Università
della
Svizzera
italiana

**Faculty
of Informatics**

Dissertation Proposal

June 27, 2018

What Do We Write?

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

Luis Mastrangelo

Abstract

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

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

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

Research Advisor
Prof. Matthias Hauswirth

Research Co-advisor
Prof. Nathaniel Nystrom

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

External Committee Members
Prof. Hridesh Rajan?, Prof. Tobias Wrigstad?, Prof. Stefan Hanenberg?

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

_____

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

_____

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

_____

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

_____

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

_____

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

_____

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

_____

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

_____

Prof. Tobias Wrigstad?, Uppsala University, Sweden?

_____

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

_____

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

_____

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

Contents

Chapter 1

Introduction

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

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

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

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

Looking at the aforementioned examples, **?** and **?** studied how developers use lambdas

---

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

in JAVA and C++ respectively; while **??** did the same for generics in JAVA. This kind of studies give an insight of the adoption of lambdas and generics; which can drive future direction for language designers and tool builders, while providing developers with best practices.

1.1 Research Question

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

Research Question

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

We believe that we — as a research community — should understand what kinds of programs are written in real codebases. We can use this information to improve several aspects of the software development process and supporting informed decisions for the driving forces mentioned above. This fact opens the door for empirical studies about language features and their use in source code repositories, *e.g.*, *GitHub*[7], *GitLab*[8] or *Bitbucket*[9], and package managers repositories, *e.g.*, *Maven Central*[10] or *npm*[11]. Since any kind of language study must be language-specific, our plan is to focus on JAVA given its wide usage and relevance for both research and industry.

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

Table 1.1: Per Feature Research Questions

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

---

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

1.2   Proposal Outline

The rest of this proposal is organized as follows: Chapter **??** gives a review of the literature in the *state-of-the-art* of the different aspects related to our goal. More specifically, Chapter **??** presents already existing code patterns related to the language features we plan to analyze. The following four chapters introduce our proposal plan for the selected features: Chapters **??**, **??**, **??** presents our *casting, reflection* and *exception handling* study respectively. Finally, Chapter [cha:unsafe]] shows the study we already made on the unsafe API in JAVA.

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

Chapter 2

Understanding How JAVA Language Features Are Used

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

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

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

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

Chapter 3

Reflection Patterns

This list of patterns are more of semantic patterns.
    When reflection and metaprogramming can be used.
    Related Work

Chapter 4

Literature Review

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

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

?

4.1 Existing Code Patterns

1. Specifying Application Extensions

   (a) Snippet

```
public void addHandlers(String path) {
    XmlIO xmlFile = new XmlIO(DiskIO.getResourceURL(path));
    xmlFile.load();
    XmlElement list = xmlFile.getRoot().getElement("handlerlist");
    Iterator it = list.getElements().iterator();
    while (it.hasNext()) {
        XmlElement child = (XmlElement) it.next();
        String id = child.getAttribute("id");
```

Table 4.1: Existing Patterns

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

```
                              String clazz = child.getAttribute("class");
                              AbstractPluginHandler handler = null;
                              try {
                                  Class c = Class.forName(clazz);
                                  handler = (AbstractPluginHandler) c.newInstance();
                                  registerHandler(handler);
                              } catch (ClassNotFoundException e) {
                                  if (Main.DEBUG) e.printStackTrace();
                              } catch (InstantiationException e1) {
                                  if (Main.DEBUG) e1.printStackTrace();
                              } catch (IllegalAccessException e1) {
                                  if (Main.DEBUG) e1.printStackTrace();
                              }
                          }
                      }
```

(b) Discussion

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

2. Custom-made Object Serialization Scheme

(a) Snippet

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

(b) Discussion

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

3. Improving Portability Using Reflection

(a) Snippet

```
try {
            Class macOS  = Class.forName("gruntspud.standalone.os.MacOSX"
            Class argC[] = {ViewManager.class};
            Object arg[] = {context.getViewManager()};
            Method init = macOS.getMethod("init", argC);
            Object obj  = macOS.newInstance();
            init.invoke(obj, arg);
        } catch (Throwable t) {
```

```
                    // not on macos
        }

        Method m = c.getMethod("clone", null);
        if (Modifier.isPublic(m.getModifiers())) {
            try {
                result = m.invoke(object, null);
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }

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

(b) Discussion

What can we say?

4. Code Unavailable Until Deployment

(a) Snippet

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

(b) Discussion

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

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

(a) Snippet

```
public JDBCCategoryDataset(String url, String driverName,
                            String user, String passwd)
    throws ClassNotFoundException, SQLException
```

```
{
    Class.forName(driverName);
    this.connection = DriverManager.getConnection(url, user, passwd);
}
```

(b) Discussion

Commonly used by JDBC API to load database drivers.

6. Getting Around Static Type Checking

(a) Snippet

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

(b) Discussion

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

7. Providing a Built-in Interpreter

(a) Snippet

(b) Discussion

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

8. Guarded Casts

(a) Snippet

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

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

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

`dead-if-guarded` cast version

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

ensure-guarded casts

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

while-guarded cast

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

9. Semi-guarded Casts

  (a) Snippet

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

10. Unguarded Casts

  (a) Snippet

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

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

11. Safe Casts

  (a) Snippet

```
(char)42
```

```
        (Integer)42
```

12. CorrectCasts

13. WellBehavedClassLoaders

14. IgnoringExceptions1

15. IgnoringExceptions2

16. IndexedCollections

17. MetaObjectsInTables

18. MultipleMetaObjects

19. EnvironmentStrings

20. UndecidableFiltering

21. NoProxy

4.2   Compilers Writers

Already **?** started to study FORTRAN programs. By knowing what kind of programs arise in practice, a compiler optimizer can focus in those cases, and therefore can be more effective. Alternatively, to measure the advantages between compilation and interpretation in BASIC, **?** has studied a representative dataset of programs. Adding to Knuth's work, **?** made an empirical study for parallelizing compilers. Similar works have been done for COBOL **??**, PASCAL **?**, and APL **??** programs.

But there is more than empirical studies at the source code level. A machine instruction set is effectively another kind of language. Therefore, its design can be affected by how compilers generate machine code. Several studies targeted the JVM **???**; while **?** did a similar study for Lilith in the past.

4.3   Benchmarks and Corpuses

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

Another suite is given in **?**. They provide a corpus of curated open source systems to facilitate empirical studies on source code.

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

4.4   Large-scale Codebase Empirical Studies

In the same direction to our plan, **?** perform a study of the dynamic features of SMALLTALK. Analogously, **??** made a similar study, but in this case targeting JAVASCRIPT's dynamic behavior and in particular the `eval` function. Also for JAVASCRIPT, **?** analyzed how fields are accessed via strings, while **?** analyzed privacy violations. Similar empirical studies were done for PHP **???** and SWIFT **?**.

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

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

Exceptions

?? focus on exceptions. They made empirical studies on how programmers handle exceptions in JAVA code. The work done by **?** categorized them in patterns. Whether **?** used a more dynamic approach by analysing stack traces and code issues in *GitHub*.

Collections and Generics

The inclusion of generics in JAVA is closely related to collections. **??** studied how generics were adopted by JAVA developers. They found that the use of generics do not significantly reduce the number of type casts.

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

4.5   Controlled Experiments on Subjects

There is an extensive literature *per se* in controlled experiments on subjects to understand several aspects in programming, and programming languages. For instance, **?** tried to understand the how expert programmers face problem solving. **?** made a empirical study on how effective is mutation testing. **?** compared how a given — fixed — task was implemented in several programming languages.

? realize that, in essence, programmers need to answer reachability questions to understand large codebases.

Several authors **???** measure whether using a static-type system improves programmers productivity. They compare how a static and a dynamic type system impact on productivity. The common setting for these studies is to have a set of programming problems. Then, let a group of developers solve them in both a static and dynamic languages.

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

The work by **??** goes towards this direction. They have examined programs written by students to understand real debugging conditions. Their focus is on ill-typed programs written in HASKELL. Unfortunately, these dataset does not correspond to real-world code. Our focus is to analyze code by experienced programmers.

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

4.6   Code Patterns Discovery

? have extended ASM **??** to implement symbolic execution and recognize call sites. However, this is only a meta-pattern detector, and not a pattern discovery. **?** used both dynamic and static analysis to discover design patterns, while **?** used only dynamic.

Trying to unify analysis and transformation tools **?, ?** built *Rascal*, a DSL that aims to bring them together.

4.7   Tools for Mining Software Repositories

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

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

To this end, **?** provides an offline mirror of *GitHub* that allows researchers to query any kind of that data. Later on, **?** published the dataset construction process of *GitHub*.

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

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

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

4.8   Selecting Good Representatives

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

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

---

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

Chapter 5

The Unsafe API

The material in this chapter is based on our previously published paper [**?**].

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

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

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

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

Chapter 6

Casts

6.1 Abstract

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

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

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

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

6.2 Introduction

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

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

---

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

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

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

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

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

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

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

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

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

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

6.3   Related Work

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

?

?

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

---

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

6.4 Casts

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

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

```
1  Object o = "foo";
2  String s = (String)o;
```
Listing 6.1: Variable o is defined as Object, then casted to String.

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

```
1  Object x = new Integer(0);
2  System.out.println((String)x);
```
Listing 6.2: Incompatible types throwing `ClassCastException` at runtime.

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

```
1  try {
2     Object x = new Integer(0);
3     System.out.println((String)x);
4  } catch (ClassCastException e) {
5     System.out.println("");
6  }
```
Listing 6.3: Catching `ClassCastException`

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

```
if (x instanceof Foo) {
   ((Foo)x).doFoo();
}
```
Listing 6.4: Use of `instanceof` operator to test whether a reference is of certain type.

---

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

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

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

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

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

6.5 Study Overview

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

6.6 Are they *casts* operator used?

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

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

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

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

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

---

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

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

So, yes, cast are used.

6.7   **Finding *casts* Usage Patterns**

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

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

Then we have used ASM **?**

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

checkcast as specified by: [11]

---

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

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

- **local**

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

- **fermat**

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

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

6.8   Preliminary Considerations

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

6.8.1   Simple cast

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

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

6.8.2   Generics vs. Non-generics

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

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

---

[12]https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html\#jvms-6.5.
instanceof
[13]https://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html

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

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

6.8.3 Upcast

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

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

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

6.8.4 Conditional Operator

Using the conditional operator produces the following bytecode. [MavenDS](https://bitbucket.org/acuarica/mavends)
   [JNIF](https://bitbucket.org/acuarica/jnif)
   ### Queries
   To retrieve the stats showed above, we have used SQL queries against the bytecode database. Each individual query is aimed to answer a precise question. The following list presents all the SQL queries used to retrieve the stats, and its respective answer (after the ';').

   1. [How many checkcast instructions?](sql/checkcast-count.out)

   2. ['checkcast' most used arguments](sql/checkcast-most-used-args.out)

3. ['checkcast' most used targets](sql/checkcast-most-used-target.out)

4. [How many classes?](sql/class-count.out)

5. [How many bytecode instructions?](sql/code-count.out)

6. [How many 'equals' methods?](sql/equals-method-count.out)

7. [How many 'equals' methods with 'checkcast'?](sql/equals-method-w-checkcast-count.out)

8. [How many 'equals' methods with 'instanceof'?](sql/equals-method-w-instanceof-count.out)

9. [How many 'instanceof' instructions?](sql/instanceof-count.out)

10. ['instanceof' most used arguments](sql/instanceof-most-used-args.out)

11. ['instanceof' most used targets](sql/instanceof-most-used-target.out)

12. [How many '.jar' files?](sql/jar-count.out)

13. [How many '.jar' files with classes?](sql/jar-w-classes-count.out)

14. [How many methods?](sql/method-count.out)

15. [How many methods with 'checkcast' instruction?](sql/method-w-checkcast-count.out)

16. [How many methods with 'instanceof' instruction?](sql/method-w-instanceof-count.out)

17. [How many methods with signature?](sql/methods-w-signature.out)

6.9   Complex Analysis

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

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

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

6.10   Preliminary

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

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

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

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

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

—

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

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

by setTag.

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

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

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

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

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

- Are casts local? In a given project, are casts limited to just a few classes or are they widespread?

- How many casts are dominated by an instanceof in the same method (or in another method)?

- How many typecase are "real" in that there's actually more than one alternative?

- How often does it happen that an unguarded cast cannot possibly fail in any execution (for instance when there's really only one class implementing an interface and therefore a cast (to the class) must succeed because there are no objects of any other class). I'm not sure if this is the right why to ask the question.

6.11   Casts Usage Patterns

6.12   Discussion

Here we discuss.

6.13   Related Work

Relwork.

6.14   Conclusions

asdf

6.15   Latex

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

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

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

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

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

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

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

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

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

? Similar to Challenges . . .

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

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

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

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

6.15.1 Exception Handling

Android~**?**

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

6.15.2 Evidence Languages

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

Chapter 7

Casts Discovery

7.1 Select All Expressions

import java

```
from  Expr  e
select  e
```

7.2 All casts

Snippet to select all cast expressions.

import java

```
from  CastExpr  ce
select  ce
```

Chapter 8

Casts Detection

8.1 Lookup by ID (135 casts)

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

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

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

import java

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

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

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

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

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

8.2.1   Description

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

8.3   Typecase (55 instanceof, 65 casts)

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

8.4   Factory method (26 casts, including 24 redundant)

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

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

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

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

```
import java

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

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

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

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

8.7   container object (16 casts)

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

8.8   testing (13 instanceof, 3 casts)

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

8.9   null (11 casts)

Cast to null to resolve method overloading ambiguity

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

import java

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

8.10   query result (11 casts)

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

8.11   Payload (10 casts)

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

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

8.12   lookup by type tag (9 casts)

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

8.13   Argument check (6 instanceof, 3 casts)

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

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

Cast result of field access or method invocation using reflection.

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

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

8.15 Stash (8 casts)

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

8.16 Object in collection (8 casts)

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

Includes one overly complicated use of Java 8 streams.

8.17 covariant field of supertype (8 casts)

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

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

typically just a method wrapping an instanceof

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

```
import java

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

8.19   type parameter (7 casts)

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

8.20   newInstance (1 instanceof, 5 casts)

cast result of Class or Array.newInstance

8.21   Redundant cast (6 casts)

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

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

import java

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

8.22   add type parameters (6 casts)

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

8.23   remove type parameter (5 casts)

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

8.24   readObject (4 casts)

cast result of readObject()

8.25   exception for rethrow (2 instanceof, 2 casts)

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

8.26   Covariant return (3 casts)

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

8.27   result check (2 instanceof, 1 cast)

check result of a call has the right type

8.28   Clone (2 casts)

cast result of clone()

8.29   Throwable.getCause (2 casts)

Throwable.getCause has type Throwable, cast to Exception

8.30   Library method returning Object (2 casts)

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

8.31   method argument of type Object (1 cast)

overridden method takes an Object not something more specific

8.32   Heterogeneous collections (1 cast)

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

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

8.33   URL.openConnection (1 cast)

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

8.34   Result of binary operation (1 cast)

Cast result of binary operation to subtype.

8.35   Type parameter resolution (1 cast)

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

8.36   Global flag (1 cast)

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

8.37   Singleton

Unguarded pattern

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

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

Chapter 9

Exceptions

Here we talk about exception, maybe?