Surprising (to Nate):

1. # of unguarded casts (~50%)
2. dirty tricks with generics
3.

Why Developers Need to Escape the Type System?

A Large Empirical Study on How Casting Operations are Used in JAVA

1st Luis Mastrangelo
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
luis.mastrangelo@usi.ch

2nd Matthias Hauswirth
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
matthias.hauswirth@usi.ch

3rd Nathaniel Nystrom
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
nathaniel.nystrom@usi.ch

Overall question to QL:
given this cast, is it safe? (demonstrate to me that there is no way a value flowing into this cast would not be a subtype of the cast's type)

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

really?

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

and, what's the answer?

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

To study how are they used, and most importantly, why are they used, we have analyzed both source code and compiled bytecode files on multiple JAVA repositories. We have discovered several cast patterns. We hope that our study gives support for more empirical studies to understand how a static type system impacts the development productivity.

long lead in, no take home

***Index Terms*—cast, patterns, mining, Java, GitHub, QL, lgtm**

I. INTRODUCTION

In programming language design, the goal of a type system is to prevent certain kind of errors at runtime. Thus, a type system is formulated as a collections of constraints that gives any expression in a program a well defined type.

One common extension of type systems is *subtyping* — or *subtype polymorphism* — usually seen in *object-oriented* programming languages. The subtype mechanism allows the interoperability of two different, but related types. As Pierce states [1], "[...] S is a subtype of T, written $S <: T$, to mean that any term of type S can safely be used in a context where a term of type T is expected. This view of subtyping is often called the *principle of safe substitution*."

Programming languages with subtyping such as JAVA or C++ provide a mechanism to *view* an expression as a different type as it was defined. This mechanism is often called *casting* and takes the form `(T)t`. Casting can be in two directions: *upcast* and *downcast*.

In a type system with subtyping, ...

An upcast conversion happens when converting from a reference type S to a reference type T, provided that T is a *supertype* of S. An upcast does not require any explicit casting operation nor compiler check. However, as we shall see later on, there are situations where an upcast requires an explicit casting operation. to fulfil its purpose.

On the other hand, a downcast happens when converting from a reference type S to a reference type T, provided that T is a *subtype* of S. Unlike upcasts, downcasts require a run-time check to verify that the conversion is indeed valid. This implies that downcasts provide the means to bypass the type system. By avoiding the type system, downcasts can pose potential threats, because it is like the developer saying to the compiler: *"Trust me here, I know what I'm doing"*. Being a escape-hatch to the type system, a cast is often seen as a design flaw or code smell [2] in an object-oriented system.

the static type system

We believe this escape-hatch provided by casting operation can reveal more than meets the eye. By studying both upcast and downcast conversions, we can find common code patterns. These patterns can provide an insight on how the language is being used by developers in real world applications. We have confidence that this knowledge can be: a) useful to make informed decisions for current & future language designers, not only JAVA, b) a reference for tool builders, *e.g.*, by providing more precise or new refactoring analyses and, c) a guide for developers for best or better practices. Therefore, it is important to understand how and why developers use casts and related operations.

if this is a paper presenting result it needs to be convincing, not state "we have confidence'

For our analysis, we have done an empirical study on a large JAVA repository. We have analyzed 24 JAVA open-source projects looking for casts and related operators, *i.e.*, `instanceof`, to see how and why are they used. We have devised 21 cast patterns that provide the rationale behind them. We provide an automatic approach to detect these patterns. Overall, these patterns categorize 73.96% of all casts instances found.

today, 24 is not large anymore

no manual parts??

The rest of this paper is organized as follows: Section II presents an overview of casts in JAVA. Section III discusses our research questions and introduces our study. Section IV presents an overview of how casts are used in real-world applications. Section V describes our methodology for finding casts usage patterns. Sections VI and VII presents and discusses the patterns we have found. Section VIII gives an overview of the related work, and finally section IX concludes the paper.

Identify applicable funding agency here. If none, delete this.

II. Casts

As defined by the Java Language Specification[1], there are several kinds of conversions: Primitive, Reference, Boxing/Unboxing, Unchecked, Capture, String, and Value set conversions. In this paper, we are interested in *Reference* conversion, *i.e.*, conversion between classes related by the subtyping relation. A cast[2] in Java serves the purpose of convert between two different, but related types.

Listing 1 shows how to use the cast operator (line 2) to treat a reference (the variable `o`) as a different type (`String`) as it was defined (`Object`).

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

Listing 1. Variable `o` (defined as `Object`) is casted to `String`.

A downcast operation can fail at runtime. Whenever it fails, a `ClassCastException` is thrown. Listing 2 shows how to detect whether a cast failed by catching this exception.

```
try {
    Object x = new Integer(0);
    System.out.println((String)x);
} catch (ClassCastException e) {
    System.out.println("");
}
```

Listing 2. Catching `ClassCastException`

often

Sometimes it is not desired to catch an exception to test whether a cast would fail otherwise. Thus, in addition to the cast operator, the `instanceof`[3] operator tests whether an expression can be casted without throwing a `ClassCastException`. Listing 3 shows a usage of the `instanceof` operator together with a cast expression.

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

Listing 3. Runtime type test using `instanceof` before applying a cast.

the reflection API.

Java provides another method to test for an `Object`'s type by means of reflection. The `getClass` method returns the runtime class of the `Object`. Listing 4 shows how to use the `getClass` method to test for an object's type.

```
if (x.getClass() == Foo.class) {
    ((Foo)x).doFoo();
}
```

Listing 4. Runtime type test using `getClass` before applying a cast.

An alternative to the `instanceof` operator is keeping track of the types at the application level, as shown in listing 5.

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

Listing 5. Keep track of the actual types instead of `instanceof`.

The cast operation provides flexibility at expense of the type system. But why developers needs to resort to cast? In the next sections we present a study to determine whether and how casts are used in real-world Java applications.

III. Overview of Our Study

Our study tries to answer the following question: *Why developers need to escape the type system?* The cast operator in Java provides the means to view a reference as a different type as it was defined. Upcasts conversions are done automatically by the compiler. Nevertheless, if some situations a developer is forced to insert upcasts. In the case of downcasts, a check is inserted at run-time to verify that the conversion is sound, thus escaping the type system. *Why is so?* Therefore, we believe we should care about how the casting operations are used in the wild. Specifically, we want to answer the following research questions:

too fluffy

- *RQ*1 : **Is casting used in common application code?** We want to understand to what extent application code actually uses casting operations.
- *RQ*2 : **How and why casts are used?** If casts are actually used in application code, we want to know how and why developers need to escape the type system.
- *RQ*3 : **How recurrent are the reasons for which casts are used?** In addition to understand how and why casts are used, we want to measure how often developers need to resort to certain idioms to solve a particular problem.

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

Source Code Analysis. We have implemented our study using the QL query language: "a declarative, object-oriented logic programming language for querying complex, potentially recursive data structures encoded in a relational data model" [3]. QL allows us to analyze programs at the source code level by abstracting the code sources into a relational data model. Besides providing structural data for programs, *i.e.*, ASTs, QL has the ability to query static types and data-flow analysis. To run our QL queries, we have used the service provided by Semmle.[4]

Projects. As a code base representative of the "real world", we have chosen open-source projects hosted in the most popular source code management repositories, *i.e.*, *GitHub*[5], *GitLab*[6], *Bitbucket*[7]. We have analyzed 24 Java projects hosted in *lgtm*.

how?

large = >1000

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

It is common that a project exhibits more than one pattern. Our list of patterns is not exhaustive. Due to the nature of the cast operator, some casts were uncategorized as they would need a whole program analysis, *e.g.*, including libraries in the analysis.

you can't blame the cast operator for this

[1] https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html

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

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

[4] https://lgtm.com/

[5] https://github.com/

[6] https://gitlab.com/

[7] https://bitbucket.org/

IV. Is the cast operator used?

To answer $RQ1$ we want to know how many cast instances are used in a given project. To this end, we gather the following statistics using QL. We show them here to give an estimation of the size of the code base being analized.

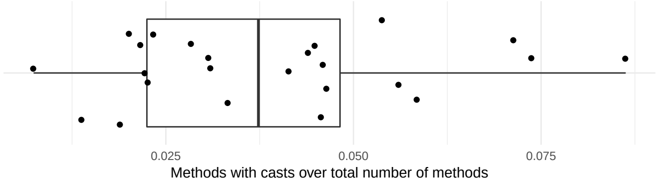
Description	Value
Number of Projects	24
Number of LOC	1,439,913
Number of Methods	121,665
Number of Methods *w/*Cast	6,091
Number of Exprs	4,324,652
Number of Casts	8,627

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

incl. subexpres sions?

For our study, we are interested in both upcasts and downcasts. This is why we want to exclude any primitive conversion in our study. The *Number of Casts* value shown above include only reference conversions. Primitive conversions are always safe (in terms of throwing `ClassCastException`. A primitive conversion happens when both the type of the expression to be casted to and the type to cast to are primitive types. Note that with this definition, we include in our study *boxed* types. Since boxed types are reference types (and therefore not necessarily safe) we want to include them for our analysis.

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



All projects have less than 10% of methods with at least a cast. Overall, around a 3.92% of methods contain at least one cast operation. This means there is a low density of casts. Given the fact that generics were introduced JAVA 5, this can explain this low density. this might deserve a discussion somewhere (parametric polymorphism vs. subtype polymorphism?)

Nevertheless, casts are still used. We want to understand why there are casts instances ($RQ2$) and how often the use cases that leads to casts are used ($RQ3$). The following sections give an answer to these questions.

The query to gather this statistics is available online.[8] The R script to further analyze the query results is available online as well.[9]

V. Finding Casts Usage Patterns

To answer both research questions $RQ2$ and $RQ3$ we have used the QL query language within the *lgtm* service to look for cast instances. As mentioned in section IV, primitive conversions are included as casts. A preliminary step is to exclude them as cast instances. The following QL query shows how to retrieve all relevant cast expressions:

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

Listing 6. QL query to retrieve all relevant cast expressions.

Figure 1 depicts our methodology. We have used this initial result as a starting point for our analysis. Afterwards, we select a random sample for manual inspection. We manually inspected the mentioned casts trying to understand why and how they were used.

By manually looking at several casts instances, we observe that certain characteristics appear often, *e.g.*, a cast in a overridden method, or a cast guarded by an `instaceof`. Using these observations, we implement a QL predicate that detects them and proceed to refine our query with this new predicate. The table of tags is presented in table I. After a new tag is added, the query is run again to iterate over the new results.

Whenever we observe that those tags do not appear randomly, we further inspect the source code to check that is indeed a pattern. We have formalize the structure of each pattern as a QL predicate. Similarly with tags, after a new pattern is added, the query is run again to inspect the casts without pattern. To sum it up, our methodology iterates over the results until no patterns can be detected. The final QL query is available online.[10]

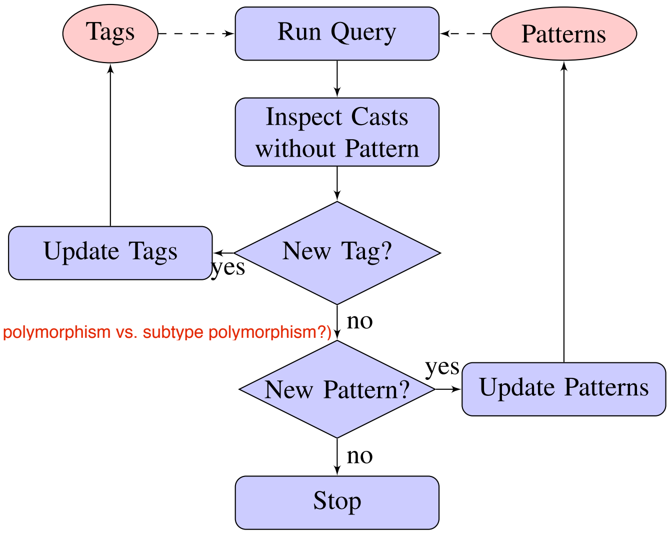


Fig. 1. Process to discover cast tags and patterns

[8] https://gitlab.com/acuarica/java-cast-queries/blob/master/stats.ql

[9] https://gitlab.com/acuarica/java-cast-queries/blob/master/stats.r

[10] https://gitlab.com/acuarica/java-cast-queries/blob/master/casts.ql

TABLE I

CAST TAGS USED TO DISCOVER CAST PATTERNS.

#	Tag	Description	Count
1	*Ref*	A cast applied to a reference value.	0
2	*Null*	A cast applied to the `null` literal.	106
3	*Object*	A cast applied to an expression whose type is `Object`.	541
4	*Subobject*	A cast applied to an expression whose type is a subtype of `Object`.	870
5	*IntoObject*	Cast an expression into `Object`.	21
6	*IntoInterface*	Cast an expression into an interface.	344
7	*ClassFamily*	The expression is a method access, it is not static, and the type is abstract.	177
8	*ToThistype*	The type being casted to is the same as the enclosing class or interface.	66
9	*ToSupertype*	The type being casted to is a supertype of the enclosing class or interface.	76
10	*ToSubtype*	The type being casted to is a subtype of the enclosing class or interface.	3
11	*Var*	The expression of the cast is either a field, a local variable, or a parameter.	0
12	*Field*	The expression of the cast is a field (class or instance variable). Note that this cast inherit from *Var* tag.	92
13	*LocalVar*	The expression of cast is a local variable. Note that this cast inherit from *Var* tag.	185
14	*Param*	The expression of cast is a parameter. Note that this cast inherit from *Var* tag.	363
15	*DefUse*	The expression is a variable which was assigned before in the same method.	150
16	*VarControlByIof*	The expression of the cast is a variable controlled by an `instanceof` operation.	0
17	*VarGuardedByIof*	The expression of the cast is a variable guarded by an `instanceof` operation.	315
18	*Typecase*	This cast is part of a typecase test.	82
19	*VarGuardedByVarIof*	This cast is guarded by a variable whose value is an `instanceof`.	8
20	*VarGuardedByGetClass*	This cast is guarded by a `java.lang.Class.getClass` test.	36
21	*OwnFieldGuardedByIof*	An instance field access guarded by an `instanceof`.	13
22	*GuardedByTag*	A cast guarded by another field in the same object.	5
23	*ArrayGuardedByTag*	The expression of the cast is an array access, whose source definition is a *GuardedByTag* cast.	11
24	*Overload*	A cast whose value is an argument of an overloaded method. ***TODO:*** **Argument/getParent**	256
25	*Varargs*	This cast is an explicit varargs array.	66
26	*MethodAccess*	The expression of the cast is a method access.	303
27	*InOverride*	This cast is in a overriden method.	412
28	*InEquals*	This cast is in an `equals` method.	76
29	*InClone*	This cast is in a `clone` method.	2
30	*InTest*	This cast is in a test method.	360
31	*InGenerated*	This cast is in a generated method.	0
32	*InMaybeGenerated*	This cast may be in a generated method.	4
33	*Clone*	The expression of this cast is an invocation to a `clone` method.	6
34	*RemovedGenericType*	The expression of this cast is a method invocation, but the declaration differs from the usage.	143
35	*NewInstance*	Call site to one of the `newInstances` methods in the `Class`, `Array`, or `Constructor` classes.	17
36	*NewProxyInstance*	Cast to `java.lang.reflect.Proxy::newInstance`.	3
37	*ConstantArgument*	Cast to a method access with only one argument, and that argument is a compile time constant.	226
38	*LookupById*	Cast to an heterogeneous collection.	31
39	*ReflectionFieldGet*	A cast to `java.lang.reflect.Field::get`	7
40	*ReflectionMethodInvoke*	A cast to `java.lang.reflect.Method::invoke`	15
41	*ReadObject*	Cast to `readObject`.	13
42	*FindViewById*	Cast to `findViewById` (seen in Android applications).	89
43	*SerializableExtra*	Cast to `getSerializableExtra`	4
44	*RawIteratorLoop*	A raw access to the `next` method of `java.util.Iterator`.	1
45	*KnowFactoryMethod*	Cast to a — known — factory method. ***TODO:*** **Improve precision for getLogger.**	10
46	*ThrowableGetCause*	Cast to `java.lang.Throwable::getCause`.	10
47	*CovariantReturn*	A covariant return cast.	59
48	*Selfcast*	The expression of the cast and type casted to are the same.	9
49	*Upcast*	An upcast, *i.e.*, the expression of the cast is a supertype of the type casted to.	84
50	*This*	A cast to the `this` reference.	9
51	*IofMethod*	***TODO:*** **Is it used?**	19
52	*IofArray*	***TODO:*** **Is it used?**	0
53	*GetClassIsArrayGuarded*	A cast guarded by `java.lang.Class::isArray`.	18
54	*UnionGuarded*	A cast controlled by more the one `instanceofs`.	0
55	*TypeParameter*	The type casted to is a generice type.	3
56	*OverridingClone*	A cast in a `clone` method implementation.	2
57	*ExceptionForRethrow*	***TODO:*** **Improve precision.**	31
58	*ObjectInCollection*	***TODO:*** **Different semantics.**	18
59	*Redundant*	A redundant cast/`instanceof`.	0
60	*RemoveTypeParameter*	A raw access instead of generic.	140
61	*SearchByType*	A cast in a loop.	101
62	*Stash*	Field stash. ***TODO:*** **Improve semantics.**	55
63	*TypeParameterResolution*	A cast to `Class`.	29
64	*Parameterized*	A parameterized cast.	141
65	*GenericType*	A generic type cast.	3
66	*BoundedType*	A bounded type cast.	0
67	*TypeVariable*	A type variable cast.	90
68	*Wildcard*	A wildcard cast.	0
69	*Unboxed*	Unboxing cast.	44

VI. Casts Usage Patterns

In this section we present the cast usage patterns we found. Figure 2 presents an overview of our patterns and their occurrences sort by most frequent.

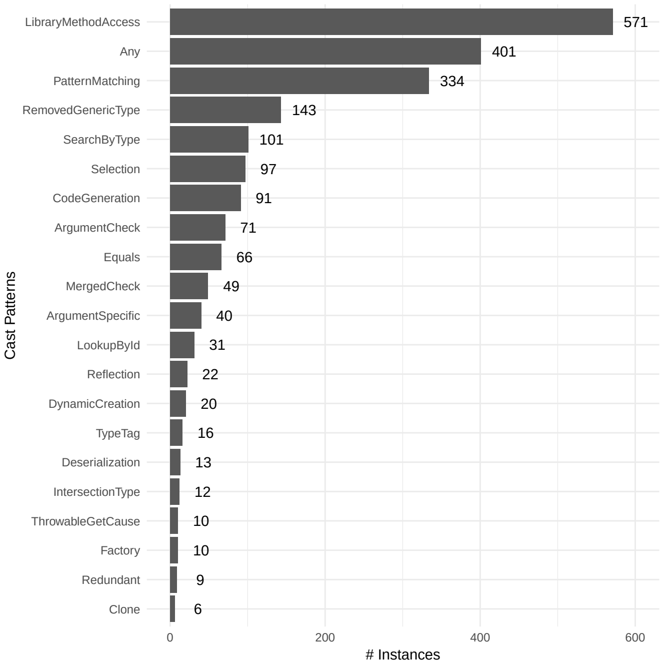


Fig. 2. Cast Patterns Occurrences

Any denotes all cast instances that were not categorized. Each pattern is described using the following template:

- **Description.** Tells what is this pattern about.
- **Instances.** Gives one or more concrete examples found in real code.[11] For each instance presented here, we provide the link to the original source code.
- **Detection.** Describes briefly how this pattern was detected in terms of the tags introduced in the previous section.
- **Discussion.** Presents suggestions, flaws, or comments about the pattern.

A. PatternMatching

Description: This pattern checks whether a parameter in an overridden method has a more specific type.

A cast to a variable guarded by an `instanceof`. A variable is /guarded/ by a condition when the condition controls that access to the variable, and there is no assignment after the condition and before the access to that variable.

Instances: The following listing shows an example of the Typecase pattern.

```java
if (res instanceof Observable) {
    return (Observable) res;
} else if (res instanceof Single) {
    return ((Single) res).toObservable();
} else if (res instanceof Completable) {
    return ((Completable) res).toObservable();
} else {
    return Observable.just(res);
}
```

Listing 7. Instance of the Typecase pattern (from http://bit.ly/2ns4EJq)

This pattern is composed of a guard (`instanceof`) followed by a cast on known subtypes of the static type. Often there is just one case and the default case (*i.e.*, `instanceof` fails) does a *no-op* or reports an error.

Listing shows the detection query for the Typecase pattern. This detection query looks for a cast guarded by an `instanceof`.

Discussion: The Typecase pattern consists of testing the runtime type of a variable against several related types. Based on rule taken from: It was taken from a *lgtm* rule[12].

It is a technique that allows a developer to take different actions according to the runtime type of an object. Depending on the — runtime — type of an object, different cases, usually one for each type will follow.

The Typecase pattern can be seen as an *ad-hoc* alternative to pattern matching. This construct can be seen in several other languages, *e.g.*, Haskell, Scala, and C#. There is an ongoing proposal[13] to add pattern matching to the Java language.

As a workaround, alternatives to the Typecase pattern can be the visitor pattern or polymorphism. But in some cases, the chain of `instanceof` s is of boxed types. Thus no polymorphism can be used.

B. RemoveGenericType

Description: When a generic method is not used as such.

Instances:

Detection:

Discussion: Generics

C. Equals

Description: This pattern is a common pattern to implement the `equals` method.

A cast applied to an expression of type 'java.lang.Object'. A cast applied to an expression whose type is a subclass of 'java.lang.Object'. @cases Cases: Using supertype

In this case, the cast type is the supertype of the owner class.

A cast expression is guarded by either an `instanceof` test or a `getClass` comparison (to the same target type as the cast); in an `equals` [14] method implementation. This is done to check if the argument has same type as the receiver (`this` argument).

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

[11] Please notice that the snippets presented here were slightly modified for formatting purposes.

[12] https://lgtm.com/rules/910065/

[13] http://openjdk.java.net/jeps/305

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

Instances: The following listing[15] shows an example of the EQUALS pattern. In this case, `instanceof` is used to guard for the same type as the receiver.

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

Listing 8. EQUALS pattern using `instanceof` as a guard.

Alternatively, listing 9 [16] shows another example of the EQUALS pattern. But in this case, a `getClass` comparison is used to guard for the same type as the receiver.

```java
@Override
public boolean equals( Object o ) {
    if ( this == o ) return true;
    if ( o == null || getClass() != o.getClass() )
        return false;

    ValuePath that = (ValuePath) o;
    return nodes.equals(that.nodes) &&
        relationships.equals(that.relationships);
}
```

Listing 9. EQUALS pattern guarded by a `getClass` comparison

Detection: The detection query looks for a cast expression inside an `equals` method implementation. Moreover, the cast needs to be guarded by either an `instanceof` test or a `getClass` comparison. The *Object* tag is not declared explicitly in the definition of the pattern. The *InEquals+Param* tags imply that the cast is applied to an expression of type `Object`, as declared in the `equals` method.

Discussion: The pattern for an `equals` method implementation is well-known. We have started looking at all `equals` methods with the following QL query:

```
import java
from EqualsMethod eqm select eqm
```

Listing 10. Fetching all `equals` method implementations.

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

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

There is an exploratory document[17] 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.

D. CODEGENERATION

Description: A cast to a method access to `findViewById` is a pattern in its own.

This is a pattern seen when using the Android platform.

E. CLONE

Description: Cast the result of the `clone`[18] method defined in a super class within a `clone` method implementation.

Instances: Listing shows an example of the CLONE pattern.

Detection: The detection for this pattern looks for a cast within an implementation of a `clone` method. This is shown in listing.

Discussion: We have used the same approach as for the EQUALS pattern. We started looking for all `clone` method implementations using the following QL query:

A common `clone` implementation, however, looks like the following,[19] which it does not include any cast operator.

This pattern suffers the same issues as the EQUALS pattern.

F. SELECTION

Description: This pattern is used to select the appropiate version of an overloaded method[20] where two or more of its implementations differ *only* in some argument type.

A cast to `null` is often used to select against different versions of a method, *i.e.*, to resolve method overloading ambiguity. Whenever a `null` value needs to be an argument of an a cast is needed to select the appropriate implementation. This is because the type of `null` has the special type *null*[21] which can be treated as any reference type. In this case, the compiler cannot determine which method implementation to select.

Instances: Listing 11 [22] shows an example of NULL pattern. Another use case is to select the appropriate the right argument when calling a method with variable arguments.

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

Listing 11. Example of SELECTION pattern.

In this example, there are three versions of the `onSuccess` method, as shown in listing 12. The cast `(String) null` is used to select the appropriate version (line 7), based on the third parameter.

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

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

[16] http://bit.ly/2vKP0MW

[17] http://cr.openjdk.java.net/∼briangoetz/amber/datum.html

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

[19] d

[20] Using ad-hoc polymorphism [5]

[21] https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.1

[22]

```
public void onSuccess(
      int statusCode, Header[] headers, String responseString) { ... }
```

Listing 12. Overloaded methods that differ only in their argument type (the third one).

Detection: Listing 13 shows how to detect this pattern. This pattern shows up when a cast is directly applied to the `null` constant.

```
import java

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

Listing 13. Detection of the NULL pattern.

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

G. DYNAMICCREATION

Description: Dynamically creation of object by means of reflection. These are the casts that can not be avoidable.

The `newInstance` method family declared in the `Class`[23], `Array`[24], [25] and `Constructor`[26] classes creates an object or array by means of reflection.

This pattern consists of casting the result of these methods to the appropriate target type.

Instances: The following example shows a cast from the `Class.newInstance()` method[27].

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

Listing 14. The NEWINSTANCE pattern using the `Class` class.

The following example shows how to dynamically create an array[28].

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

Listing 15. Example of the NEWINSTANCE pattern using the `Array` class.

Whenever a constructor other than the default constructor is needed, the `newInstance` method declared in the `Constructor` class should be used to select the appropriate constructor, as shown in the following example.[29]

```
return (Exception) Class
                      .forName(className)
                      .getConstructor(String.class)
                      .newInstance(message);
```

Listing 16. Example of the NEWINSTANCE pattern using the `Constructor` class.

detection

This detection query looks for casts, where the expression being cast is a call site to methods mentioned above.

```
import java

predicate isByReflection(string qname) {
  qname = "java.lang.reflect.Array" or
  qname = "java.lang.Class<?>" or
  qname = "java.lang.reflect.Constructor<?>"
}

from CastExpr ce, MethodAccess ma, Method m
where ma = ce.getExpr()
  and m = ma.getMethod()
  and m.getName() = "newInstance"
  and isByReflection(m.getDeclaringType().getQualifiedName())
select ce, m.getDeclaringType().getQualifiedName()
```

Listing 17. Fetching all casts to `newInstance()`.

Discussion: The cast here is needed because of the dynamic essence of reflection. This pattern is unguarded, that is, the application programmer knows what is the target type being created.

H. LOOKUPBYID

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

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

Instances: In the example shown in listing [30], the `getAttribute` method returns `Object`. The variable `context` is of type `BasicHttpContext`, which is implemented with `HashMap`.

```
˜AuthState˜ ‘authState‘ =
    (˜AuthState˜)‘context‘.|getAttribute|(˜ClientContext˜.ˆTAR
```

Listing 18. Example of the LOOKUP BY ID pattern.

Detection: The QL query to detect this pattern is shown in listing. Notice that we check for the argument to be static final (constant).

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

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

- *What kind of analysis is needed to detect the source of the cast?*
- *Is worth to have it?*
- *Is better to change API?*

[23] https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#newInstance--

[24] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html#newInstance-java.lang.Class-int-

[25] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html#newInstance-java.lang.Class-int...-

[26] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Constructor.html#newInstance-java.lang.Object...-

[27] d

[28] d

[29] d

[30] d

- *How other — statically typed — languages support this kind of idiom?*
- *Could generative programming a.k.a. templates solve this problem?*

I. TypeTag

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

J. Reflection

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

Instances: The following cast[31] uses this pattern:

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

Listing 19. Using `Field::get` to gain access to a field.

K. Redundant

Description: A redundant cast is a cast or `instanceof` test that always succeed based on the static type.

The expression of the cast and type casted to are the same.

Instances:

Instances: The following example shows a redundant cast.[32] The `instanceof` tests is done in line 8 against the `contextClassLoader` variable. However, notice its definition in line 1 as `URLClassLoader` (which implements `Closeable`. The test will always succeed, being a redundant cast.

```java
1  URLClassLoader contextClassLoader =
2      new URLClassLoader(
3          new URL[]{gradleJar.toURI().toURL()},
4          ClassLoader.getSystemClassLoader().getParent());
5  Thread.currentThread().setContextClassLoader(
6      contextClassLoader);
7  Class<?> mainClass = contextClassLoader.loadClass(
8      "org.gradle.launcher.GradleMain");
9  Method mainMethod = mainClass.getMethod("main",
10     String[].class);
11 mainMethod.invoke(null, new Object[]{args});
12 if (contextClassLoader instanceof Closeable) {
13     ((Closeable) contextClassLoader).close();
14 }
```

Listing 20. Example of Redundant Cast

This detection pattern is already a *lgtm* rule.[33]

Discussion: This is a cast that should always succeed based on the static type. Some of these seem to be because some of the types changed during a refactoring and the cast was not removed. It can be placed maybe for documentation purposes.

L. SearchByType

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

VII. Discussion

***TODO:* Here it goes the discussion.**

Where is the discussion of that other cast study paper? Shouldn't it be the top priority related work and discussed first?

VIII. Related Work

This kind of cast is called *semi guarded* casts [6].

* Relate null as theoretical point of view in the TAPL book.

* Featherweight Java is sound unless you use cast, as a motivation provide a bridge between compile-time and runtime checking.

Notice that and not on bytecode. We examine these kinds of artifacts to analyze how they use casting operations. We use a bytecode analysis library to search for cast operations. & lgtm, since it is quite powerful and efficient. However, there are some issues that limit us to perform a full study using your platform. In particular, the issues that we found are: This could point out ways in which the Java language need to be evolved to provide the same functionality, but in a safer way.

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

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

A. Compilers Writers

Already [7] started to study Fortran programs. By knowing what kind of programs arise in practice, a compiler optimizer can focus in those cases, and therefore can be more effective. Alternatively, to measure the advantages between compilation and interpretation in Basic, [8] has studied a representative dataset of programs. Adding to Knuth's work, [9] made an empirical study for parallelizing compilers. Similar works have been done for Cobol [10], [11], Pascal [12], and APL [13], [14] programs.

[31] d

[32] d

[33] https://lgtm.com/rules/2970081/

B. Benchmarks and Corpuses

Benchmarks are crucial to properly evaluate and measure product development. This is key for both research and industry. One popular benchmark suite for JAVA is DaCapo [15]. This suite has been already cited in more than thousand publications, showing how important is to have reliable benchmark suites.

Another suite was developed by in [16]. They provide a corpus of curated open source systems to facilitate empirical studies on source code.

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

C. Large-scale Codebase Empirical Studies

In the same direction to our plan, [17] perform a study of the dynamic features of SMALLTALK. Analogously, [18], [19] made a similar study, but in this case targeting JAVASCRIPT's dynamic behavior and in particular the `eval` function. Also for JAVASCRIPT, [20] analyzed how fields are accessed via strings, while [21] analyzed privacy violations. Similar empirical studies were done for PHP [22]–[24] and SWIFT [25].

Going one step forward, [26] studied the correlation between programming languages and defects. One important note is that they choose relevant project by popularity, measured *stars* in *GitHub*. We argue that it is more important to analyse projects that are *representative*, not *popular*.

For JAVA, [27] made a study about how programmers use contracts in *Maven Central*. For their analysis[34], they have use JavaParser[35].

[28] have analyzed the relevance of static analysis tools with respect to reflection. They made an empirical study to check how often the reflection API is used in real-world code. They argue, as we do, that controlled experiments on subjects need to be correlated with real-world use cases, *e.g.*, *GitHub* or *Maven Central*. [6] have implemented a flow-sensitive analysis that allows to avoid manually casting once a guarded `instanceof` is provided. [29] have studied how changes in API library impact in JAVA programs. Notice that they have used the Qualitas Corpus [16] mentioned above for their study.

Exceptions: [30], [31] focus on exceptions. They made empirical studies on how programmers handle exceptions in JAVA code. The work done by [32] categorized them in patterns. Whether [33] used a more dynamic approach by analysing stack traces and code issues in *GitHub*.

Collections and Generics: The inclusion of generics in JAVA is closely related to collections. [34], [35] studied how generics were adopted by JAVA developers. They found that the use of generics do not significantly reduce the number of type casts.

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

D. Controlled Experiments on Subjects

There is an extensive literature *per se* in controlled experiments on subjects to understand several aspects in programming, and programming languages. For instance, [37] tried to understand the how expert programmers face problem solving. [38] made a empirical study on how effective is mutation testing. [39] compared how a given — fixed — task was implemented in several programming languages.

[40] realize that, in essence, programmers need to answer reachability questions to understand large codebases.

Several authors [41]–[43] measure whether using a static-type system improves programmers productivity. They compare how a static and a dynamic type system impact on productivity. The common setting for these studies is to have a set of programming problems. Then, let a group of developers solve them in both a static and dynamic languages.

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

The work by [44], [45] goes towards this direction. They have examined programs written by students to understand real debugging conditions. Their focus is on ill-typed programs written in HASKELL. Unfortunately, these dataset does not correspond to real-world code. Our focus is to analyze code by experienced programmers.

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

E. Code Patterns Discovery

[46] have extended ASM [47], [48] to implement symbolic execution and recognize call sites. However, this is only a meta-pattern detector, and not a pattern discovery. [49] used both dynamic and static analysis to discover design patterns, while [50] used only dynamic.

Trying to unify analysis and transformation tools [51], [52] built RASCAL, a DSL that aims to bring them together.

F. Tools for Mining Software Repositories

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

[53], [54] built *Boa*, both a domain-specific language and an online platform[36]. It is used to query software repositories

[34] https://bitbucket.org/jensdietrich/contractstudy

[35] http://javaparser.org/

[36] http://boa.cs.iastate.edu/

on two popular hosting services, *GitHub*[37] and *SourceForge*[38]. The same authors of *Boa* made a study on how new features in JAVA were adopted by developers [55]. This study is based *SourceForge* data. The current problem with *SourceForge* is that is outdated.

To this end, [56] provides an offline mirror of *GitHub* that allows researchers to query any kind of that data. Later on, [57] published the dataset construction process of *GitHub*.

Similar to *Boa*, *lgtm*[39] is a platform to query software projects properties. It works by querying repositories from *GitHub*. But it does not work at a large-scale, *i.e.*, *lgtm* allows the user to query just a few projects. Unlike *Boa*, *lgtm* is based on QL, an object-oriented domain-specific language to query recursive data structures [3].

On top of *Boa*, [58] built *Candoia*[40]. Although it is not a mining software repository *per se*, it eases the creation of mining applications.

Another tool to analyze large software repositories is presented in [59]. In this case, the analysis is dynamic, based on program traces. At the time of this writing, the service[41] was unavailable for testing.

sourcegraph[42] is a tool that allows regular expression and diff searches. It integrates with source repositories.

G. Selecting Good Representatives

Another dimension to consider when analyzing large codebases, is how relevant the repositories are. [60] made a study to measure code duplication in *GitHub*. They found out that much of the code there is actually duplicated. This raises a flag when consider which projects analyze when doing mining software repositories.

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

[62] Visualization API/usage patterns/mining.

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

IX. Conclusions

There is an extensive literature on how reflection is used in JAVA. But casting was never studied in its own.

In this paper we have presented several *cast* patterns. We expect this paper gives a clear understanding of how cast is used in JAVA.

We hope these patterns can aid language and tool designers, researchers, and developers to take advantage of these essential feature.

[37] https://github.com/

[38] https://sourceforge.net/

[39] https://lgtm.com/

[40] http://candoia.github.io/

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

[42] https://sourcegraph.com

References

[1] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 1st ed., 2002.

[2] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, (Piscataway, NJ, USA), pp. 403–414, IEEE Press, 2015.

[3] P. Avgustinov, O. d. Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented Queries on Relational Data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (S. Krishnamurthi and B. S. Lerner, eds.), vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 2:1–2:25, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[4] M. Vaziri, F. Tip, S. Fink, and J. Dolby, "Declarative Object Identity Using Relation Types," in *ECOOP 2007 – Object-Oriented Programming*, Lecture Notes in Computer Science, pp. 54–78, Springer, Berlin, Heidelberg, July 2007.

[5] C. Strachey, "Fundamental Concepts in Programming Languages," *Higher-Order and Symbolic Computation*, vol. 13, pp. 11–49, Apr. 2000.

[6] J. Winther, "Guarded Type Promotion: Eliminating Redundant Casts in Java," in *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, (New York, NY, USA), pp. 6:1–6:8, ACM, 2011.

[7] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience*, vol. 1, pp. 105–133, Apr. 1971.

[8] J. Hammond, "BASIC - an evaluation of processing methods and a study of some programs," *Software: Practice and Experience*, vol. 7, pp. 697–711, Nov. 1977.

[9] Z. Shen, Z. Li, and P. C. Yew, "An empirical study of Fortran programs for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 356–364, July 1990.

[10] A. Salvadori, J. Gordon, and C. Capstick, "Static Profile of COBOL Programs," *SIGPLAN Not.*, vol. 10, pp. 20–33, Aug. 1975.

[11] R. J. Chevance and T. Heidet, "Static Profile and Dynamic Behavior of COBOL Programs," *SIGPLAN Not.*, vol. 13, pp. 44–57, Apr. 1978.

[12] R. P. Cook and I. Lee, "A contextual analysis of Pascal programs," *Software: Practice and Experience*, vol. 12, pp. 195–203, Feb. 1982.

[13] H. J. Saal and Z. Weiss, "Some Properties of APL Programs," in *Proceedings of Seventh International Conference on APL*, APL '75, (New York, NY, USA), pp. 292–297, ACM, 1975.

[14] H. J. Saal and Z. Weiss, "An empirical study of APL programs," *Computer Languages*, vol. 2, pp. 47–59, Jan. 1977.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, (New York, NY, USA), pp. 169–190, ACM, 2006.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *2010 Asia Pacific Software Engineering Conference*, pp. 336–345, Nov. 2010.

[17] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, "How (and why) developers use the dynamic features of programming languages: the case of smalltalk," *Empirical Software Engineering*, vol. 18, pp. 1156–1194, Dec. 2013.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), pp. 1–12, ACM, 2010.

[19] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, (Berlin, Heidelberg), pp. 52–78, Springer-Verlag, 2011.

[20] M. Madsen and E. Andreasen, "String Analysis for Dynamic Field Access," in *Compiler Construction*, Lecture Notes in Computer Science, pp. 197–217, Springer, Berlin, Heidelberg, Apr. 2014.

[21] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 270–283, ACM, 2010.

[22] M. Hills, P. Klint, and J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, (New York, NY, USA), pp. 325–335, ACM, 2013.

[23] J. Dahse and T. Holz, "Experience Report: An Empirical Study of PHP Security Mechanism Usage," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, (New York, NY, USA), pp. 60–70, ACM, 2015.

[24] M. Doyle and J. Walden, "An Empirical Study of the Evolution of PHP Web Application Security," in *2011 Third International Workshop on Security Measurements and Metrics*, pp. 11–20, Sept. 2011.

[25] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor, "An Empirical Study on the Usage of the Swift Programming Language," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 634–638, Mar. 2016.

[26] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A Large-scale Study of Programming Languages and Code Quality in GitHub," *Commun. ACM*, vol. 60, pp. 91–100, Sept. 2017.

[27] J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada, "Contracts in the Wild: A Study of Java Programs," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)* (P. Müller, ed.), vol. 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 9:1–9:29, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[28] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 507–518, May 2017.

[29] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 64–73, Feb. 2014.

[30] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining Programmer Practices for Locally Handling Exceptions," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 484–487, ACM, 2016.

[31] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How Developers Use Exception Handling in Java?," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 516–519, ACM, 2016.

[32] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of Exception Handling Patterns in Java Projects: An Empirical Study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 500–503, ACM, 2016.

[33] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, (Piscataway, NJ, USA), pp. 134–145, IEEE Press, 2015.

[34] C. Parnin, C. Bird, and E. Murphy-Hill, "Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, (New York, NY, USA), pp. 3–12, ACM, 2011.

[35] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Software Engineering*, vol. 18, pp. 1047–1089, Dec. 2013.

[36] D. Costa, A. Andrzejak, J. Seboek, and D. Lo, "Empirical Study of Usage and Performance of Java Collections," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, (New York, NY, USA), pp. 389–400, ACM, 2017.

[37] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 595–609, Sept. 1984.

[38] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, (New York, NY, USA), pp. 220–233, ACM, 1980.

[39] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, pp. 23–29, Oct. 2000.

[40] T. D. LaToza and B. A. Myers, "Developers Ask Reachability Questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 185–194, ACM, 2010.

[41] A. Stuchlik and S. Hanenberg, "Static vs. Dynamic Type Systems: An Empirical Study About the Relationship Between Type Casts and Development Time," in *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, (New York, NY, USA), pp. 97–106, ACM, 2011.

[42] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 683–702, ACM, 2012.

[43] I. R. Harlin, H. Washizaki, and Y. Fukazawa, "Impact of Using a Static-Type System in Computer Programming," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pp. 116–119, Jan. 2017.

[44] B. Wu and S. Chen, "How Type Errors Were Fixed and What Students Did?," *Proc. ACM Program. Lang.*, vol. 1, pp. 105:1–105:27, Oct. 2017.

[45] B. Wu, J. P. Campora III, and S. Chen, "Learning User Friendly Type-error Messages," *Proc. ACM Program. Lang.*, vol. 1, pp. 106:1–106:29, Oct. 2017.

[46] D. Posnett, C. Bird, and P. Devanbu, "THEX: Mining metapatterns from java," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 122–125, May 2010.

[47] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," in *In Adaptable and extensible component systems*, 2002.

[48] E. Kuleshov, *Using the ASM framework to implement common Java bytecode transformation patterns*. 2007.

[49] L. Hu and K. Sartipi, "Dynamic Analysis and Design Pattern Detection in Java Programs.," in *20th International Conference on Software Engineering and Knowledge Engineering, SEKE 2008*, pp. 842–846, Jan. 2008.

[50] F. Arcelli, F. Perin, C. Raibulet, and S. Ravani, "Design Pattern Detection in Java Systems: A Dynamic Analysis Based Approach," in *Evaluation of Novel Approaches to Software Engineering*, Communications in Computer and Information Science, pp. 163–179, Springer, Berlin, Heidelberg, May 2008.

[51] J. Vinju and J. R. Cordy, "How to make a bridge between transformation and analysis technologies?," in *Transformation Techniques in Software Engineering* (J. R. Cordy, R. Lämmel, and A. Winter, eds.), Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[52] P. Klint, T. v. d. Storm, and J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 168–177, Sept. 2009.

[53] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 422–431, May 2013.

[54] R. Dyer, H. Rajan, and T. N. Nguyen, "Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, (New York, NY, USA), pp. 23–32, ACM, 2013.

[55] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 779–790, ACM, 2014.

[56] G. Gousios, "The GHTorent Dataset and Tool Suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, (Piscataway, NJ, USA), pp. 233–236, IEEE Press, 2013.

[57] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean GHTorrent: GitHub Data on Demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 384–387, ACM, 2014.

[58] N. M. Tiwari, G. Upadhyaya, H. A. Nguyen, and H. Rajan, “Candoia: A Platform for Building and Sharing Mining Software Repositories Tools as Apps,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 53–63, May 2017.

[59] S. Brandauer and T. Wrigstad, “Spencer: Interactive Heap Analysis for the Masses,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 113–123, May 2017.

[60] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “DéJàVu: A Map of Code Duplicates on GitHub,” *Proc. ACM Program. Lang.*, vol. 1, pp. 84:1–84:28, Oct. 2017.

[61] M. Nagappan, T. Zimmermann, and C. Bird, “Diversity in Software Engineering Research,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 466–476, ACM, 2013.

[62] M. A. Saied, O. Benomar, and H. Sahraoui, “Visualization based API usage patterns refining,” in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pp. 155–159, Sept. 2015.

[63] M. Kechagia and D. Spinellis, “Undocumented and Unchecked: Exceptions That Spell Trouble,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 312–315, ACM, 2014.