Casting about in the Dark

An Empirical Study of Cast Operations in Java Programs

LUIS MASTRANGELO, Università della Svizzera italiana, Switzerland

MATTHIAS HAUSWIRTH, Università della Svizzera italiana, Switzerland

NATHANIEL NYSTROM, Università della Svizzera italiana, Switzerland

The main goal of a static type system is to prevent certain kinds 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 *casting*.

We want to understand how and when developers escape the static type system to use dynamic typing. We empirically study how casting is used by developers in more than seven thousand Java projects. We find that casts are widely used (8.7% of methods contain at least one cast) and that 50% of casts we inspected are not guarded locally to ensure against potential run-time errors.

To help us better categorize use cases and thus understand how casts are used in practice, we identify 25 cast-usage patterns—recurrent programming idioms using casts to solve a specific issue. 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.

CCS Concepts: • **Software and its engineering** → **General programming languages; Object oriented languages; Software libraries and repositories.**

Additional Key Words and Phrases: cast, type safety, Java

ACM Reference Format:

Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 158 (October 2019), 31 pages. <https://doi.org/10.1145/3360584>

1 INTRODUCTION

The main goal of a *static* type system is to prevent certain kinds of errors from happening at run time. A type system is formulated as a set of constraints that gives expressions or terms in a program well-defined types. Any program not satisfying the constraints specified by the type system is rejected by the compiler.

Nevertheless, often a static type system is insufficiently precise. The type checker is necessarily conservative: it should not accept invalid programs, but it may reject programs that are valid but whose validity cannot be ensured at compile time. Moreover, there are situations when the

Authors' addresses: Luis Mastrangelo, Faculty of Informatics, Università della Svizzera italiana, Lugano, 6900, Switzerland, luis.mastrangelo@usi.ch; Matthias Hauswirth, Faculty of Informatics, Università della Svizzera italiana, Lugano, 6900, Switzerland, matthias.hauswirth@usi.ch; Nathaniel Nystrom, Faculty of Informatics, Università della Svizzera italiana, Lugano, 6900, Switzerland, nate.nystrom@usi.ch.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2019 Copyright held by the owner/author(s).

2475-1421/2019/10-ART158

<https://doi.org/10.1145/3360584>

developer has more information about the program than can be encoded—or encoded easily—into the types. To that end, programming languages often provide mechanisms to make the typing constraints less strict, allowing more valid programs at the expense of more errors at run time.

A common mechanism for circumventing static typing in object-oriented languages is *casting*. In programming languages with subtyping such as Java or C++, casting allows an expression to be viewed as a different type than the type system would prescribe. Casts are checked dynamically to ensure that the object being cast is an instance of the desired type.

We aim to understand why developers use casts. Why is the static type system insufficient, requiring an escape hatch into dynamic type checking? Specifically, we attempt to answer the following three research questions:

RQ1 : How frequently is casting used in common application code? To what extent does application code actually use casting operations?

RQ2 : How and when are casts used? If casts are used in application code, how and when do developers use them?

RQ3 : How recurrent are the patterns for which casts are used? When using casts, how often do developers resort to certain idioms to solve particular problems.

To answer these research questions, we identify *usage patterns*. Usage patterns are *recurrent programming idioms* used by developers to solve a specific issue. Usage patterns enable the categorization of different kinds of cast usages and thus provide insights into how the language is being used by developers in real-world applications. Our cast-usage patterns can be: (1) a reference for current and future language designers to make more informed decisions about programming languages, e.g., the addition of *smart casts* in Kotlin,¹ (2) a reference for tool builders, e.g., by providing more precise or new refactoring or code-smell analyses, (3) 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 (4) a guide for developers for better practices. To answer our research questions, we empirically study how casts are used by developers. We focus on Java due to its wide usage and relevance for both research and industry.

1.1 Casts in Java

While casts should be familiar to most programmers of object-oriented languages, because they have different semantics in different programming languages we briefly summarize the meaning of casts in Java and the terminology used in the rest of this paper.

A cast operation, written $(T) e$ in Java consists of a *target type* T and an *operand* e . The operand evaluates to a *source value* which has a run-time *source type*. In Java, a source reference type is always a class type or array type. For a particular cast evaluated at run time, the *source* of the cast is the expression in the program that created the source value. For reference casts, the source is an object allocation. The source may or may not be known statically.

An *upcast* occurs when the cast is from a source reference type S to a target reference type T , where T is a supertype of S . In our terminology, upcasts include identity casts where the target type is the same as the type of the operand. An upcast does not require a run-time check. A *downcast*, on the other hand, occurs when converting from a source reference type S to a target reference type T , where S is not a subtype of T . In type-safe OO languages, downcasts require a run-time check to ensure that the source value is an instance of the target type. This run-time check can either *succeed* or *fail*, throwing `ClassCastException` when it fails. This exception is unchecked in Java, i.e., the programmer is neither required to handle it nor to specify the exception in the method signature.

¹<https://kotlinlang.org/docs/reference/typecasts.html#smart-casts>

A *guard* is a conditional expression on which a (down)cast is control-dependent and that ensures that the cast is evaluated only if it will succeed. Guards are often implemented using the `instanceof` operator, which tests if an expression is an instance of a given reference type. If an `instanceof` guard returns true, the guarded cast should not throw a `ClassCastException`.

In Java, an object's type can also be checked using reflection: the `getClass` method² returns the run-time class of an object. This `Class` object can then be compared against a class literal, e.g., `x.getClass() == C.class`. This test is more precise than an `x instanceof C` test since it succeeds only when the operand's class is exactly `C`, rather than any subtype of `C`.

Because they can fail, downcasts pose potential threats. Unguarded downcasts in particular are worrisome because the developer is essentially telling the compiler “*Trust me, I know what I'm doing.*” Because downcasts are an escape-hatch from the static type system—they permit dynamic type errors—a cast is often seen as a design flaw or code smell in an object-oriented system [Tufano et al. 2015]. A cast can also be rejected at compile time if the cast operand and the target type are incompatible. For instance, in the expression `(String) new Integer(1)` a value of type `Integer` can *never* be converted to `String`, so the compiler rejects the cast expression.

Another form of casts in Java are *primitive conversions*, or more specifically *numeric conversions*. These are conversions from one primitive (non-reference) type, usually a numeric type, to another. These conversions can result in loss of precision of the numeric value, although they do not fail with a run-time exception.

Boxing and *unboxing* occur when casting from a primitive type to a reference type or vice versa, e.g., `(Integer) 3` converts the primitive `int 3` into a boxed `java.lang.Integer`. Like downcasts, unboxing casts can fail at run time if the source value cannot be converted to the target type. Java supports *autoboxing* and *autounboxing* between primitives and their corresponding boxed type.

Generics were introduced into Java to provide more static type safety. For instance, the type `List<T>` contains only elements of type `T`. The underlying implementation of generics, however, erases the actual type arguments when compiling to bytecode. To ensure type safety in the generated bytecode, the compiler inserts cast instructions into the generated code. Improper use of generic types or mixing of generic and raw types can lead to dynamic type errors—i.e., `ClassCastException`. Our study, however, does not consider these compiler-inserted casts. We are only concerned with programmer-inserted casts in the source code, not in the generated code.

1.2 Issues Developers have Applying the Cast Operator

Several studies [Coelho et al. 2015; Kechagia and Spinellis 2014; Zhitnitsky 2016] suggest that in Java, the `ClassCastException` is in the top ten of exceptions being thrown when analyzing stack traces. To illustrate the sort of problems developers have when applying casting conversions, we performed a search for commits and issues including the term `ClassCastException` within projects marked as using the Java language on *GitHub*. Our searches returned about 171K commits³ and 73K issues,⁴ respectively, at the time of this writing. At first glance, these results indicate that `ClassCastException` indeed represents a source of problems for developers.

Typical classes of bugs encountered when using a cast are using the wrong target type, using the wrong operand, or failing to guard a cast. More subtle, however, is the interaction between casting and generics. For example, the following call to `getProperty` throws a `ClassCastException`.

²<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass-->

³<https://github.com/search?l=Java&q=ClassCastException&type=Commits>

⁴<https://github.com/search?l=Java&q=ClassCastException&type=Issues>

```
config.getProperty("peer.p2p.pingInterval", 5L) http://bit.ly/ethereum\_ethereumj\_2vw4lf8
<T> T getProperty(String propName, T defaultValue) http://bit.ly/ethereum\_ethereumj\_getProperty\_2vwQIBH
```

The first argument to the method is the name of a property, used to lookup a value in a table. The second argument is a default value to use if the property is not in the table. If the lookup is successful, the method casts the value found to type `T`. In the call, the given property `"peer.p2p.pingInterval"` is in the table and mapped to an `Integer`. However, Java uses the type of the `defaultValue` argument, in this case `Long`, to instantiate the type parameter `T`. Note, however, that the cast inside `getProperty`, which in this context should cast from `Integer` to `Long`, *does not fail*. This is because the Java compiler erases type parameters like `T` and so dynamic type tests are not performed on them. Instead, the compiler inserts a cast where the return value of `getProperty` is used later with type `Long`. It is this cast that fails and that is reported at run time.

The fix for this bug is to change the default value argument from `5L` to just `5`. This causes the call's inferred return type to be `Integer`, so the compiler-inserted cast succeeds. As this example shows, problems with casts are not always obvious. In this paper we aim to uncover the many different ways in which developers use casts by manually analyzing a large sample of cast usages in open-source software.

1.3 Outline

In Section 2 we introduce the methodology we used to analyze casts and to identify cast-usage patterns. Sections 3 and 4 present the cast-usage patterns and answer our research questions. Section 5 discusses related work and Section 6 concludes.

2 FINDING CAST-USAGE PATTERNS

To answer our research questions, we need a corpus of representative “real-world” code and we need to perform source-code analysis to identify cast operations and to help classify these operations.

2.1 Corpus Analysis

We gathered cast-usage data 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. QL extracts the source code of a project 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 used the lgtm.com service provided by Semmle, the developers of QL.

The *lgtm* project database includes—at the time of writing—7,559 Java projects imported from open-source projects hosted in *GitHub*. The *lgtm* database was constructed by importing popular open-source projects, *e.g.*, Apache Maven, Neo4j and Hibernate. Additionally it includes projects exported by developers to *lgtm* to query them for bug finding, smell detection, and other analyses. We argue that this project selection provides a wide coverage over realistic Java applications, excluding potentially misleading projects, *e.g.*, student projects.

2.2 Is the Cast Operator Used?

We want to know how common cast usage is across projects (RQ1).⁵ The box plot in Figure 1 shows, for each project, the fraction of non-native non-abstract methods containing at least one cast.

⁵ We collected the data for this section after completing our manual analysis of casts (Section 2.3). Since the *lgtm* database is continuously evolving, we were unable to analyze the exact same set of projects from which we had drawn our sample.

The x-axis represents the fraction (1 means 100% of methods contain casts). The y-axis has no meaning and is just used to randomly spread the data points for outliers. For outliers, each dot represents a project, and its size is given by the number of compilation units (C.U.) in the project. There are projects where none of the methods contain a cast (at $x = 0$), but there are also four small projects where all methods contain a cast (at $x = 1$; e.g., an SSL Ping tool implemented in a single method). The plot shows that for most projects, fewer than a quarter of their methods contain casts. Overall, of the 24,559,050 non-abstract non-native methods in the database, 2,139,582 (8.7%) contained at least one cast. The following sections analyze why there are cast instances (RQ2) and how often the use cases that lead to casts occur (RQ3).

Fig. 1. Projects by fraction of their methods containing casts. Bulk of data summarized by box plot, outliers shown as individual points.

2.3 Methodology

To identify patterns of cast usage, we analyzed all Java projects in the *lgtm* database, 7,559 projects with a total 10,193,435 casts at the time of writing. There are 215 projects in the database for which we could not retrieve the source code. In total, these 215 projects contain 1,162,583 casts. Moreover, there are also 516 projects that do not contain any cast. Therefore the total cast population to be analyzed consists of 9,030,852 casts in 6,840 projects.

Because the number of cast instances is large, it is not feasible to manually analyze all of them. Therefore we randomly sampled a subset of cast instances to analyze. We wanted to have a high chance to find any pattern that would cover at least 0.1% of casts. The probability of seeing a pattern that covers 0.1% of the cast population when picking a random cast instance is 0.001. Conversely, the probability of missing the pattern is $(1 - 0.001)$. If we analyzed not one, but a sample of 5000 cast instances, this would give us a very low probability of $(1 - 0.001)^{5000} = 0.67\%$ of missing such an infrequent pattern. We settled on this sample size of 5000. Thus the probability of our random sample missing a pattern covering at least 0.1% of cast instances is 0.67%.

Our sample represents a set of casts coming from various projects in the database. We sampled 5000 casts, but there are more than 5000 projects in the database, so not every project is represented in our sample. Figure 2 compares the projects in the database to the projects from which we sampled at least one cast. The x-axis shows the number of casts in a project; the y-axis shows the fraction of projects with fewer than x casts. In the population, 50% of projects have fewer than 100 casts, but in the sample, only 6% of projects have fewer than 100 casts.

Our sample is thus somewhat biased towards larger projects, which is to be expected, given that projects with more casts had a larger probability to be sampled. Remember, we sampled casts, not

Fig. 2. Cast distribution

projects. Nevertheless, the sample does include projects across the entire spectrum, with 50% of projects having fewer than 2,000 casts.

The patterns were arrived at by an iterative process. An initial pilot set of casts (disjoint from the sample) was analyzed by two of the authors, to get a first idea of the kinds of patterns to expect. The full sample of 5,000 casts was then analyzed by the first author, who assigned a pattern to each cast. If no pattern fit the given cast, a new pattern was invented and described. All authors together discussed the patterns and their instances at weekly meetings, refining, merging, or splitting them into new patterns, sometimes reassigning instances to other patterns. This was repeated until consensus among the authors was reached. The analysis of the entire sample took about 10 weeks. Because the pattern assignment was performed primarily by a single author, the particular categorization here is subjective, and the distribution of patterns over the sampled casts may not necessarily generalize to casts in all real-world Java programs.

3 OVERVIEW OF THE SAMPLED CASTS

Our initial sample of 5,000 casts included 526 casts for which the source code was not accessible during our analysis (and thus manual code inspection was impossible). Thus we sampled additional casts until we had exactly 5,000 analyzable casts. As Table 1 shows, we found 1,043 primitive conversions and 3,957 reference casts (upcasts, downcasts, boxing casts, or unboxing casts). The reference casts can be classified as either *guarded* or *unguarded* casts. A guard is a conditional expression on which the cast is control dependent, that, if successful, ensures the cast will not fail. Guards are typically implemented using the `instanceof` operator or using a test of the source value's class (retrieved using the `Object::getClass` method) against a subtype

of the target type. Guards can also be implemented in an application-specific manner, for instance by associating a “type tag” with the source value that can be used to distinguish the run-time type.

Of the 3,957 analyzed reference casts, we found that 1,458 were guarded by a guard in the same method as the cast and 2,499 were either unguarded or had a guard in another method. In the latter case, which we refer to as *possibly unguarded*, determining by manual inspection if a guard is actually present is often infeasible. The possibly unguarded casts are cases where the application developer has some reason for believing the cast will succeed, but it is not immediately apparent in the source code.

As with any expression, casts can appear in either application/library code, in test code, or in generated code. As expected, most casts appear in application or library code (68.54%). However, casts in test and generated code are non-negligible (10.98% and 21.08%, respectively). Casts appearing in code executed by test frameworks, *e.g.*, JUnit, were identified as test code. A compilation unit was classified as generated code either if it contained a comment indicating it was generated or if its source file path indicated it was generated, *e.g.*, by containing the word “generated”. The remaining casts were classified as application or library code.

As we describe in the next section, nearly all guarded casts fit into just a few patterns. Unguarded or possibly unguarded casts account for most of the patterns.

Table 1. Statistics on Sampled Casts

All sampled casts	5,000	100%
Reference casts	3,957	79.14%
Primitive casts	1,043	20.86%
Upcasts	100	2.00%
Downcasts	3,857	77.14%
Boxing casts	11	0.22%
Unboxing casts	18	0.36%
Guarded by instanceof	881	17.62%
Guarded by getClass	64	1.28%
Guarded by type tag	237	4.74%
Unguarded or possibly unguarded	2,499	49.98%
In application/library code	3,427	68.54%
In test code	549	10.98%
In generated code	1,054	21.08%

Table 2. Categorization of Analyzed Cast-Usage Patterns. These are the patterns identified in our manual analysis. The categorization is subjective, thus the distribution of patterns over the analyzed casts is not necessarily generalizable to casts in all Java programs.

	Pattern	Description	# Casts	%
Guarded Group. The cast patterns in this group are guarded casts.				
1	TYPECASE	Cast guarded with <code>instanceof</code> , class literal, or application-specific tag.	1,182	23.64%
2	EQUALS	A cast used in the implementation of the well-known <code>equals</code> method.	247	4.94%
3	PARSERSTACK	A cast to an heterogenous stack.	29	0.58%
API Group. Cast patterns that depends on an API definition.				
4	STASH	A cast to an heterogenous collection element.	559	11.18%
5	FACTORY	A cast used to convert a newly created object.	378	7.56%
6	KNOWNRETURNTYPE	The client of an API knows the exact return type of a method invocation.	89	1.78%
7	DESERIALIZATION	A cast used to convert newly created objects in deserialization.	71	1.42%
8	NEWDYNAMICINSTANCE	Cast the result of <code>newInstance</code> in Class, Constructor, or Array.	59	1.18%
9	COMPOSITE	A composite cast.	21	0.42%
Covariance Group. Patterns related to different kinds of covariance.				
10	FAMILY	A cast applied in a family of classes.	343	6.86%
11	COVARIANTRETURNTYPE	A cast when the return type of a method is covariant.	106	2.12%
12	FLUENTAPI	Cast to permit a fluent API through method chaining.	23	0.46%
Generics Group. Patterns related to use or misuse of generics.				
13	USERAWTYPE	A cast to a raw type (instead of using the generic type).	335	6.70%
14	REMOVEWILDCARD	A cast to remove a wildcard in a generic type.	33	0.66%
15	COVARIANTGENERIC	Remove type parameter to permit covariant generics.	10	0.20%
16	SELECTTYPEARGUMENT	An upcast to guide the type checker to provide the right return type.	9	0.18%
17	GENERICARRAY	A cast to create a generic array.	5	0.10%
18	UNOCCUPIEDTYPEPARAMETER	A cast used to change an unoccupied type parameter in a generic type.	1	0.02%
Type-Hacking Group. Casts due to hacking the type system.				
19	SELECTOVERLOAD	A cast to disambiguate between overloaded methods.	97	1.94%
20	SOLESUBCLASSIMPLEMENTATION	A cast to the only subclass implementation.	57	1.14%
21	IMPLICITINTERSECTIONTYPE	A cast to implicitly use an intersection type.	45	0.90%
22	REFLECTIVEACCESSIBILITY	Cast the result of the <code>Method::invoke</code> or <code>Field::get</code> .	26	0.52%
23	ACCESSSUPERCLASSFIELD	A cast to access a private field in a superclass.	4	0.08%
Code Smell Group. The patterns in this group are regarded as code smells.				
24	REDUNDANT	A cast that is not necessary for compilation.	117	2.34%
25	VARIABLESUPERTYPE	A cast to a variable that could be declared to be more specific.	64	1.28%
26	OBJECTASARRAY	A cast to a constant array slot used as a field of an object.	47	0.94%

4 CAST-USAGE PATTERNS

From our manual inspection, we have identified 26 cast-usage patterns. Table 2 presents our patterns and their occurrences sorted by frequency. We initially sought to describe patterns precisely as QL queries so that detection and categorization was repeatable, but we found this was infeasible because of the complexity of the reasoning involved in identifying a pattern. Often determining the pattern to which a cast belongs requires reasoning about the run-time source of the cast, which might be non-local and might depend on external application frameworks or on generated code. We do not claim that our list of patterns is exhaustive, although our methodology should ensure that any pattern that occurs more than 0.1% of the time has a small probability of being excluded.

We are also interested in the scope of the cast instance, *i.e.*, *does it appear in application/library code, test code, or generated code?* Figure 3 shows our patterns and their occurrences grouped by scope and sorted by frequency.

To organize the presentation, we have grouped the patterns into six categories. The Guarded group contains those patterns where a guard of some kind is used to decide whether to perform the cast. The API group contains patterns that depend on some API definition. Patterns in the Covariance group include casts used to recover a concrete type that is covariant with another type. The Generics group contains patterns related to the use of Java generics, and type erasure in particular. The Type Hacking group contains patterns used to use—or to work around—some feature of the Java type system. Finally, the Code Smell group contains patterns that might be regarded as code smells.

Fig. 3. Cast-Usage Pattern Occurrences

Each pattern is described below using the following template:

- *Description*. Tells what the pattern is about and gives a general overview of its structure.
- *Instances*. Gives one or more concrete examples found in real code. The code snippets presented here were modified for formatting purposes. Each example contains a highlighted line that shows the cast instance being inspected. For each instance presented here, we provide the link to the source-code repository in *lgtm*. Instead of presenting long *lgtm* URLs, we have used the URL shortening service *Bitly* for easier reading. Each *Bitly* link was customized to include the project name. Because projects can be removed from the *lgtm* service after the time of writing, some links may no longer work.
- *Issues*. Discusses the reasons for, flaws of, and alternatives to the pattern that achieve the same goal without casting.

For space reasons, discussion of some patterns is necessarily concise. We attempt to give at least a description of all patterns. More details and further examples can be found in the supplementary material⁶ and in the first author’s Ph.D. thesis [Mastrangelo 2019].

4.1 Guarded Group

4.1.1 TYPECASE.

Description. The TYPECASE pattern consists of dispatching to different cases depending on the run-time type of the source value. The run-time type is tested against known subtypes of the operand type, with each test followed by a cast to that type. The guard may be implemented using one of three variants: an instanceof operator (*GuardByInstanceOf*), a comparison of the run-time class against a class literal (*GuardByClassLiteral*), or an application-specific type tag (*GuardByTypeTag*).

When implementing the pattern, care must be taken with complex operands that the value of the operand is not changed between the guard and the cast, possibly even by another thread. For instance, in some situations the operand expression is a method invocation. The value returned by the method should be the same for both the instanceof and the cast, thus the method should be a pure method. Typically, this problem is avoided by using an effectively final local variable in both the guard and the cast operand.

Instances: 1,182 (23.64%). We found 1,050 in application code, 17 in test code, and 115 in generated code. TYPECASE is by far the most common pattern. Figure 4 shows the different variants of the pattern. The *GuardByInstanceOf* is the most used variant. Often there is just one case and the default case, i.e., when the guard fails, performs a no-op or reports an error. The following listing shows an example of the TYPECASE pattern, using the *GuardByInstanceOf* variant.

Fig. 4. TYPECASE Variant Occurrences

```

1  if (object instanceof Item) {
2      return getStringFromStack(new ItemStack((Item) object));
3  } else if (object instanceof Block) {
4      return getStringFromStack(new ItemStack((Block) object));
5  } else if (object instanceof ItemStack) {
6      return getStringFromStack((ItemStack) object);
7  } else if (object instanceof String) {
8      return (String) object;
9  } else if (object instanceof List) {
10     return getStringFromStack((ItemStack) ((List) object).get(0));
11 } else return "";

```

http://bit.ly/PenguinSquad_Enchiridion_2HnNwB7

The next example shows that TYPECASE can also be used to filter elements by type within a stream. The cast is applied to stream operations (line 1) over the `caseAssignments` collection. The instanceof guard is tested in line 1.

⁶Our data set and the scripts used to analyze the data have been uploaded to Zenodo <https://doi.org/10.5281/zenodo.3369397>.

```

user = (User) caseAssignments.stream().filter(oe -> oe instanceof User).findFirst()
                                   .orElseThrow(() -> new IllegalArgumentException());
http://bit.ly/kielogroup\_jbpm\_2ENCL8a

```

Rather than using an `instanceof` guard, in the following example the target type of the parameter reference is determined by the value of the parameter `referenceType`, which acts as a *type tag* for reference.

```

1 switch (referenceType) {
2   case ReferenceType.FIELD: return fieldSection.getItemIndex((FieldRefKey) reference);
3   case ReferenceType.METHOD: return methodSection.getItemIndex((MethodRefKey) reference);
4   case ReferenceType.STRING: return stringSection.getItemIndex((StringRef) reference);
5   case ReferenceType.TYPE: return typeSection.getItemIndex((TypeRef) reference);
6   case ReferenceType.METHOD_PROTO: return protoSection.getItemIndex((ProtoRefKey) reference);
7   default: throw new ExceptionWithContext("/* [...] */", referenceType);
8 }
http://bit.ly/JesusFreke\_smali\_2Ho8bVL

```

In some cases, the target types of the casts are the same in every branch. In the following snippet, the cast is applied to the `message.obj` field (line 11) according to the value of the `message.what` field (line 1). However, a similar cast is applied in the first branch (line 3). In both branches `message.obj` is of type `Object[]`, but with different lengths. The casts in the calls to `onSuccess` and `onFailure` (lines 5, 13–14) are instances of the **OBJECTASARRAY** pattern.

```

1 switch (message.what) {
2   case SUCCESS_MESSAGE:
3     response = (Object[]) message.obj;
4     if (response != null && response.length >= 3) {
5       onSuccess((Integer) response[0], (Header[]) response[1],
6                (byte[]) response[2]);
7     } else { ... }
8     break;
9   case FAILURE_MESSAGE:
10    response = (Object[]) message.obj;
11    if (response != null && response.length >= 4) {
12      onFailure((Integer) response[0], (Header[]) response[1],
13               (byte[]) response[2], (Throwable) response[3]);
14    } else { ... }
15    break;
16    // ...
17 }
http://bit.ly/loopj\_android\_async\_http\_2IpiULk

```

In the *GuardByClassLiteral* variant, a cast uses an application-specific guard, but the guard depends on a class literal. The next snippet contains several type cases. Each type case is guarded by an equals comparison between a class literal and the `clazz` parameter. The cast is applied to the type parameter `T` only if the guard succeeds.

```

1 @Override
2 @SuppressWarnings("unchecked")
3 public <T> T get(String fieldName, Class<T> clazz) throws DecodingException {
4   // ...
5   if (clazz.equals(ExtensionObject.class))
6     return (T) getExtensionObject(fieldName);
7   // ...
8 }
http://bit.ly/OPCFoundation\_UA\_Java\_Legacy\_2Fb2xmZ

```

Having only a single case—that is, a single guard and cast—is common. In the 742 instances of TYPECASE that used instanceof, 511 (69%) had only one case.

Issues. The TYPECASE pattern can be seen as an *ad hoc* alternative to typecase or pattern matching [Milner 1984] as a language construct. In Kotlin, flow-sensitive typing is used so that immutable values can be used as a subtype when a type guard on the value is successful.⁷ This feature eliminates much of the need for the guarded casts. Pattern matching can be seen in several other languages, e.g., SML, Scala, C#, and Haskell. For instance, in Scala the pattern-matching construct is achieved using the match keyword.

Alternatives to the TYPECASE pattern would be to use the visitor pattern or to use virtual dispatch on the match scrutinee. However, both of these alternatives might be difficult to implement when the scrutinee is defined in a library or in third-party code. There is an ongoing proposal [Goetz 2017a] to add pattern matching to the Java language. The proposal explores changing the instanceof operator in order to support pattern matching. Java 12 extends the switch statement to be used as either a statement or an expression [Bierman 2019; Goetz 2017b]. This enhancement aims to ease the transition to a switch expression that supports pattern matching.

The *GuardByClassLiteral* variant may be used instead of the instanceof operator when the developer wants to match exactly the runtime class of an object. The instanceof operator⁸ returns true if the expression could be cast to the specified type, whereas using a class-literal comparison returns true if the expression is exactly the runtime class.

In some cases, the *GuardByTypeTag* variant can be replaced by *GuardByInstanceOf*. However, if the application-specific tag is a numeric value, the *GuardByTypeTag* could perform better than the *GuardByInstanceOf* using instanceof. Moreover, there are situations where the instanceof operator cannot be used because the cast target types in the different cases are identical.

4.1.2 EQUALS.

Description. This is a common pattern to implement the well-known equals method (declared in java.lang.Object). It is a particular kind of guarded cast. 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 equals⁹ method implementation. This is done to check if the argument has the same type as the receiver (this). Notice that a cast in an equals method is needed because it receives an Object as a parameter.

To exhibit this pattern, a cast must be applied to the parameter of the equals method. The result value of the cast must then be compared against the receiver.

Instances: 247 (4.94%). We found 202 in application code, 0 in test code, and 45 in generated code. This pattern accounts for 16.94% of guarded casts, 247 instances out of 1,458.

The following listing shows an example of the EQUALS pattern. In this case, an instanceof guards for the same type as the receiver.

```

1  @Override
2  public boolean equals(Object obj) {
3      if ( this == obj ) return true;
4      if ( (obj instanceof Difference) ) {
5          Difference that = (Difference) obj;
6          return actualFirst == that.actualFirst && expectedFirst == that.expectedFirst
7              && actualSecond == that.actualSecond && expectedSecond == that.expectedSecond

```

⁷<https://kotlinlang.org/docs/reference/typecasts.html#smart-casts>

⁸<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2>

⁹<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object->

```

8      && key.equals( that.key );
9  }
10     return false;
11 }

```

http://bit.ly/neo4j_neo4j_2vjw94J

In some situations, the type cast is not the same as the enclosing class. Instead, the target type of the cast is a superclass or a super interface of the enclosing class. For example, this happens when the Google AutoValue library¹⁰ is used. AutoValue is a code generator for value classes. The following example shows this scenario.

```

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

```

http://bit.ly/square_sqlbrite_2HmHMYE

Issues. The pattern for an equals method implementation is well known. Most equals methods in our sample are implemented with the same boilerplate structure: first check if the parameter is another reference to this, and then check if the argument is of the right class using either an instanceof test or a getClass comparison. Using getClass requires an additional check that the argument is not null. If all checks succeed, a cast follows, and field-by-field comparisons are performed.

To avoid this boilerplate, other languages bake in deep equality comparisons, at least for some types (e.g., Scala case classes), or provide mechanisms to generate the boilerplate code (e.g., deriving Eq in Haskell or #[derive(Eq)] in Rust). 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 document¹¹ 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.

This pattern can be seen as a special instance of the **TYPECASE** pattern.

4.1.3 *PARSERSTACK*.

Description. The **PARSERSTACK** pattern consists of multiple cases dispatched depending on some application-specific control state with casts of the top elements of a stack-like collection in each case. An application invariant ensures that if the application is in a given state then the top elements of the stack should be of known run-time types.

Instances: 29 (0.58%). We found 13 in application code, 0 in test code, and 16 in generated code. The following example, shows a cast whose value is on top of a stack (line 2). In this case, the code is

¹⁰<https://github.com/google/auto/tree/master/value>

¹¹<http://cr.openjdk.java.net/~briangoetz/amber/datum.html>

transforming a parse tree into an abstract syntax tree. The casts in the switch case are guarded by the parse-tree node type and its arity.

```
1 case JJTASSERT_STMT:
2   exprType msg = arity == 2 ? ((exprType) stack.popNode()) : null;
3   test = (exprType) stack.popNode();
4   return new Assert(test, msg);
```

http://bit.ly/fabioz_Pydev_2HF6nrF

Similar to the previous example, in the following case a guarded cast is performed on a stack of grammar symbols. The code was generated using an LR parser generator. The guard ensures that the parser has already matched a given prefix of the input and so the top of the stack should contain the expected symbols.

```
1 case 40:
2   final Symbol _symbol_n = _symbols[offset + 1];
3   final IdUse n = (IdUse) _symbol_n.value;
4
```

http://bit.ly/Sable_soot_2MZLZ3m

Issues. In our sample, this pattern is always seen when implementing grammar-related operations, such as parsers or interpreters. In some situations, similar to the **STASH** pattern, this pattern could be replaced with a strongly typed heterogeneous collection [Kiselyov et al. 2004].

Similar to **TYPECASE**, multiple cases are evaluated with casts to different types, depending on application-specific guards. However, unlike **TYPECASE**, the success of the casts is ensured not by a type-tag-like value, but by application-specific state (e.g., the current parser state or the state of an evaluator) and proper use of the stack.

4.2 API Group

4.2.1 STASH.

Description. This pattern is used when looking up a value in a heterogeneous container (usually implemented as `Collection<Object>` or as `Map<K, Object>`) used on a compile-time constant identifier, tag, or name. The result of the lookup is then cast to a target type determined by the value of the identifier.

Often the pattern is used to retrieve a value instantiated from a static resource file, e.g., an XML, HTML or a Java properties file. The file contents are (in theory) known at compile time, and the file is included in the binary distribution of the application. These files are often built using tools such as GUI builders.

Instances: 559 (11.18%). We found 354 in application code, 63 in test code, and 142 in generated code. In the example shown below, the return type of the `getAttribute` method is `Object`. The variable context is of type `BasicHttpContext`, which is implemented with `HashMap`.

```
(AuthState) context.getAttribute(TARGET_AUTH_STATE);
```

http://bit.ly/loopj_android_async_http_2SUzY4E

The following example is from an Android application. A cast is applied to the `findViewById` method invocation. View classes are instantiated by the application framework using an XML resource file. The `findViewById` method looks up the view by its ID.

```
(TextView) findViewById(R.id.mobile_network_type);
```

http://bit.ly/pwittchen_NetworkEvents_2HGbrMq

In the following snippet a cast is performed on a `getSerializable` invocation (line 8). This method gets a `Serializable` value given the specified key, `TAG_CUR_DIR` in this case. To set a

value with a specified key, the `putSerializable` method is used. The mentioned cast succeeds because a value of the appropriate type is set in line 15 using the `putSerializable` method.

```

1 private TorrentContentFileTree curDir;
2 @Override
3 public void onCreate(@Nullable Bundle savedInstanceState) {
4     super.onCreate(savedInstanceState);
5     if (activity == null)
6         activity = (AppCompatActivity) getActivity();
7     if (savedInstanceState != null) {
8         curDir = (TorrentContentFileTree) savedInstanceState.getSerializable(TAG_CUR_DIR);
9     } else {
10         makeFileTree();
11     }
12 }
13 @Override
14 public void onSaveInstanceState(Bundle outState) {
15     outState.putSerializable(TAG_CUR_DIR, curDir);
16 }

```

http://bit.ly/proninyaroslav_libretorrent_2TxpZCM

Issues. This pattern suggests a heterogeneous dictionary. In our manual inspection, all dictionary keys and the resulting types are known at compile time, however a cast is needed because the dictionary type does not encode the relationship between key values and the result type. Casts in this pattern are typically not guarded, indicating that the programmer knows the source of the cast based on the value of the key. Some instances of the pattern could be replaced by strongly typed heterogeneous collections [Kiselyov et al. 2004], although implementing these in Java could prove difficult.

For instances of the pattern where the collection is created from a static resource file, code generation could be used to generate the corresponding Java code. For example, the Butter Knife framework¹² for Android uses annotations to avoid the “manual” casting of calls to `findViewById`. Instead, code is generated that casts the result of the call to the appropriate type.

Since this pattern casts a value to a known type from a method invocation, it can be seen as a kind of **KNOWNRETURNTYPE** pattern.

4.2.2 FACTORY.

Description. This pattern is characterized by a cast where the operand is a method call that returns a new object based on the arguments to the call. Usually the arguments that determine the run-time type of the new value are known at compile time, resembling a “type tag” descriptor (cf. **TYPECASE**). *Instances:* 378 (7.56%). We found 144 in application code, 146 in test code, and 88 in generated code. In the example below, the cast instance (line 2) is applied to the result of the `parse` method. The return type of `parse` is of type `Statement`, but, since the statement is a `SELECT` statement, the value returned by the `parse` method is known to be of type `Select` and the cast should succeed.

```

1 statement = "SELECT * FROM mytable WHERE mytable.col = 9 LIMIT :param_name";
2 select = (Select) parserManager.parse(new StringReader(statement));
3

```

http://bit.ly/JSQlParser_JSqParser_2TecMyB

Another example is when using the `openConnection` method of the standard library class `java.net.URL`. The method returns a `URLConnection`, but depending on the URL scheme (e.g.,

¹²<http://jakewharton.github.io/butterknife/>

http), the method will return a particular subclass of `URLConnection` (e.g., `URLConnection`) that must then be downcast to be used.

Issues. In some situations, the use of this pattern can be seen as breaking the method abstraction. This happens because the caller needs to know how the method is implemented in order to determine the run-time return type. In **FACTORY**, there is a known type hierarchy below the return type and the caller casts to a known subtype in that hierarchy based on the arguments passed into the factory method.

The **KNOWNRETURN TYPE** pattern is similar to **FACTORY**, because both depend on the knowledge that a method returns a more specific type. However, in **KNOWNRETURN TYPE** the method always returns a value of the same type, independent of the arguments.

This pattern is prevalent in test code (38.62%). This is because when testing, known parameters are passed to factory methods, and the returned values are then downcast to check specific assertions.

4.2.3 **KNOWNRETURN TYPE.**

Description. There are cases when a method's return type is less specific than the actual returned value's type. This is often to hide implementation details, but may also be because the method overrides another method with a less-specific type and the return type is not changed covariantly.

This pattern is used to cast from the method's return type to the *known* actual return type. This pattern is characterized by a method that always returns a value of the same type, a subtype of the declared return type, regardless of the context or the arguments to the method call.

Instances: 89 (1.78%). We found 61 in application code, 23 in test code, and 5 in generated code. In the following example, a cast is performed to a call to the `getRealization` method (line 1). Its implementation returns a value of type `CubeInstance` (line 5).

```

1 List<CubeSegment> ms = ((CubeInstance) seg.getRealization()).getMergingSegments((CubeSegment) seg);
2 public class CubeSegment implements IBuildable, ISegment, Serializable {
3     private CubeInstance cubeInstance;
4     public IRealization getRealization() {
5         return cubeInstance;
6     }
7 }

```

http://bit.ly/apache_kylin_2SJooO

Issues. This pattern usually indicates an abstraction violation: the caller needs to know the method implementation to know the correct target type.

The **COVARIANTRETURN TYPE** pattern can be considered a special case of this pattern where the return type is known to vary with the receiver type. Like that pattern, associated types [Chakravarty et al. 2005] in languages like Haskell or Rust could be used to avoid the cast.

4.2.4 **DESERIALIZATION.**

Description. This pattern is used to deserialize an object at run time. In its more common form, this pattern is characterized by a cast to the `readObject` method on an `ObjectInputStream` object.

Instances: 71 (1.42%). We found 37 in application code, 12 in test code, and 22 in generated code. The following example shows how the **DESERIALIZATION** pattern is used to create objects from a file system (line 4).

```

1 ObjectInputStream ois = new ObjectInputStream(fis);
2 CrawlURI curi = (CrawlURI)ois.readObject();
3 curi = (CrawlURI)ois.readObject();
4 curi = (CrawlURI)ois.readObject();
5 assertEquals("...", this.seed.toString(), curi.toString());

```

http://bit.ly/internetarchive_heritrix3_2SF4j7k

Issues. The serialization API dates back to Java 1.1 in 1997. Since then, newer serialization APIs have been developed. For instance, Apache Avro¹³ uses generics and class literals to specify the expected type of an object read. In some languages, type-safe serialization and deserialization boilerplate code can be automatically generated. For instance, in Rust the Serde library¹⁴ can generate code to serialize most data types in a variety of formats.

Both this pattern and the **NEWDYNAMICINSTANCE** pattern create objects by using reflection. While it might be considered a special case of **KNOWNRETURNTYPE**, **DESERIALIZATION** differs in that the run-time result type of the `readObject` depends on the state of the input stream and can change depending on context.

4.2.5 NEWDYNAMICINSTANCE.

Description. In the **NEWDYNAMICINSTANCE** pattern, a new object or array is created by means of reflection. The type of the object being created is determined at run time, and the new object is cast to the run-time type or a supertype of the run-time type.

Instances: 59 (1.18%). We found 44 in application code, 5 in test code, and 10 in generated code. The following example shows a cast of the result of the `Class.newInstance()` method.

```
logger = (AuditLogger) Class.forName(className).newInstance(); http://bit.ly/apache\_hadoop\_2HC3IPg
```

Issues. The cast here is needed because of the dynamic nature of reflection. This pattern is usually unguarded, that is, the programmer knows what target type is being created.

Generics could be used to avoid the cast on `newInstance`, assuming the `Class` instance is not a raw type or the wildcard type `Class<?>`. However, the usual API for getting a class instance, `Class.forName`, indeed returns the type `Class<?>`. Because of type erasure, the following two examples compile to the same bytecode:

```
Class<?> c = Class.forName("Main");
Main pf = (Main) c.newInstance();
```

```
Class<Main> c = (Class<Main>) Class.forName("Main");
Main pf = c.newInstance();
```

In the first case, the cast is on the `newInstance` method, an instance of the **NEWDYNAMICINSTANCE** pattern. In the second case, the cast is on the call to `Class.forName`, an instance of the **FACTORY** pattern.

This pattern is related to **DESERIALIZATION**, since both create an object dynamically. It is also related to **REFLECTIVEACCESSIBILITY**, where both retrieve objects by using reflection.

4.2.6 COMPOSITE.

Description. The **COMPOSITE** pattern is characterized by a cast to another element of a composite data structure, typically a tree, where the target type is known because of its position in the data structure.

Instances: 21 (0.42%). All 21 instances were found in application code. The following example shows a cast from a `Box`—as returned by the `getPreviousSibling` method—to a `TableSectionBox`. The programmer reasons that the cast will succeed because the source of the cast is a sibling of another `TableSectionBox`.

```
1 public class TableBox extends BlockBox {
2     protected TableSectionBox sectionAbove(TableSectionBox section) {
3         TableSectionBox prevSection = (TableSectionBox)section.getPreviousSibling();
4     }
```

¹³<https://avro.apache.org/docs/current/>

¹⁴<https://serde.rs/>

```

5  }
6  public abstract class Box implements Styleable {
7      public Box getPreviousSibling() { }
8  }

```

http://bit.ly/flyingsaucerproject_flyingsaucer_2N2nYbY

Issues. The pattern is typical of hierarchical data structures such as abstract syntax trees, document models, or UI layouts. Based on the grammar of the data structure, the types of adjacent objects in the structure can be known. The cast succeeds if the data structure is well-formed. This pattern is only seen in application code, since it is used when designing an extensible API.

More precise typing of the links in the data structure could eliminate the need for the casts. For example, in the above example, the sibling of a `TableSectionBox` might be declared to have type `TableSectionBox`. However, this may require the programmer to override methods to refine return types covariantly. Language features available in other languages like generalized algebraic data types (GADTs) [Peyton Jones et al. 2006] or self types [Bruce 2003; Odersky and Zenger 2005] could also be used to provide a more precise typing.

The pattern can be thought of as a more dynamic variant of the **FAMILY** pattern. Rather than reasoning that the cast will succeed because of the source type's relative position in the class hierarchy, the cast will succeed because of the source value's position in a composite data structure.

4.3 Covariance Group

4.3.1 FAMILY.

Description. The FAMILY pattern implements casts to provide a sort of *family polymorphism* [Ernst 2001]. A “family” consists of multiple mutually-dependent types designed to collaborate with each other. Each type has a role in the family. Deriving from a base family to form another family requires subclassing all the members of the base family, with the subclasses in the new family retaining their respective roles in the new family.

Because method parameter types are invariant in Java, and because covariant parameter types are unsound in general, the method parameter types in the derived family are the same as in the base family. Casts are therefore necessary for one member of a derived family to access another member using its derived family type rather than its base family type.

Instances: 343 (6.86%). We found 256 in application code, 37 in test code, and 50 in generated code. The following example shows an instance of this pattern. In this case, the interfaces `StepInterface`, `StepMetaInterface`, and `StepDataInterface` are part of a base family, and the `stopRunning` method has parameters of these types. In the derived family the roles of these three interfaces are implemented by the classes `DynamicSQLRow`, `DynamicSQLRowMeta`, and `DynamicSQLRowData`. A cast is applied to the parameter `smi` of `stopRunning` in `DynamicSQLRow` (line 8). This cast is necessary to convert the method parameter, of the base family type `StepDataInterface`, into the derived family type with the same role.

```

1  public interface StepInterface extends VariableSpace, HasLogChannelInterface {
2      public void stopRunning(StepMetaInterface smi, StepDataInterface sdi) throws KettleException;
3  }
4  public class DynamicSQLRow extends BaseStep implements StepInterface {
5      private DynamicSQLRowMeta meta;
6      private DynamicSQLRowData data;
7      public void stopRunning(StepMetaInterface smi, StepDataInterface sdi) throws KettleException {
8          meta = (DynamicSQLRowMeta) smi;
9          data = (DynamicSQLRowData) sdi;
10     }
11 }

```

http://bit.ly/pentaho_pentaho_kettle_2FN59J8

Issues. Family polymorphism can be encoded in Java using F-bounded polymorphism [Canning et al. 1989]. Indeed one of the main uses of F-bounded polymorphism in Java is to encode type families [Greenman et al. 2014]. However, the encoding may be considered cumbersome and using casts may enable a simpler encoding, albeit one that is no longer statically type safe.

Various proposals have been made to better support family polymorphism (and the related “expression problem” [Wadler 1998]) in object-oriented languages, including the use of design patterns [Nystrom et al. 2003; Oliveira and Cook 2012; Wang and Oliveira 2016], and type systems [Ernst 2000; Kiselyov et al. 2009; Nystrom et al. 2006; Odersky and Zenger 2005; Oliveira et al. 2016; Zhang et al. 2015; Zhang and Myers 2017] that permit some restricted form of covariant method parameters.

4.3.2 COVARIANTRETURN TYPE.

Description. The COVARIANTRETURN TYPE pattern is used to cast a call to a method that returns an instance of a type that is covariant with the receiver type. Commonly the method returns an instance of the receiver type itself.

Instances: 106 (2.12%). We found 85 in application code, 3 in test code, and 18 in generated code. A common instance of this pattern is for calls to the `clone` method of `java.lang.Object` (70 instances), which returns an object of the same dynamic type as the receiver but whose static type is `Object`. The following snippet shows a cast on the `clone` method.

```
1  @Override
2  public ListTagsForResourceResult clone() {
3      try {
4          return (ListTagsForResourceResult) super.clone();
5      } catch (CloneNotSupportedException e) {
6          throw new IllegalStateException( ... );
7      }
8  }
```

http://bit.ly/aws_aws_sdk_java_2GvHhYt

In the following example, the `initCause` method—from the `java.lang.Throwable` class—has return type `Throwable`. Nevertheless, this method returns the receiver (after setting the cause exception). Therefore a cast is needed to recover the original exception type, as shown in the following example. This use case resembles the FLUENTAPI pattern.

```
throw (IllegalArgumentException) new IllegalArgumentException(...).initCause(e);
```

http://bit.ly/apache_activemq_2EnSivc

Issues. The situation of returning this could be avoided if Java supported self types [Bruce 2003]. More generally, associated types [Chakravarty et al. 2005] can provide a statically typed solution, for instance in the second example above.

4.3.3 FLUENTAPI.

Description. A fluent API is an API that allows the developer to operate repeatedly on the same object using method chaining. This pattern is exhibited when the receiver (the `this` reference) is cast to a type parameter that is itself bounded by the self type.

Instances: 23 (0.46%). We found 18 in application code, 0 in test code, and 5 in generated code. In the following snippet, the receiver (`this`) is cast to a type parameter (`B`) in line 4. This allows subclasses to reuse the methods in the base class without overriding them just to change the return type.

```

1 public class ClockBuilder <B extends ClockBuilder<B>> {
2     public final B alarms(final Alarm... ALARMS) {
3         properties.put("alarmsArray", new SimpleObjectProperty<>(ALARMS));
4         return (B) this;
5     }
6 }

```

http://bit.ly/HanSolo_Medusa_2TyBObH

Issues. In most cases, this pattern is concerned with a particular implementation of fluent APIs where recursive generics are used to mimic self types [Bruce 2003]. Other implementations of fluent APIs simply return this without a cast, but these are less extensible.

4.4 Generics Group

4.4.1 USERAWTYPE.

Description. A cast is in the USERAWTYPE pattern when a *raw type* is used rather than a generic type. Methods of raw types typically return `Object` rather than a more specific type.

Instances: 335 (6.70%). We found 176 in application code, 18 in test code, and 141 in generated code. For example, in the following code, the collection `c` and iterator `it` are declared to be of the raw types `Collection` and `Iterator` rather than as parameterized types. The call to `next` on line 4 must be cast to a more specific type because static type information was lost by the use of raw types.

```

1 Collection c = recipients.getRecipients();
2 assertTrue(c.size() >= 1 && c.size() <= 2);
3 Iterator it = c.iterator();
4 verifyRecipient((RecipientInformation)it.next(), privateKey);
5

```

http://bit.ly/begit_bc_java_2SD2HLm

Issues. Raw types exist in Java to support legacy code. Best practice would be to rewrite the code to use generics, but this is not always feasible or cost effective.

This pattern is prevalent in generated code (42.09% of generated instances). Since these casts will not be seen by a developer, code generators make less effort to avoid them.

Casts among generic types and between raw types and generic types are unchecked at run time, although other casts are typically inserted by the compiler to ensure type safety dynamically. When these inserted casts fail, the reported location of the failure may not match the programmer's expectation. Indeed, this is similar to the problem of *blame* in gradually typed languages [Wadler and Findler 2009].

4.4.2 REMOVEWILDCARD.

Description. A cast is in the REMOVEWILDCARD pattern when a *wildcard type* is used rather than a generic type.

Instances: 33 (0.66%). We found 26 in application code, 7 in test code, and 0 in generated code. In the following example, `unit` is declared as `Unit<?>`, but to actually be able to use it a cast to a concrete type is needed.

```
copy.setUnitOfMeasure( (Unit<Length>) unit );
```

http://bit.ly/eclipse_jetty_project_2WMI0Ld

Issues. Wildcard types are a form of existential type and consequently can limit access to members of a generic type. Casts are used to restore access at a particular type.

Since this pattern is an unchecked cast, the discussion about compiler-inserted casts and blame is similar to the `USERAWTYPE` pattern.

4.4.3 COVARIANTGENERIC.

Description. The COVARIANTGENERIC pattern occurs when a cast is used to use an invariant generic type as if it were covariant. It can be implemented by casting a generic type like `List<S>` to a raw type (`List`), which can then be assigned to a variable of `List<T>`, where `S` is a subtype of `T`.

Instances: 10 (0.20%). We found 8 in application code, 2 in test code, and 0 in generated code. In the following snippet we show an instance of this pattern.

```
1 private final List<VariableExpression> dataProcessorVars = new ArrayList<>();
2 new ArrayExpression(ClassHelper.OBJECT_TYPE, (List) dataProcessorVars);
3 public class ArrayExpression extends Expression {
4     public ArrayExpression(ClassNode elementType, List<Expression> exprs) {}
5 }
```

http://bit.ly/spockframework_spock_2UYEsF5

Issues. Altidor et al. [2011] define a type system that adds definition-site variance to Java. This could reduce the need for this pattern, although not in the instance above since `List` is invariant. Scala addresses this issue by taking advantage of definition-site variance in the collections library, for instance by providing a covariant immutable list type.

4.4.4 SELECTTYPEARGUMENT.

Description. This pattern is used to prevent the compiler from inferring a collection element type that is too precise. It guides the type checker to provide the right return type of a generic method.

Instances: 9 (0.18%). We found 4 in application code, 5 in test code, and 0 in generated code. In the following snippet, an upcast is performed to ensure that the inferred type of the call to `singletonList` (line 3) is `List<Framedata>` rather than `List<FrameBuilder>`. Because `List<FrameBuilder>` is not a subtype of `List<Framedata>`, a compilation error would occur if the cast were omitted.

```
1 public List<Framedata> createFrames( String text, boolean mask ) {
2     FrameBuilder curframe = new FramedataImpl1();
3     return Collections.singletonList( (Framedata) curframe );
4 }
5 public interface FrameBuilder extends Framedata { }
```

http://bit.ly/arpruss_rasperryjammod_2USL7Ai

Issues. In some cases, instead of casting, this pattern could be avoided using explicit type arguments, e.g., `Collections.<Framedata>singletonList(curframe)`. With Java 8 this cast became unnecessary due to better type inference.¹⁵

4.4.5 GENERICARRAY.

Description. A cast due to the instantiation of an array with a parameterized base type. In Java these arrays cannot be instantiated; instead an `Object[]` or an array of raw types must be created. The cast is necessary to use the array at the intended type.

Instances: 5 (0.10%). We found 5 in application code, 0 in test code, and 0 in generated code. In the following snippet, a cast is required when accessing an element in the array (line 5). The array is created using the raw type `List[][]` and assigned to a variable of the wildcard type `List<?>[][]` (line 1). It is not possible to simply allocate a `List<byte[]>`.

```
1 List<byte[]> build(int tx, int ty, ByteOrder order, boolean cCompatibility);
2 List<?>[][] partialResults = new List[th][tw];
3 for (...) {
4     partialResults[ty][tx] = build(tx, ty, order, cCompatibility);
5     // ...
}
```

¹⁵<https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html#jls-18.5>


```

6  layers.addAll((List<byte[]>) partialResults[y][x]);
7  }

```

http://bit.ly/ppiastucki_recast4j_2EM7zWK

Instead of casting individual elements, the following example shows a cast applied directly when the array is created.

```

T[] newArray = (T[]) new Object[growSize(currentSize)];

```

http://bit.ly/seven332_Nimingban_2UdBwIL

Issues. This pattern occurs because generic type parameters are not reified at run time, but array types are reified. To create a generic `T[]`, for instance, since the parameter `T` is not known statically, the compiler cannot know the run-time representation of the array. The Java specification just forbids these problematic cases and therefore requires programmers to create arrays of raw types and to use casts.

4.4.6 UNOCCUPIEDTYPEPARAMETER.

Description. This pattern occurs when a cast is used to change the type parameter of a generic type. Moreover, the cast is safe because the type parameter holds no values.

Instances: 1 (0.02%). This instance was found in application code. This cast is used to implement an `Either` type. A value of type `Either<L, R>` can be either a value of type `L` or of type `R`. In this instance, the receiver—of type `Either<L, R>`—is cast to `Either<U, R>` (line 8). There is no subtype relation between `L` and `U`. However, the cast succeeds because the programmer ensures (using the guard `isLeft` in line 6) that no value of type `U` is accessible from this. Note that this cast does not conform to the `TYPECASE` pattern, despite the guard, because the target type is not a subtype of the cast operand. The cast succeeds only because of Java's type erasure implementation.

```

1  public interface Either<L, R> extends Value<R>, Serializable {
2      @SuppressWarnings("unchecked")
3      default <U> Either<U, R> mapLeft(Function<? super L, ? extends U> leftMapper) {
4          Objects.requireNonNull(leftMapper, "leftMapper is null");
5          if (isLeft()) {
6              return Either.left(leftMapper.apply(getLeft()));
7          } else {
8              return (Either<U, R>) this;
9          }
10     }
11 }

```

http://bit.ly/vavr_io_vavr_2SMIfI2

Issues. This pattern also occurs with empty collections. For instance, the Java standard library implementation of the method `Collections.<T>emptyList` casts a private constant with raw type `List` to a `List<T>`. This is safe because the list is empty and has no elements of type `T`.

Scala has an unoccupied `Nothing` type to handle this situation (at least with covariant collections). For instance, an (immutable) empty list has `List[Nothing]`, which is a subtype of `List[T]` for any type `T`.

4.5 Type-Hacking Group

4.5.1 SELECTOVERLOAD.

Description. This pattern is used to select the appropriate version of an overloaded method.

A cast on `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, a cast is needed to select the appropriate implementation. This is because the type of `null` has the special type

`null`¹⁶ which can be treated as any reference type. In this case, the compiler cannot determine which method implementation to select.

Instances: 97 (1.94%). We found 51 in application code, 45 in test code, and 1 in generated code. In the following example, `actual.data()` returns a boxed `Long`. Because implicit upcasts have precedence over implicit unboxing conversions, the call is needed to invoke the method that takes a `long` (line 3) rather than the method that takes an `Object` (line 2).

```
1 assertEquals(expected, (long) actual.data());
2 void assertEquals(Object expected, Object actual) { ... }
3 void assertEquals(long expected, long actual) { ... } http://bit.ly/spullara\_redis\_protocol\_2FC9Llb
```

Issues. Oostvogels et al. [2018] propose an extension to TypeScript to express constraints between properties, which can then be mapped onto optional parameters.

Both the `ACCESSSUPERCLASSFIELD` and this pattern are used to select class members. While this pattern is used to select the appropriate overloaded method, the `ACCESSSUPERCLASSFIELD` is used to select a field in a superclass.

4.5.2 SOLESUBCLASSIMPLEMENTATION.

Description. The `SOLESUBCLASSIMPLEMENTATION` occurs when an interface or abstract class has only one implementing subclass. Casting the interface to this class must succeed because it cannot possibly be an instance of another class.

Instances: 57 (1.14%). We found 28 in application code, 6 in test code, and 23 in generated code. In the following example the `jobId` variable is cast to the sole implementation (`JobIdImpl`).

```
return Longs.compare(id, ((JobIdImpl) jobId).id); http://bit.ly/ow2\_proactive\_scheduling\_2Ulcjfs
```

Issues. This pattern occurs when there is high coupling between super and subclass. In some cases, the cast instance appears in a generated class. This mechanism allows the developer to extend this class to add custom code. Therefore, this high coupling is acceptable. The developer assumes that there is no other implementation of the base class, otherwise the cast instance fails.

4.5.3 IMPLICITINTERSECTIONTYPE.

Description. This pattern occurs when there is a downcast of reference v of type T to a target interface type I . Although T does not implement I , the cast succeeds because all possible run-time types of v do implement I .

Instances: 45 (0.90%). We found 19 in application code, 21 in test code, and 5 in generated code. For instance, in the following example the method call returns a `Number`, which does not implement `Comparable`; however, all values that could be returned by the method are subclasses of `Number` in `java.lang`, which each implement `Comparable`.

```
Comparable max = (Comparable) properties.getMaxValue(); http://bit.ly/senbox\_org\_snap\_desktop\_2FQOt4v
```

Issues. Fourtounis et al. [2018] propose a static analysis of dynamic proxies, which are a special case of this pattern. Their analysis is implemented using Doop [Bravenboer and Smaragdakis 2009].

¹⁶<https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.1>

4.5.4 REFLECTIVEACCESSIBILITY.

Description. This pattern accesses a field of an object by means of reflection. Typically reflection is used because the field is private and therefore inaccessible at compile time and the developer cannot change the field declaration itself. In this case, the method `Field::setAccessible(true)` is invoked on the field before getting the value of the field. The cast is needed because `Field::get` returns an `Object`.

Instances: 26 (0.52%). We found 21 in application code, 5 in test code, and 0 in generated code. The following snippet shows how this pattern is used:

```
HttpEntity wrapped = (HttpEntity) f.get(entity); http://bit.ly/loopj\_android\_async\_http\_2SOISRr
```

Issues. Using reflection to access a field is a common workaround to tight access-control restrictions. As with **DESERIALIZATION**, this pattern is necessary because a library method can return values of many different types at run time, and so is declared to return `Object`.

4.5.5 ACCESSSUPERCLASSFIELD.

Description. Perform an upcast to access a field of a superclass of the cast operand.

Instances: 4 (0.08%). All 4 instances were found in generated code. The following snippet shows an instance of this pattern.

The instance below has a method whose parameter is a subclass of the current class. The cast is needed to access a private field of the current class. Being an upcast, the cast is always safe. However, the base class is generated code; possibly, a manually written version would just combine the two classes.

```
1 public abstract class StudentsPerformanceReport_Base extends QueueJobWithFile {
2     public ExecutionSemester getValue(StudentsPerformanceReport o1) {
3         return ((StudentsPerformanceReport_Base)o1).executionSemester.get();
4     }
5     private OwnedVBox<ExecutionSemester> executionSemester;
6 }
7 public class StudentsPerformanceReport extends StudentsPerformanceReport_Base {
8 }
```

http://bit.ly/FenixEdu_fenixedu_academic_2SQxlkC

Issues. Another use of the pattern, although not observed in our sample, is to upcast a value to access a field of a superclass that is shadowed by another field of the same name in the subclass.

The **REFLECTIVEACCESSIBILITY** pattern is also used to access private or protected fields, albeit fields of unrelated classes that cannot be accessed simply by casting to another type. Like **SOLE-SUBCLASSIMPLEMENTATION**, this pattern occurs when there is high coupling between super and subclass.

Both **SELECTOVERLOAD** and this pattern are used to select class members. While this pattern is used to select a field in a superclass, the **SELECTOVERLOAD** is used to select the appropriate overloaded method.

4.6 Code Smell Group

4.6.1 REDUNDANT.

Description. A redundant cast is a cast that is not necessary for compilation. The cast could be removed from source code without affecting the application.

To detect the **REDUNDANT** pattern, the expression being cast needs to be of the same type as the type being cast to.

Instances: 117 (2.34%). We found 64 in application code, 12 in test code, and 41 in generated code. The following cast instance is trivially redundant: both the target type and the static type of the operand `count(b)` are `BigDecimal`.

```
BigDecimal count = (BigDecimal) count(b);
```

http://bit.ly/sigmoidanalytics_spork_2SIqWYq

Issues. Redundant casts are generally upcasts. This pattern arises often in generated code. It may also appear due to code refactorings that change a type and therefore make the cast redundant, or the cast may be useful for improving code clarity.

4.6.2 VARIABLE SUPERTYPE.

Description. This pattern occurs when a cast is applied to a variable (local variable, parameter, or field), that has usually been assigned just once and is declared with a proper supertype of the value assigned into it. The type of the value being assigned to can be determined locally either within the enclosing method or class.

Instances: 64 (1.28%). We found 53 in application code, 8 in test code, and 3 in generated code. The following snippet shows an example of this pattern in line 4. The `samlTokenRenewer` variable is being cast to the `SAMLTokenRenewer` class. The variable is declared with type `TokenRenewer` (super-class of `SAMLTokenRenewer`) in line 1. However, the variable is being initialized with the expression `new SAMLTokenRenewer()`. Thus, the cast instance could be trivially avoided by changing the declaration of the `samlTokenRenewer` variable to `SAMLTokenRenewer` instead of `TokenRenewer`.

```
1 TokenRenewer samlTokenRenewer = new SAMLTokenRenewer();
2 samlTokenRenewer.setVerifyProofOfPossession(false);
3 samlTokenRenewer.setAllowRenewalAfterExpiry(true);
4 ((SAMLTokenRenewer) samlTokenRenewer).setMaxExpiry(1L);
5
```

http://bit.ly/apache_cxf_2SNoUXj

In some cases, the variable only has a particular type under certain conditions. For instance, in the snippet below, the parameter `k1` is cast to the `Comparable` class (line 6). `k1` is declared as `E` (line 4), an unbounded type parameter (line 1). The developer likely designed the class so that `E` must be `Comparable` when `comparator` is `null`, providing an API with two ways to compare list elements.

```
1 public class SortedArrayList<E> extends ArrayList<E> {
2     protected final Comparator<E> comparator;
3     @SuppressWarnings( {"unchecked"})
4     protected int compare(final E k1, final E k2) {
5         if (comparator == null) {
6             return ((Comparable) k1).compareTo(k2);
7         }
8         return comparator.compare(k1, k2);
9     }
10 }
```

http://bit.ly/oblac_jodd_2UKxm6H

Issues. In many 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, the cast could be eliminated. However, there may be cases when declaring the variable with the supertype may make the code more readable.

The cast could also be avoided using type inference of variables. The variable type would be inferred to be the more specific type. Java 10 supports type inference, however only for local variables and only when there is an initializer expression [Goetz 2018].

This pattern is sometimes related to the **REDUNDANT** pattern. Although the cast is not redundant in **VARIABLESUPERTYPE**, by only changing the declaration of the variable to a more specific type, in many instances of the pattern, the cast does become redundant.

4.6.3 OBJECTASARRAY.

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

Instances: 47 (0.94%). We found 36 in application code, 10 in test code, and 1 in generated code. The following example shows an instance of this pattern. The variable `currentState` contains an `Object[]` with a fixed schema.¹⁷ Each element of the array contains values of different types. A cast is performed on a constant array slot: `(BitSet) currentState[3]` on line 5.

```

1 BitSet theLoadedFields = (BitSet)currentState[2];
2 for (int i = 0; i < this.loadedFields.length; i++) {
3     this.loadedFields[i] = theLoadedFields.get(i);
4 }
5 BitSet theModifiedFields = (BitSet)currentState[3];
6 for (int i = 0; i < dirtyFields.length; i++) {
7     dirtyFields[i] = theModifiedFields.get(i);
8 }
9 setVersion(currentState[1]);

```

http://bit.ly/datanucleus_datanucleus_core_2S1L5Zf

Issues. This pattern usually suggests an abuse of the type system. Using an object with statically typed fields might be a better alternative. The pattern may be useful when the value needs to be extensible or if the fields of the array need to be iterated over.

5 RELATED WORK

5.1 Cast Analysis

Prior work investigated various aspects of casts. **Winther [2011]** classified casts based on their safety (he found that roughly one quarter of casts were guarded, and our results corroborate that finding) and presented a path-sensitive analysis that allows a developer to avoid casting once a guarded instance of is provided. **Tsantalis et al. [2008]** demonstrated a tool for detecting casts similar to our **TYPECASE** pattern, and for refactoring them into uses of polymorphism and the State/Strategy design patterns. **Livshits [2006]** and **Livshits et al. [2005]** described how to construct call graphs in the presence of reflection. They identified common reflection-usage patterns, and most of these patterns use casts. **Landman et al. [2017]** 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 identified reflection AST patterns, which often involve the use of casts. **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.

5.2 Code Corpora

A number of prior publications selected and curated corpora of projects, for performance benchmarking [**Blackburn et al. 2006**], for static analysis [**Tempero et al. 2010**], for dynamic analysis [**Dietrich et al. 2017b**], and for repository mining in general [**Allamanis and Sutton 2013**]. **Lopes et al. [2017]** conducted a study to measure code duplication in *GitHub*. They found out that much of the code there is actually duplicated. We believe that the lgtm project database of 7,559 Java projects imported from *GitHub* represents a meaningful corpus for the study we present in this paper.

¹⁷<http://www.datanucleus.org/javadocs/core/5.0/org/datanucleus/enhancement/Detachable.html>

5.3 Tools for Mining Software Repositories

There exist a number of query languages that can be used to analyze large-scale code bases. [Urma and Mycroft \[2012\]](#) evaluate seven such languages: *Browse-By-Query*¹⁸, *Jackpot*¹⁹, *PMD*²⁰, *Java Tools Language* [[Cohen et al. 2006](#)], *SOUL* [[De Roover et al. 2011](#)], *JQuery* [[Volder 2006](#)], and *.QL* [[de Moor et al. 2007](#)]. They implemented—whenever possible—four use cases using the languages mentioned above, and they found that only *SOUL* and *.QL* have the minimal features needed to implement all their use cases.

[Dyer et al. \[2013a,b\]](#) built *Boa*, a domain-specific language 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](#)].

Similar to *Boa*, *lgtm* is a platform to query software-project properties. It works by querying repositories from *GitHub*. But it does not work at a large scale, i.e., *lgtm* web interface allows the user to query just a few projects. Unlike *Boa*, *lgtm* is based on *QL*—previously named *.QL*—an object-oriented domain-specific language to query recursive data structures [[Avgustinov et al. 2016](#)]. In our work, we used *QL* to query the entire *lgtm* database by working with *lgtm* developers to run the query directly on their servers.

[Bajracharya et al. \[2009\]](#) provide a tool to query large code bases by extracting the source code into a relational model. *Sourcegraph*²¹ is a tool that allows regular expression and diff searches. It integrates with source repositories to easily navigate software projects. Trying to unify analysis and transformation tools, [Vinju and Cordy \[2006\]](#) and [Klint et al. \[2009\]](#) built *Rascal*, a DSL that aims to bring them together by querying the AST of a program.

We initially used *QL* to find cast-usage patterns. However, we found that cast usages that looked different on the surface often belonged to the same meaningful pattern. Thus we moved to a manual analysis approach, which allowed us to better classify the various instances. Of course this came at the cost of only studying a sample of the casts. We countered this by using a sample size that keeps a low probability of missing patterns with a given minimum number of instances.

5.4 Empirical Studies of Large Code Bases

There is a considerable body of research studying the use of programming-language features by analyzing large code bases. [Callaú et al. \[2013\]](#) performed an empirical study to assess how much the dynamic and reflective features of *Smalltalk* are actually used in practice. [Mastrangelo et al. \[2015\]](#) studied how Java’s *sun.misc.Unsafe* library is used in libraries and applications. Analogously, [Richards et al. \[2010\]](#), [Richards et al. \[2011\]](#), and [Wei et al. \[2016\]](#) conducted a similar study, but in this case targeting JavaScript’s dynamic behavior and in particular the *eval* function. Also for JavaScript, [Madsen and Andreasen \[2014\]](#) analyzed how fields are accessed via strings, while [Jang et al. \[2010\]](#) analyzed privacy violations. Similar empirical studies were done for PHP [[Dahse and Holz 2015](#); [Doyle and Walden 2011](#); [Hills et al. 2013](#)] and Swift [[Rebouças et al. 2016](#)]. [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.

¹⁸<http://browsebyquery.sourceforge.net>

¹⁹<http://wiki.netbeans.org/Jackpot>

²⁰<https://pmd.github.io>

²¹<https://sourcegraph.com>

Dietrich et al. [2017a] conducted a study about how programmers use contracts in *Maven Central*. Dietrich et al. [2014] studied how API changes impact Java programs in a study of the Qualitas Corpus. Mazinianian et al. [2017] and Uesbeck et al. [2016] studied how developers use lambdas in Java and C++ respectively. Costa et al. [2017] have mined *GitHub* corpora to study the use and performance of Java collections, and how these usages can be improved. They found that in most cases there is an alternative usage that improves performance. 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*.

Related to studies of language-feature usage, some studies focused on code smells. 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. Palomba et al. [2013] propose to detect code smells using change-history information.

6 CONCLUSIONS

Many of the patterns we found should be unsurprising to most object-oriented programmers. That nearly 50% of casts in our sample are (possibly) unguarded suggests that developers use application-specific knowledge that cannot be easily encoded in the type system to ensure the absence of run-time type errors.

Our study provides insight on the boundary between static and dynamic typing, which may inform research on both static and dynamic, as well as gradual [Siek and Taha 2006], type systems. Conversely, this research can inform the design of extensions of the Java type system to reduce the need for casting. Many programming languages provide features to ameliorate the more common use cases of casts. For instance, Kotlin's smart casts couple together `instanceof` and cast operations, providing direct support for the `TYPECASE` pattern. More generally, ML-style pattern matching subsumes this pattern. Other language features that might at least partially obviate the need for some of the patterns are intersection types (cf. `IMPLICITINTERSECTIONTYPE`), and self types or associated types (cf. `FACTORY`, `KNOWNRETURNTYPE`, `DESERIALIZATION`, `COVARIANTRETURNTYPE`, `FLUENTAPI`). Virtual classes [Ernst 2000; Odersky and Zenger 2005] and languages that support family polymorphism [Ernst 2001] would help with casts in the `FAMILY` pattern.

Many cast patterns (e.g., `REMOVEWILDCARD`, `GENERICARRAY`, `COVARIANTGENERIC`, `UNOCCUPIED-TYPEPARAMETER`) are used either to work around—or to take advantage of—the erasure of generic type parameters in Java. Reified generics would reduce the need for these patterns.

Our study also suggests analyses could be performed to improve code quality and eliminate some cast usages, for instance removing redundant casts, finding opportunities to use generics instead, or locating code smells (cf. `USERAWTYPE`, `KNOWNRETURNTYPE`, `VARIABLESUPERTYPE`). We are currently working to define static analyses to detect some of these patterns automatically. With these analyses, tools can be developed to identify instances of the pattern and to ensure that they are being implemented properly.

ACKNOWLEDGMENTS

This paper is based upon work supported by the Swiss National Science Foundation under Grant CRSII2_136225 / 2. The authors thank Max Schäfer at Semmler for helping us with QL and for running queries on their code database.

REFERENCES

- Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, San Francisco, CA, USA, 207–216. <https://doi.org/10.1109/MSR.2013.6624029>

- John Altidor, Shan Shan Huang, and Yannis Smaragdakis. 2011. Taming the Wildcards: Combining Definition- and Use-Site Variance. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 602–613. <https://doi.org/10.1145/1993498.1993569>
- Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-Oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. <https://doi.org/10.4230/LIPIcs.ECOOP.2016.2>
- S. Bajracharya, J. Ossher, and Cristina Lopes. 2009. Sourcerer: An Internet-Scale Software Repository. In *Tools and Evaluation 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure*. 1–4. <https://doi.org/10.1109/SUITE.2009.5070010>
- Gavin Bierman. 2019. JEP 354: Switch Expressions. (2019). <https://openjdk.java.net/jeps/354>
- 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. 2006. 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)*. ACM, New York, NY, USA, 169–190. <https://doi.org/10.1145/1167473.1167488>
- Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. (2009), 11.
- Kim B. Bruce. 2003. Some Challenging Typing Issues in Object-Oriented Languages. *Electronic Notes in Theoretical Computer Science* 82, 8 (Oct. 2003), 1–29. [https://doi.org/10.1016/S1571-0661\(04\)80799-0](https://doi.org/10.1016/S1571-0661(04)80799-0)
- Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. *Empirical Software Engineering* 18, 6 (Dec. 2013), 1156–1194. <https://doi.org/10.1007/s10664-012-9203-2>
- Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. 273–280. <https://doi.org/10.1145/99370.99392>
- Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 241–253. <https://doi.org/10.1145/1086365.1086397>
- Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. 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)*. IEEE Press, Piscataway, NJ, USA, 134–145.
- Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. 2006. JTL: The Java Tools Language. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 89–108. <https://doi.org/10.1145/1167473.1167481>
- Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 389–400. <https://doi.org/10.1145/3030207.3030221>
- Johannes Dahse and Thorsten Holz. 2015. Experience Report: An Empirical Study of PHP Security Mechanism Usage. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 60–70. <https://doi.org/10.1145/2771783.2771787>
- Oege de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. 2007. Keynote Address: QL for Source Code Analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 3–16. <https://doi.org/10.1109/SCAM.2007.31>
- Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. 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)*. ACM, New York, NY, USA, 71–80. <https://doi.org/10.1145/2093157.2093168>
- J. Dietrich, K. Jezek, and P. Brada. 2014. 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)*. 64–73. <https://doi.org/10.1109/CSMR-WCRE.2014.6747226>
- Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada. 2017a. Contracts in the Wild: A Study of Java Programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:29. <https://doi.org/10.4230/LIPIcs.ECOOP.2017.9>
- Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017b. XCorpus – An Executable Corpus of Java Programs. *The Journal of Object Technology* 16, 4 (2017), 1:1. <https://doi.org/10.5381/jot.2017.16.4.a1>
- M. Doyle and J. Walden. 2011. An Empirical Study of the Evolution of PHP Web Application Security. In *2011 Third International Workshop on Security Measurements and Metrics*. 11–20. <https://doi.org/10.1109/Metrisc.2011.18>

- R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. 2013a. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. 422–431. <https://doi.org/10.1109/ICSE.2013.6606588>
- Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. 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)*. ACM, New York, NY, USA, 779–790. <https://doi.org/10.1145/2568225.2568295>
- Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013b. 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)*. ACM, New York, NY, USA, 23–32. <https://doi.org/10.1145/2517208.2517226>
- Erik Ernst. 2000. gbeta - a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. *DAIMI Report Series* 29, 549 (May 2000). <https://doi.org/10.7146/dpb.v29i549.7654>
- Erik Ernst. 2001. Family Polymorphism. In *ECOOP 2001 — Object-Oriented Programming (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 303–326. https://doi.org/10.1007/3-540-45337-7_17
- George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 209–220. <https://doi.org/10.1145/3213846.3213864>
- Brian Goetz. 2017a. JEP 305: Pattern Matching for instanceof (Preview). (2017). <https://openjdk.java.net/jeps/305>
- Brian Goetz. 2017b. JEP 325: Switch Expressions (Preview). (2017). <https://openjdk.java.net/jeps/325>
- Brian Goetz. 2018. JEP 286: Local-Variable Type Inference. (2018). <https://openjdk.java.net/jeps/286>
- Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking App Behavior Against App Descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1025–1035. <https://doi.org/10.1145/2568225.2568276>
- Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. 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)*. ACM, New York, NY, USA, 11:1–11:10. <https://doi.org/10.1145/1852786.1852801>
- Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded Polymorphism into Shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 89–99. <https://doi.org/10.1145/2594291.2594308>
- Mark Hills, Paul Klint, and Jurgen Vinju. 2013. 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)*. ACM, New York, NY, USA, 325–335. <https://doi.org/10.1145/2483760.2483786>
- Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. 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)*. ACM, New York, NY, USA, 270–283. <https://doi.org/10.1145/1866307.1866339>
- Maria Kechagia and Diomidis Spinellis. 2014. Undocumented and Unchecked: Exceptions That Spell Trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 312–315. <https://doi.org/10.1145/2597073.2597089>
- Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell - Haskell '04*. ACM Press, Snowbird, Utah, USA, 96. <https://doi.org/10.1145/1017472.1017488>
- Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. 2009. Fun with type functions. <https://www.microsoft.com/en-us/research/publication/fun-type-functions/>
- P. Klint, T. v d Storm, and J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 168–177. <https://doi.org/10.1109/SCAM.2009.28>
- D. Landman, A. Serebrenik, and J. J. Vinju. 2017. Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 507–518. <https://doi.org/10.1109/ICSE.2017.53>
- Benjamin Livshits. 2006. *Improving Software Security with Precise Static and Runtime Analysis*. Ph.D. Dissertation. Stanford University, Stanford, California.
- Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 139–160. https://doi.org/10.1007/11575467_11
- Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajjani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 84:1–84:28. <https://doi.org/10.1145/3133908>
- Magnus Madsen and Esben Andreasen. 2014. String Analysis for Dynamic Field Access. In *Compiler Construction*, 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 (Eds.). Vol. 8409. Springer Berlin Heidelberg, Berlin, Heidelberg, 197–217. https://doi.org/10.1007/978-3-642-54807-9_12
- Luis Mastrangelo. 2019. *When and How Java Developers Give Up Static Type Safety*. Ph.D. Dissertation. Università della Svizzera italiana.
- Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. 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)*. ACM, New York, NY, USA, 695–710. <https://doi.org/10.1145/2814270.2814313>
- Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 85:1–85:31. <https://doi.org/10.1145/3133909>
- Robin Milner. 1984. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 184–197. <https://doi.org/10.1145/800055.802035>
- Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. 2015. 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)*. ACM, New York, NY, USA, 404–414. <https://doi.org/10.1145/2786805.2786834>
- Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 138–152. https://doi.org/10.1007/3-540-36579-6_11
- Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: Nested Intersection for Scalable Software Composition. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, 21–36. <https://doi.org/10.1145/1167515.1167476>
- Martin Odersky and Matthias Zenger. 2005. Independently Extensible Solutions to the Expression Problem. In *Proceedings of the Twelfth International Workshop on Foundations of Object-Oriented Languages (FOOL '05)*. <http://homepages.inf.ed.ac.uk/wadler/fool>.
- Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses. In *ECOOP 2012 – Object-Oriented Programming*, 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 James Noble (Eds.). Vol. 7313. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-642-31057-7_2
- Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 364–377. <https://doi.org/10.1145/2951913.2951945>
- Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. 2018. Static Typing of Complex Presence Constraints in Interfaces. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 14:1–14:27. <https://doi.org/10.4230/LIPIcs.ECOOP.2018.14>
- Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 268–278. <https://doi.org/10.1109/ASE.2013.6693086>
- F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2015. Landfill: An Open Dataset of Code Smells with Public Evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 482–485. <https://doi.org/10.1109/MSR.2015.69>
- Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. ACM, New York, NY, USA, 3–12. <https://doi.org/10.1145/1985441.1985446>
- Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and Use of Java Generics. *Empirical Software Engineering* 18, 6 (Dec. 2013), 1047–1089. <https://doi.org/10.1007/s10664-012-9236-6>
- Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM Press, 50–61. <https://www.microsoft.com/en-us/research/publication/simple-unification-based-type-inference-for-gadts/>
- M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor. 2016. An Empirical Study on the Usage of the Swift Programming Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 634–638. <https://doi.org/10.1109/SANER.2016.66>

- Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. 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)*. Springer-Verlag, Berlin, Heidelberg, 52–78.
- Gregor Richards, Sylvain Lebesne, Brian Burg, and Jan Vitek. 2010. 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)*. ACM, New York, NY, USA, 1–12. <https://doi.org/10.1145/1806596.1806598>
- Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and functional programming workshop*. 81–92.
- Andreas Stuchlik and Stefan Hanenberg. 2011. 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)*. ACM, New York, NY, USA, 97–106. <https://doi.org/10.1145/2047849.2047861>
- E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345. <https://doi.org/10.1109/APSEC.2010.46>
- N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. 329–331. <https://doi.org/10.1109/CSMR.2008.4493342>
- Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 403–414.
- Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering* 43, 11 (Nov. 2017), 1063–1088. <https://doi.org/10.1109/TSE.2017.2653105>
- Phillip Merlin Uesbeck, Andreas Stefk, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 760–771. <https://doi.org/10.1145/2884781.2884849>
- Raoul-Gabriel Urma and Alan Mycroft. 2012. 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*. ACM Press, Tucson, Arizona, USA, 35. <https://doi.org/10.1145/2414721.2414728>
- Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. 2007. Declarative Object Identity Using Relation Types. In *ECOOP 2007 - Object-Oriented Programming (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 54–78. https://doi.org/10.1007/978-3-540-73589-2_4
- Jurgen Vinju and James R. Cordy. 2006. How to Make a Bridge between Transformation and Analysis Technologies?. In *Transformation Techniques in Software Engineering (Dagstuhl Seminar Proceedings)*, James R. Cordy, Ralf Lämmel, and Andreas Winter (Eds.). Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.
- Kris De Volder. 2006. JQuery: A Generic Code Browser with a Declarative Configuration Language. In *In Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006*. Springer, 88–102.
- Philip Wadler. 1998. The Expression Problem. <https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>.
- Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, 1–16.
- Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA, 37–41. <https://doi.org/10.1145/2889443.2889448>
- Shiyi Wei, Francesca Xhakaj, and Barbara G. Ryder. 2016. Empirical Study of the Dynamic Behavior of JavaScript Objects. *Software: Practice and Experience* 46, 7 (July 2016), 867–889. <https://doi.org/10.1002/spe.2334>
- Johnni Winther. 2011. Guarded Type Promotion: Eliminating Redundant Casts in Java. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs (FtFJP '11)*. ACM, New York, NY, USA, 6:1–6:8. <https://doi.org/10.1145/2076674.2076680>
- Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. 2015. Lightweight, Flexible Object-oriented Generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 436–445. <https://doi.org/10.1145/2737924.2738008>
- Yizhou Zhang and Andrew C. Myers. 2017. Familia: Unifying Interfaces, Type Classes, and Family Polymorphism. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 70 (Oct. 2017), 31 pages. <https://doi.org/10.1145/3133894>
- Alex Zhitnitsky. 2016. 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/>.