStackTrace();
11     }
12     return null;
13 }
```

[46] http://bit.ly/2HC3IPg
[47] http://bit.ly/2Hp5Hqc
[48] http://bit.ly/2HsUgOo
[49] http://bit.ly/2HC33xg

There are cases when the cast is not directly applied to the result of the `newInstance` method. The following snippet shows such a case.[50] The cast is used to convert from `Class<?>` to `Class<ConfigFactory>` (line 4). The `newInstance` invocation then does not need a direct cast (line 8) given the definition of the `clazz` variable (line 2). Nevertheless, the cast is unchecked, and a `checkcast` instruction is going to be emitted anyway for the result of the `newInstance` invocation.

```
1 ClassLoader tccl = Thread.currentThread().getContextClassLoader();
2 final Class<ConfigFactory> clazz;
3 if (tccl == null) {
4     clazz = (Class<ConfigFactory>) Class.forName(factoryName);
5 } else {
6     clazz = (Class<ConfigFactory>) Class.forName(factoryName, true, tccl);
7 }
8 final ConfigFactory factory = clazz.newInstance();
```

Detection. This detection query looks for casts, where the expression being cast is a call site to methods mentioned above.
Discussion. The cast here is needed because of the dynamic essence of reflection. This pattern is mostly unguarded, that is, the application programmer knows what is the target type being created.

The following two code snippets:

```
1 Class<?> c = Class.forName("java.lang.String");
2 String pf = (String) c.newInstance();
```

```
1 Class<String> c = (Class<String>) Class.forName("java.lang.String");
2 String pf = c.newInstance();
```

compile to the same bytecode below.

```
1 ldc           #24    // String java.lang.String
2 invokestatic  #26    // Method java/lang/Class.forName
3 astore_1
4 aload_1
```

[50] http://bit.ly/2HJtXUn

```
5 invokevirtual #32    // Method java/lang/Class.newInstance
6 checkcast     #36    // class java/lang/String
```

Related Patterns. Reflection.

4.3.25 ReflectiveAccessibility

Description.

This pattern accesses a field of an object by means of reflection. It uses reflection because at compile time the field is unaccesible. Usually the method `setAccessible(true)` is invoked on the field before actually getting the value from an object.

Instances.

The following cast uses this pattern:

```
1 f.setAccessible(true);
2 HttpEntity wrapped = (HttpEntity) f.get(entity);
```

Listing 4.5. Using `Field::get` to gain access to a field.

UNCHECKED Patterns

4.3.26 UncheckedCast

Description.
Instances.
Detection.
Discussion.
Related Patterns.

4.3.27 GenericArray

Description.
Instances.
Detection.
Discussion.
Related Patterns.

4.3.28 RemoveWildcard

Description.
Instances.
Detection.
Discussion.
Related Patterns.

Code Smell Patterns

These are code smells.

4.3.29 Literal

Description. Cast a numeric or character literal or constant — defined as `static final` — to a primitive type.
Instances. Casting a literal to initialize an array is shown in the following listing.[51] Here a cast is used to initialize an array (line 5). This kind of cast is seen often in generated code. Note that the cast may be actually redundant if the value being casted is within range of the type being casted to, *e.g.*, −128..127 for `byte`.

```
1 private static byte[] piTable =
2 {
3     (byte)0xd9, (byte)0x78, (byte)0xf9, (byte)0xc4, (byte)0x19, (byte)0xdd,
4     (byte)0xb5, (byte)0xed, (byte)0x28, (byte)0xe9, (byte)0xfd, (byte)0x79,
5     (byte)0x4a, (byte)0xa0, (byte)0xd8, (byte)0x9d,
6     // [...]
7 }
```

The following listing shows an example of the Literal pattern being used in an invocation context.[52]

```
1 public static final TcpPort POWERBURST =
2     new TcpPort((short)485, "Air Soft Power Burst");
```

[51] http://bit.ly/2Fz3fuN
[52] http://bit.ly/2FuEqkC

Another example[53] using a constant instead of a literal to initialize an array (line 22).

```
1  public static enum WindowsKey {
2      // [...]
3      DELETE_KEY(83),
4      // [...]
5      ;
6
7      public final int code;
8
9      WindowsKey(final int code) {
10         this.code = code;
11     }
12 }
13
14 @Test
15 public void testNumpadDeleteOnWindowsTerminal() throws Exception {
16     // [...]
17     char[] characters = new char[]{
18         'S', 's',
19         (char) NUMPAD_KEY_INDICATOR.code,
20         (char) LEFT_ARROW_KEY.code,
21         (char) NUMPAD_KEY_INDICATOR.code,
22         (char) DELETE_KEY.code, '\r', 'n'
23     };
24     // [...]
25 }
```

Detection. This pattern looks for casting a literal or constant to a primitive type.

Discussion. JAVA already provides several types of numeric literals: `int`, `long`, `float` and `double`.

Related Patterns. Also it is related with the SELECTOVERLOAD because it does not permit for narrowing conversions.[54] Related to REDUNDANT because of cast automatically generated.

[53] `http://bit.ly/2FuBp3M`
[54] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.3`

4.3.30 RawTypes

Description. When a generic method is not used as such. The expression of this cast is a method invocation, but the declaration differs from the usage.
Instances. The following snippet[55]

TODO: Not necessarily mistakes, maybe requirement is jdk 1.2 !

```
class SecuritySupport12 extends SecuritySupport {
    ClassLoader getSystemClassLoader() {
        return (ClassLoader)
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    ClassLoader cl = null;
                    try {
                        cl = ClassLoader.getSystemClassLoader();
                    } catch (SecurityException ex) {}
                    return cl;
                }
            });
    }
}

public final class AccessController {
    // [...]
    public static <T> T doPrivileged(PrivilegedAction<T> action) {
        return action.run();
    }
    // [...]
}
```

The following example[56] uses the raw type of the `Comparable` — generic — interface.[57]

```
public class McpSettlementDetailDto implements Comparable {
    // [...]

    @Override
```

[55] http://bit.ly/2FAI5x5
[56] http://bit.ly/2FSZKzm
[57] https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
5       public int compareTo(Object o){
6           McpSettlementDetailDto mcpSettlementDetailDto=(McpSettlementDetailDto)o;
7           Integer newConsume=(int)mcpSettlementDetailDto.getConsume();
8           Integer temp=((int)this.consume);
9           return temp.compareTo(newConsume);
10      }
11  }
```

Detection.
Discussion.
Related Patterns.

4.3.31 Redundant

Description. A cast that is not necessary for compilation.
Instances. The following example[58]

```
1   transactionTemplate.execute((TransactionCallback<Void>) transactionStatus -> {
2       Post post = new Post();
3       entityManager.persist(post);
4       return null;
5   });
```

Detection.
Discussion.
Related Patterns.

4.3.32 VariableLessSpecificType

Description. This pattern occurs when a cast is applied to a variable (local variable, parameter, or field), that is usually being assigned once and is declared with a less specific type than the type of the value that is being assigned to. The type of the value being assigned to can be determined locally either within the enclosing method or class.
Instances. The following example[59] shows the VARIABLELESSSPECIFICTYPE pattern. We can see that the field `uncompressedDirectBuf` is being casted to the

[58] `http://bit.ly/2FWXw2e`
[59] `http://bit.ly/2FuDeO7`

`java.nio.ByteBuffer` class (line 13) but it is declared as `java.nio.Buffer` (line 3). Nevertheless, the field is assigned only once in the constructor (line 7) with a value of type `java.nio.ByteBuffer`. The value assigned is returned by the method `ByteBuffer.allocateDirect`.[60] Inspecting the enclosing class, there is no other assignment to the `uncompressedDirectBuf` field, thus making possible to declare it as `final`. Therefore, the cast pattern in line 13 will always succeed. Any other similar use of the `uncompressedDirectBuf` field needs to be casted to as well.

```
1  public class SnappyCompressor implements Compressor {
2      // [...]
3      private Buffer uncompressedDirectBuf = null;
4      // [...]
5      public SnappyCompressor(int directBufferSize) {
6          // [...]
7          uncompressedDirectBuf = ByteBuffer.allocateDirect(directBufferSize);
8          // [...]
9      }
10     // [...]
11     synchronized void setInputFromSavedData() {
12         // [...]
13         ((ByteBuffer) uncompressedDirectBuf).put(userBuf, userBufOff,
14             uncompressedDirectBufLen);
15         // [...]
16     }
17     // [...]
18 }
```

Detection. To detect this pattern, a cast needs to be applied to a variable whose value can be determined simply by looking at the enclosing method or class.
Discussion. In most the cases this can be considered as a bad practice or code smell. This is because by only changing the declaration of the variable to a more specific type type, the cast can be simply eliminated.
Related Patterns. This pattern is related to the REDUNDANT pattern. Although VARIABLELESSSPECIFICTYPE is not redundant, by only changing the declaration of the variable to a more specific type, the cast becomes redundant.

[60] `https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html#allocateDirect(int)`

You need at least one section to close this chapter.

Some kind of discussion/reflection, and conclusion.

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.

Expand significantly. Maybe to 5 pages or more. What insights did you gain, can you summarize to what degree you answered the RQs? What limitations? What future work?

Appendix A

JNIF: Java Native Instrumentation

This appendix presents JNIF, our library to instrument JAVA applications in native code using C/C++. Although the material presented here is not directly related to this thesis, we have used JNIF in several experiments during the development of both chapters 3 and 4. The original article have been published in Mastrangelo and Hauswirth [2014].

A.1 Introduction

Program analysis tools are important in software engineering tasks such as comprehension, verification and validation, profiling, debugging, and optimization. They can be broadly categorized either as static or dynamic, based on the input that they take. Static analysis tools carry out their task using as input only a program in a given representation, *e.g.*, source code, abstract syntax tree, bytecode, or binary code. In contrast, dynamic analysis tools observe the program being analyzed by collecting runtime information. Many dynamic analysis tools rely on instrumentation to achieve their goals.

In the context of the JVM, static analysis and instrumentation for dynamic analysis often happens on the level of Java bytecode. Analysis tools thus need to decode and analyze—and in the case of instrumentation also edit and encode—Java bytecode. Given the relative complexity of the Java class file format, a diverse set of libraries (see Section A.2) has been created for this purpose. All those libraries are implemented in Java.

Instrumentation at bytecode level can be done in two ways: using a JAVA instrumentation agent or using a native JVMTI agent.[1] A Java instrumentation

[1] `http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/index.html`

agent is written in Java and runs in the same JVM as the application. This leads to two main problems: poor isolation and poor coverage. It provides *poor isolation* because to instrument the VM, the agent's classes must be loaded in the same VM, and this can lead to perturbation in the VM. It provides *poor coverage* because an instrumentation agent (implemented in Java) will require some runtime library classes to be loaded before it can start instrumenting, and those runtime classes thus cannot be instrumented at load time.

A native JVMTI agent can instrument every class that the VM loads, including runtime classes. The main issue when using JVMTI is that instrumentation must be done in a native language, usually C or C++. Using C/C++ as the instrumentation language can be problematic, because of the lack of a C/C++ library for Java bytecode rewriting. Therefore developers have been using an extra JVM as an "instrumentation server" in which they could use Java-based bytecode rewriting libraries. The C/C++ JVMTI agent thus only has to send code to the server, and no native bytecode rewriting library is needed. However, this approach has a drawback: it requires an additional JVM, and it causes IPC traffic between the observed JVM and the instrumentation server.

We created JNIF to overcome this problem. To the best of our knowledge, JNIF is the first native Java bytecode rewriting library. JNIF is a C++ library for decoding, analyzing, editing, and encoding Java bytecode. The main benefit of JNIF is that it can be plugged into a JVMTI agent for instrumenting all classes in a JVM transparently, i.e., without connecting to another JVM and without perturbing the observed JVM.

Starting with JAVA 6, class files can include stack maps to simplify bytecode verification for the JVM. JAVA 7 made those stack maps mandatory. Thus, unless one wants to disable the JVM's verifier, code rewriting tools need to also generate stack maps. Stack maps contain, for each basic block, type information for each local variable and operand stack slot. To generate stack maps, a bytecode rewriting tool needs to perform a static analysis. Due to the fact that bytecode does not contain type declarations of variable slots and local variables, these types have to be inferred using an intra-procedural data flow analysis. For reference types, computing the least upper bound of two types in a join point of a control flow graph even requires access to the class hierarchy of the program. Thus, the seemingly innocuous requirement for stack maps significantly complicates the creation of a bytecode rewriting library. JNIF solves these issues, also thanks to the fact that it can be used in-process in a JVMTI agent, and thus can determine the necessary subtyping relationships by requesting the bytes of arbitrary classes loaded or loadable at any given point in time. This works for classes loaded via user-defined class loaders as well as for classes generated dynamically on-the-fly.

Overall, the main contributions of this paper are:

- We present JNIF, a C++ library for decoding, analyzing, editing, and encoding Java class files.
- JNIF includes a data-flow analysis for stack map generation, a complication necessary for any library that provides editing and encoding support for modern JVMs with split-time verification.
- We evaluate JNIF by comparing its performance against the most prevalent Java bytecode rewriting library, ASM.

The rest of this chapter is organized as follows: Section A.2 presents related work. In Section A.3 we show how to use the JNIF API. Section A.4 describes the design of JNIF. Section A.5 explains how we validated JNIF. Section A.6 evaluates JNIF's performance against the mainstream bytecode manipulator, ASM. Section A.7 discusses limitations, and Section A.8 concludes.

A.2 Related Work

We now discuss low-level Java bytecode rewriting libraries, JVM hooks for dynamic bytecode rewriting, high-level dynamic bytecode rewriting frameworks, and how they relate to JNIF.

Low-level rewriting libraries. JNIF certainly is not the first Java bytecode analysis and instrumentation framework. The probably earliest is BCEL[2], a well-designed Java library with a tree-based API. The probably most prevalent is ASM[3] [Bruneton et al., 2002; Kuleshov, 2007], which aims to be more efficient, especially due to the addition of a visitor-based streaming API, but which has a somewhat less encapsulated design. SOOT[4] [Vallée-Rai et al., 1999] is a Java bytecode optimization framework supporting whole-program analysis with four different intermediate representations: Baf, which is simple to manipulate, Jimple, which is easy to optimize, Shimple, an SSA-based variant of Jimple, and Grimp, focused on decompilation. WALA[5] is a framework for static analysis, which also includes SHRIKE[6], a library for instrumenting bytecode using a patch-based approach. Unlike the above libraries, Javassist[7] Chiba and Nishizawa

[2] http://commons.apache.org/bcel/
[3] http://asm.ow2.org/
[4] http://www.sable.mcgill.ca/soot/
[5] http://wala.sourceforge.net/
[6] http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview
[7] http://www.javassist.org/

[2003] provides an API for editing class files like they were Java source code, thereby enabling developers who do not understand bytecode to instrument class files.

Dynamic instrumentation hooks. The most limited way for dynamically rewriting JAVA classes at runtime is the use of a custom class loader. This requires modifications to the application, so that it uses that class loader. This can be problematic for applications—especially for large programs based on powerful frameworks—that already use their own class loaders. This limitation can be circumvented by using dynamic instrumentation hooks provided by the JVM Lindholm et al.. Java provides two such hooks: Java agents and JVMTI. Java agents[8] are supported via the `-javaagent` JVM command line argument. They are implemented in JAVA and use the `java.lang.instrument` package to interact with the JVM. This allows them to get notified when classes are about to get loaded, and it allows them to modify the class bytecode. They can also modify and reload already loaded classes, however the kinds of transformations allowed with class reloading are severely limited. JVMTI (the *Java Virtual Machine Tool Interface*) is a native API into the JVM that, amongst many other things, provides hooks that allow the rewriting of bytecode. The advantage of JVMTI over Java agents is that JVMTI allows the instrumentation of *all* Java classes, including the entire runtime library. Java also provides JDI[9] (the *Java Debug Interface*), a high-level interface on top of JVMTI to control a running application in a remote JVM.

High-level dynamic analysis frameworks. We now discuss dynamic analysis frameworks that are built on top of the previously mentioned rewriting libraries and use the above instrumentation hooks. These frameworks do not allow arbitrary code transformations and they shield the developer from the necessary instrumentation effort. Sofya[10] Kinneer et al. [2007] is a dynamic analysis framework that runs the analysis in a separate JVM from the observed application. It provides analysis developers with a set of observable events, to which the analyses can subscribe. Sofya combines bytecode instrumentation using BCEL with the use of JDI for capturing events. FERRARI Binder et al. [2007] is a dynamic bytecode instrumentation framework that combines static instrumentation of runtime library classes with dynamic instrumentation of application classes to achieve full coverage. FERRARI hooks into the JVM using a Java agent. DiSL Marek et al. [2012a,b] is a domain-specific aspect language for dynamic analysis. It eliminates the need for static instrumentation from FERRARI

[8] `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`

[9] `https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/`

[10] `http://sofya.unl.edu`

by using a separate JVM for instrumentation. It uses JVMTI to hook into the JVM and forwards loaded classes to an instrumentation server, where it performs instrumentation using the ASM rewriting library. Turbo DiSL Zheng et al. [2012] significantly improves the performance of DiSL by partially evaluating analysis code at instrumentation time. RoadRunner[11] Flanagan and Freund [2010] is a high-level framework for creating dynamic analyses focusing on concurrent programs. An analysis implemented on top of RoadRunner simply consists of analysis code in the form of a class that can handle the various event types (such as method calls or field accesses) that RoadRunner can track. RoadRunner uses a custom classloader to be able to rewrite classes at load time, and it uses ASM for bytecode rewriting. Btrace[12] is an instrumentation tool that allows developers to inject probes based on a predefined set of probe types (such as method entry, or bytecode for a specific source line number). Btrace uses the Java agent hooks and builds on top of ASM for instrumentation. Chord[13] Naik [2011] is a static analysis framework based on Datalog. It uses joeq[14] to decode classes and convert bytecode into a three-address quadcode internal representation for static analysis. Chord also supports dynamic analysis, for which it instruments programs using Javassist.

How JNIF differs. Similar to BCEL, JNIF is a low-level library that uses a clean object model to represent java class files. However, unlike all the libraries described above, JNIF is not implemented in Java, but in C++. This allows JNIF to be used directly inside a JVMTI agent. Java-based libraries do not allow dynamic instrumentation in this way: they either are limited to Java agents (which only provide limited coverage), or they require out-of-process instrumentation inside a second JVM (a so-called instrumentation server), and inter-process communication between the JVMTI agent and the instrumentation server.

JNIF simplifies the development and deployment of full-coverage dynamic analysis tools, because one does not need to run an instrumentation server in a separate JVM process. The fact that this is essential is demonstrated by the HPROF[15] profiling agent coming with the JVM. HPROF does not use Java libraries for rewriting bytecode, but implements (a limited form of) class file instrumentation as native code inside a JVMTI agent.

The high-level frameworks described above all abstract away from the underlying instrumentation approach. Thus, they could make use of JNIF to provide

[11] `http://dept.cs.williams.edu/~freund/rr/`
[12] `https://kenai.com/projects/btrace`
[13] `http://pag.gatech.edu/chord/`
[14] `http://joeq.sourceforge.net`
[15] `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`

their users with full-coverage while eliminating the need for a separate instrumentation server.

A.3 Using JNIF

We now briefly show how to use JNIF. JNIF can be used both in stand-alone tools or embedded inside a JVMTI agent. The complete API documentation and more extensive examples are available online[16]. Listing A.1 shows how to read and write a class file.

```
// Decode the binary data into a ClassFile object
const char  data = ...;
int len = ...;
jnif::ClassFile cf(data, len);

// Analyze or edit the ClassFile
...

// Encode the ClassFile into binary
int newlen = cf.computeSize();
u1  newdata = new u1[newlen];
cf.write(newdata, newlen);

// Use newdata and newlen
...

// Free the new binary
delete [] newdata;
```

Listing A.1. Decoding and encoding a class

JNIF's `ClassFile` class provides fields and methods for analyzing and editing a Java class. Listing A.2 shows how to traverse all methods in a class to dump their names and descriptors.

```
for (jnif::Method  m : cf.methods) {
  cout << "Method: ";
  cout << cf.getUtf8(m->nameIndex);
  cout << cf.getUtf8(m->descIndex);
  cout << endl;
}
```

Listing A.2. Traversing all methods in a class

[16] `http://acuarica.bitbucket.org/jnif/`

Listing A.3 shows how to find all constructors in a class and how to inject instrumentation, in the form of a call to a static method `static void alloc( Object o)` of an analysis class, at the beginning of each constructor.

```
ConstIndex mid = cf.addMethodRef(classIndex,
  "alloc", "(Ljava/lang/Object;)V");

for (Method  method : cf.methods) {
  if (method->isInit()) {
    InstList& instList = method->instList();
    Inst  p =  instList.begin();
    instList.addZero(OPCODE_aload_0, p);
    instList.addInvoke(OPCODE_invokestatic, mid, p);
  }
}
```

Listing A.3. Instrumenting constructor entries

Besides providing access to all members of a class, `ClassFile` also provides access to the constant pool via methods like `getUtf8()` and `addMethodRef()`.

This section shows common use cases of the JNIF library, such as writing instrumentation code and analyzing class files, thus giving an overview of the library. Its components are explained in more detail in section A.4. We present the examples in an incremental fashion, adding complexity in each example.

In order to be able to work with class files, they must me parsed. Given a buffer with a class file and its length, Listing A.4 shows how to parse it.

```
const char  data = ...;
int len = ...;

jnif::ClassFile cf(data, len);
```

Listing A.4. Decoding a class

The class `ClassFile` represents a JAVA class file and contains the definition for each method and fields. The full documentation can be found online.[17]

Once a class file is correctly parsed and loaded it can be manipulated using the methods and fields in `ClassFile`. For instance, in order to write back the parsed class file in a new buffer, the write method is used in conjunction with the computeSize method as shown in listing A.5.

```
const char  data = ...;
int len = ...;
jnif::ClassFile cf(data, len);
int newlen = cf.computeSize();
```

[17] `https://acuarica.gitlab.io/jnif/`

```
u1  newdata = new u1[newlen];
cf.write(newdata, newlen);

// Use newdata and newlen

delete [] newdata;
```

Listing A.5. Encoding a class

The `ClassFile` class has a collection of fields and methods which can be used to discover the members of the class file. The listing A.6 prints in the standard output every method's name and descriptor in a class file. Note that every jnif class overloads the `operator<<` in order send it to an std::ostream.

```
const char  data = ...;
int len = ...;
jnif::ClassFile cf(data, len);
for (jnif::Method  m : cf.methods) {
  cout << "Method: ";
  cout << cf.getUtf8(m->nameIndex);
  cout << cf.getUtf8(m->descIndex);
  cout << endl;
}
```

Listing A.6. Traversing all methods in a class

To hook every invocation of a constructor, a method named <init> in Java bytecode, one can traverse the method list and check whether the current method is an <init> method. Once detected, it is possible to add instrumentation code, like for instance call a static method in a given class. Figure A.7 shows how to add instruction to the instruction list.

```
ConstIndex mid = cf.addMethodRef(classIndex,
  "alloc", "(Ljava/lang/Object;)V");

for (Method  method : cf.methods) {
  if (method->isInit()) {
    InstList& instList = method->instList();

    Inst  p =  instList.begin();
    instList.addZero(OPCODE_aload_0, p);
    instList.addInvoke(OPCODE_invokestatic, mid, p);
  }
}
```

Listing A.7. Instrumenting constructor entries

Another common use case is to instrument every method entry and exit. In order to do so, one can add the instrumentation code at the beginning of the

instruction list to detect the method entry. To detect method exit, it is necessary to look for instructions that terminate the current method execution, i.e., xRETURN family and ATHROW as showed in figure A.8.

```
ConstIndex sid = cf.addMethodRef(proxyClass, "enterMethod",
        "(Ljava/lang/String;Ljava/lang/String;)V");
ConstIndex eid = cf.addMethodRef(proxyClass, "exitMethod",
        "(Ljava/lang/String;Ljava/lang/String;)V");
ConstIndex classNameIdx = cf.addStringFromClass(cf.thisClassIndex);

...

InstList& instList = method->instList();

ConstIndex methodIndex = cf.addString(m->nameIndex);

Inst  p =  instList.begin();

instList.addLdc(OPCODE_ldc_w, classNameIdx, p);
instList.addLdc(OPCODE_ldc_w, methodIndex, p);
instList.addInvoke(OPCODE_invokestatic, sid, p);

for (Inst  inst : instList) {
  if (inst->isExit()) {
    instList.addLdc(OPCODE_ldc_w, classNameIdx, inst);
    instList.addLdc(OPCODE_ldc_w, methodIndex, inst);
    instList.addInvoke(OPCODE_invokestatic, eid, inst);
  }
}
```

Listing A.8. Instrumenting <init> methods

A.4 JNIF Design and Implementation

JNIF is written in C++11 [ISO, 2012], in an object-oriented style similar to JAVA-based class rewriting APIs.

Design

JNIF consists of five main modules: model, parser, writer, printer, and analysis. *Model* implements JNIF's intermediate representation. It is centered around its `ClassFile` class. It is possible to create and manipulate class files from scratch. *Parser* implements the parsing of class files from a given byte array. The parser

parses a byte array and translates it to the model's IR. Once a `ClassFile` is created by the parser, it can be manipulated with the methods available in the model. *Writer* and *printer* represent two back-ends for the model. *Writer* serializes the entire `ClassFile` into a byte array ready to be loaded inside a JVM. *Printer* instead serializes the `ClassFile` into a textual format useful for debugging. Finally, *analysis* implements the static analyses necessary for computing stack maps.

JVM-Independence

JNIF is a stand-alone C++ library that can be used outside a JVM. It does not depend on JVMTI or JNI. However, for the purpose of stack map generation, it may need to determine the common super class of two classes. For this it will need to retrieve a class file given the name of an arbitrary class. This functionality is provided by a plugin that implements JNIF's `IClassPath`. JNIF comes with such a plugin that uses JNI in case it is running inside a JVM.

Explicit Constant Pool Management

Unlike some other class rewriting libraries, JNIF exposes the constant pool instead of hiding it. Our reasons for this design decision were two-fold: (1) We wanted to fully control the structure of the class file, and for that it is necessary to expose the constant pool. To reduce the additional complexity, we provide a rich set of methods that facilitate constant pool management. (2) We wanted to preserve, whenever possible, the original structure of the class file. This means that if one parses and then writes a class file, the original bytes will be obtained. This decreases the perturbation done by the instrumentation and allows for better testing.

Memory Management

Given that JNIF is implemented in an unmanaged language, we have to worry about memory deallocation. Our API follows a simple ownership model where all IR objects are owned by their enclosing objects. This means, that the `ClassFile` object owns the complete IR of a class. Our API design enforces this ownership model by requiring IR objects to be created by their enclosing objects. For example, to create a `Method`, one has to use the `ClassFile::addMethod()` factory method instead of directly allocating a new `Method` object.

Stack Map Generation

When encoding a `ClassFile` into a byte array, JNIF needs to generate stack maps. The necessary static analyses are implemented in the analyzer module. This module uses data flow analysis and abstract interpretation to determine the types of operand stack slots and local variables. The analysis module first builds a control flow graph of the method. The data flow analysis associates to each basic block an input and output stack frame, which represents the types of the local variables and operand stack slots at that point in the code. The input frame represents the type before any instruction in the basic block is executed. The output frame is computed by abstract interpretation of each instruction in the basic block. The entry basic block has an empty stack and each entry in its local variable table is set to top. Then the algorithm starts from the entry block and follows each edge. If a basic block is reachable from multiple edges, then a merge is involved.

Merging involves finding the least upper bound of multiple incoming types. While this is trivial for primitive types, it can require access to the class hierarchy for reference types. This requirement represents a severe complication for binary rewriting tools: when rewriting a single class, they may require access to many other classes in the program. JNIF solves this problem by providing the `IClassPath` interface. Different `IClassPath` implementations can provide different ways for getting access to classes. For example, a static instrumentation tool may use a user-defined class path to find classes, while a dynamic instrumentation tool may use JNI to request the bytes of a class given that class' name.

Running JNIF Inside a JVMTI Agent

When using JNIF inside a JVMTI agent, JNIF uses an `IClassPath` implementation that uses JNI to load the bytes of classes required for least upper bound computations during stack map generation.

Avoiding Premature Static Initialization

Using JNI to load a class (with `ClassLoader.loadClass()`), however, will call that class' static initializer. This is a side effect that may change the observable behavior of the program under analysis. To avoid this, one can request the bytes of the class (with `ClassLoader.getResourceAsStream()`) instead of loading the class. It can then parse the bytes of the class into its IR to determine that class' supertypes.

Avoiding the Loading of the Class Being Instrumented

If during the instrumentation of a class `X` JNIF needs to perform a least upper bound computation involving type `X`, then using `ClassLoader.loadClass()` to load class `X` would cause an infinite recursion. The above solution with `getResourceAsStream()` also prevents this problem.

Avoiding Premature ClassNotFoundException

If during the instrumentation of a class `X` JNIF needs to perform a least upper bound computation involving a type `Y`, and if class `Y` cannot be found, then throwing a `ClassNotFoundException` at that time would be premature (because without instrumentation, such an exception would only be thrown later). We solve that problem by assuming a least upper bound of `java.lang.Object` in that case.

A.5 Validation

We used a multitude of testing strategies to ensure JNIF is working correctly.

Unit Tests

JNIF includes a unit test suite that tests individual features of its various modules.

Integration Tests

Our integration test suite includes six different JNIF clients we run on over 40000 different classes. Each test reads, analyzes, and possibly modifies, prints, or writes classes from the Java runtime library (rt.jar), and all Dacapo benchmarks, Scala benchmarks, and the JRuby compiler.

testPrinter. This test parses and prints all classes. Its main goal is to cover the printing functionality. It has no explicit assertions. We consider it passed if it does not throw any exceptions.

testSize. This test covers the decoding and encoding modules. It asserts that the encoded byte array has the same length as the original byte array.

testWriter. This is similar to testSize, but it asserts that the contents of the encoded byte array is identical to the original bytes.

testNopAdderInstrPrinter. This also tests the instrumentation functionality, by injecting NOP instructions and dumping the result. It passes if it does not throw any exceptions.

testNopAdderInstrSize. This is similar to testSize, however it performs NOP injection. The resulting size must be identical to the original size plus the size of the injected NOP instructions.

testNopAdderInstrWriter. This is similar to testNopAdderInstrSize, but it asserts that the resulting array is identical except for the modified method bytecodes.

The "size" and "writer" tests work thanks to the fact that JNIF produces output identical to its input as long as classes are not modified and stack maps do not need to be re-generated.

Live Tests

Our live tests use JNIF inside a JVMTI dynamic instrumentation agent to ensure that the output of JNIF can successfully be loaded, verified, and run by a JVM. In addition to the aspects covered by the unit and integration tests, the live tests also validate that stack map generation works correctly, essentially by using the JVM's verifier to check correctness. For the live tests, we run a set of microbenchmarks, the Dacapo benchmarks, the Scala Benchmarking Project[18], and a microbenchmark using the JRuby compiler, with the goal of including InvokeDynamic bytecode instructions generated by JRuby.

Assertions and Checks

The JNIF code is sprinkled with calls to `Error::assert` that check preconditions, postconditions, and invariants. To provide a developer experience similar to Java's, all assertion violations print out call stack traces in addition to understandable error messages.

Moreover, JNIF checks its inputs (such as class files while parsing, or instrumented code while generating stack maps), and it calls `Error::check` to throw exceptions with stack traces and helpful messages when checks fail.

[18] `http://www.benchmarks.scalabench.org/`

A.6 Performance Evaluation

We evaluated the performance of a JNIF-based dynamic instrumentation approach versus an approach using an ASM-based instrumentation server.

Measurement Contexts

We ran our experiments on three different machines: (1) A machine with two Intel Xeon E5-2620 2 GHz CPUs, each with 6 cores and 2 threads per core, and 8 GB RAM, running Debian Linux x86 64 3.10.11-1. (2) A Dell PowerEdge M620, 2 NUMA node with 64 GB of RAM, Intel Xeon E5-2680 2.7 GHz CPU with 8 cores, CPU frequency scaling and Turbo Mode disabled, running Ubuntu Linux x86 64 3.8.0-38. For consistent memory access speed, we bound our program to a specific NUMA node using `numactl`. (3) A MacBook Pro with an Intel Core i7 2.7 GHz CPU with 4 cores and 16 GB running Mac OS X 10.8.2.

Benchmarks

We used the Dacapo benchmarks, except for tradebeans and tradesoap, which suffer from a well known issue[19]. We also include the Scala benchmarks (except for the subset identical to Dacapo).

Subjects

We compare JNIF to ASM for the purpose of performing dynamic instrumentation. For JNIF we built a JVMTI agent that directly includes JNIF to instrument loaded classes. For ASM, we use a JVMTI agent that forwards loaded classes to an instrumentation server that uses ASM's streaming API (which is faster than ASM's tree API).

Results

Figure A.1 shows the results of our performance evaluation in terms of time spent instrumenting classes. The figure shows the results from our first machine. The other machines produced results similar to Figure A.1, and we omit them for space reasons. The figure shows box plots summarizing five measurement runs. It shows one box for JNIF and two boxes for ASM. The "ASM Server" box

[19] `http://sourceforge.net/p/dacapobench/bugs/70/`

represents the time as measured on the instrumentation server. This is equivalent to the time a static instrumentation tool would take. It excludes the time spent in the JVMTI agent and the time for the IPC between the agent and the server. The "ASM Server on Client" box represents the total time needed for instrumentation, as measured in the JVMTI agent, and thus includes the IPC and JVMTI agent time.

Each chart in the figure consists of five groups of boxes: "Empty" is the time when using a JVMTI agent that does not process bytecodes at all. "Identity" is for an agent that simply decodes and encodes each class, without any instrumentation, and without recomputing stack maps. "ComputeFrames" also includes recomputing stack maps. "Allocations" represents a useful dynamic analysis that captures all allocations. "Nop Padding" is a different dynamic analysis that injects NOPs after each bytecode instruction.

The figure shows that frame computation adds significant overhead, on ASM as well as JNIF. Moreover, it shows that except for dacapo-eclipse, dacapo-jython, and scala-scalatest, JNIF is faster even than just the ASM Server time.

Reproducibility

To run these evaluations, a Makefile script is provided in the git repository. These tasks take care of the compilation of the JNIF library and also all java files needed. The repository is self-contained, no need to download dacapo benchmarks separately.

```
> make testapp
```

Listing A.9. Running testapp

```
> make testapp
```

Listing A.10. Running dacapo

To run a particular dacapo benchmark with default settings

```
> make dacapo BENCH=avrora
```

Listing A.11. Running dacapo

To run a full evaluation with all dacapo benchmarks in all configuration a task -eval- is provided. You can set how many times run each configuration with the variable times, like

```
> make eval times=5
```

Listing A.12. Running full eval five times

Finally, there is a task to create plots for the evaluation. This task needs R with the package ggplot2.

```
> make plots
```

Listing A.13. Plots

A.7 Limitations

JNIF still has some limitations.

jsr/ret. JNIF does not support stack map generation for jsr and ret. Class files requiring stack maps do not include jsr/ret.

invokedynamic. JNIF's support for invokedynamic is not yet fully tested, but our initial tests with JRuby have been successful (using `-Djruby.compile.invokedynamic=true`).

Stack map generation with full coverage. When the JVM loads the first few runtime library classes, and calls the JVMTI agent to have those classes instrumented, it is still too early to use JNI for loading classes needed for computing least upper bounds for stack map generation. For this reason, we do not generate stack maps for runtime library classes. This no problem, because the JVM does not verify the runtime library classes by default, and thus it does not need stack maps for those classes. However, should developers decide to explicitly turn on the verification of runtime library classes (with `-Xverify:all`), the verifier would complain because JNIF would not have generated stack maps.

To get full coverage for the instrumentation inside a JVMTI agent, it is necessary to instrument every class, even the whole java class library. If the instrumentation needs to change or add branch targets, the compute frames option must be used, but it cannot be used against the class library, because to compute frames, the class hierarchy must be known, and this imposes a depedency with a classloader which is not yet available.

Luckily, by default the Java library classes are not verified, because they are trusted. Thus the instrumentation only needs to compute frames on classes not belonging to java library.

A.8 Conclusions

Until now, full-coverage dynamic instrumentation in production JVMs required performing the code rewriting in a separate JVM, because of the lack of a native bytecode rewriting framework. This paper introduces JNIF, the first full-coverage in-process dynamic instrumentation framework for Java. It discusses the key

issues of creating such a framework for Java—such as stack-map generation—and it evaluates the performance of JNIF against the most prevalent Java-level framework: ASM. We find that JNIF is faster than using out-of-process ASM in most cases. We hope that thanks to JNIF, and this paper, a broader number of researchers and developers will be enabled to develop native JVM agents that analyze and rewrite Java bytecode without limitations.

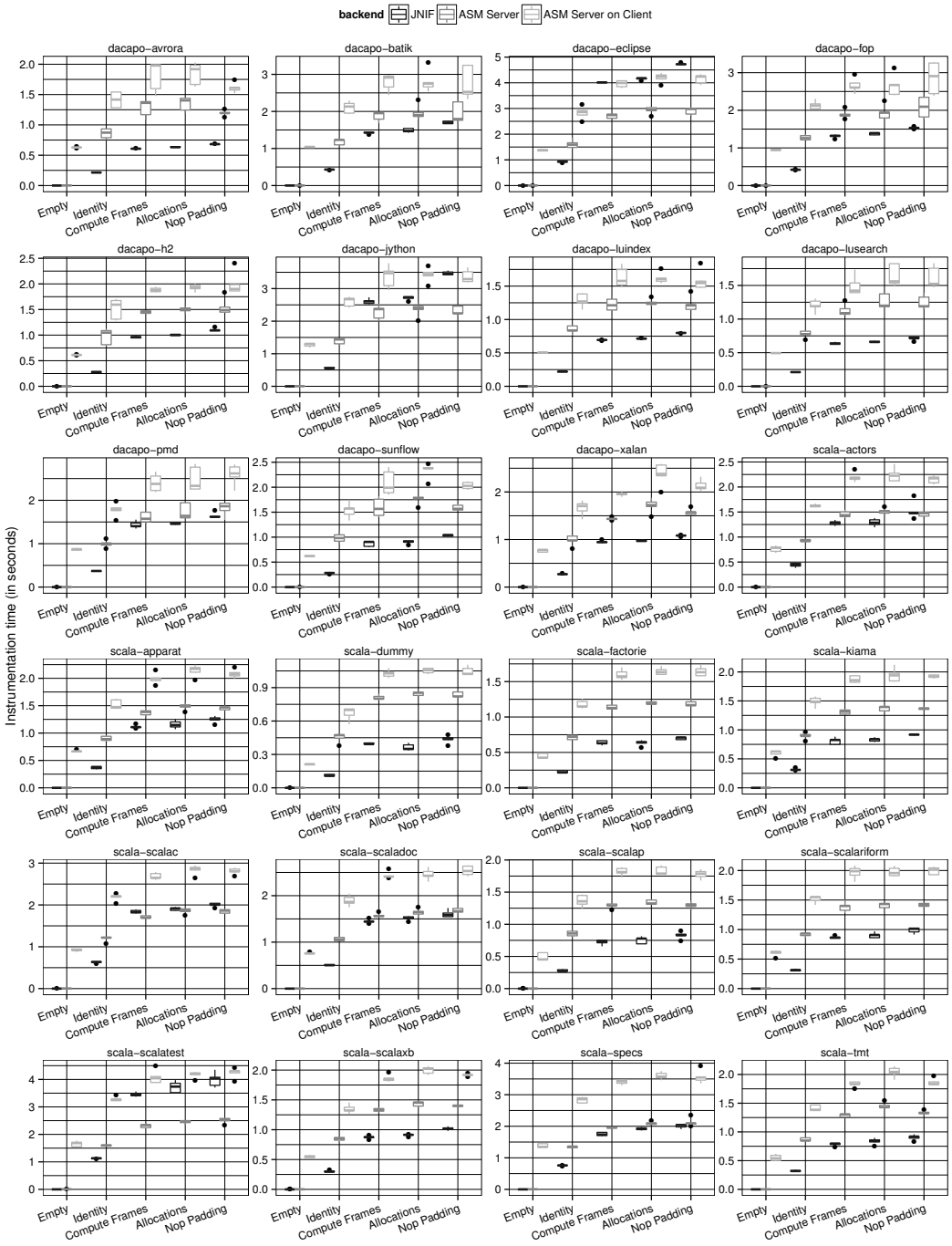


Figure A.1. Instrumentation time on DaCapo and Scala benchmarks

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.

David F. Bacon, Perry Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-628-9. doi: 10.1145/604131.604155.

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.

W. Binder, J. Hulaas, and P. Moret. Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 91–100, September 2007. doi: 10.1109/SCAM.2007.20.

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.

Shigeru Chiba and Muga Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, Lecture Notes in Computer Science, pages 364–376. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-39815-8.

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.

Erik Ernst. Family Polymorphism. In *ECOOP 2001 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 303–326. Springer, Berlin, Heidelberg, June 2001. ISBN 978-3-540-42206-8 978-3-540-45337-6. doi: 10.1007/3-540-45337-7_17.

Cormac Flanagan and Stephen N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0082-7. doi: 10.1145/1806672.1806674.

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.

James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013. ISBN 0-13-326022-4 978-0-13-326022-9.

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.

ISO. *ISO/IEC 14882:2011 Information Technology — Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

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.

A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java. In *29th International Conference on Software Engineering - Companion, 2007. ICSE 2007 Companion*, pages 51–52, May 2007. doi: 10.1109/ICSECOMPANION.2007.68.

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.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification. page 626.

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.

Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development - AOSD '12*, page 239, Potsdam, Germany, 2012a. ACM Press. ISBN 978-1-4503-1092-5. doi: 10.1145/2162049.2162077.

Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi, and Petr Tuma. DiSL: An Extensible Language for Efficient and Comprehensive Dynamic Program Analysis. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1128-1. doi: 10.1145/2162037.2162046.

Luis Mastrangelo and Matthias Hauswirth. JNIF: Java Native Instrumentation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 194–199, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2926-2. doi: 10.1145/2647508.2647516.

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.

Mayur Naik. Chord: A Versatile Platform for Program Analysis. 2011.

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.

Fridtjof Siebert. Eliminating External Fragmentation in a Non-moving Garbage Collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM. ISBN 978-1-58113-338-7. doi: 10.1145/354880.354883.

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.

Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, April 2000. ISSN 1388-3690, 1573-0557. doi: 10.1023/A:1010000313106.

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.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative Object Identity Using Relation Types. In *ECOOP 2007 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 54–78. Springer, Berlin, Heidelberg, July 2007. ISBN 978-3-540-73588-5 978-3-540-73589-2. doi: 10.1007/978-3-540-73589-2_4.

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.

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.

Yudi Zheng, Danilo Ansaloni, Lukas Marek, Andreas Sewe, Walter Binder, Alex Villazón, Petr Tuma, Zhengwei Qi, and Mira Mezini. Turbo DiSL: Partial Evaluation for High-Level Bytecode Instrumentation. 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, Carlo A. Furia, and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, volume 7304, pages 353–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30560-3 978-3-642-30561-0. doi: 10.1007/978-3-642-30561-0_24.

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.