Large-Scale Study of Substitutability in the Presence of Effects

Jackson Maddox
Iowa State University, USA
jlmaddox@iastate.edu

Yuheng Long*
Google Inc., USA
csgzlong@iastate.edu

Hridesh Rajan
Iowa State University, USA
hridesh@iastate.edu

ABSTRACT

A majority of modern software is constructed using languages that compute by producing side-effects such as reading/writing from/to files, throwing exceptions, acquiring locks, etc. To understand a piece of software, e.g. a class, it is important for a developer to understand its side-effects. Similarly, to replace a class with another, it is important to understand whether the replacement is a safe substitution for the former in terms of its behavior, a property known as *substitutability*, because mismatch may lead to bugs. The problem is especially severe for superclass-subclass pairs since at runtime an instance of the subclass may be used in the client code where a superclass is mentioned. Despite the importance of this property, we do not yet know whether substitutability w.r.t. effects between subclass and superclass is preserved in the wild, and if not what sorts of substitutability violations are common and what is the impact of such violations. This paper conducts a large scale study on over 20 million Java classes, in order to compare the effects of the methods of subclasses and superclasses in practice. Our comprehensive study considers the exception, synchronization, I/O, and method call effects. It reveals that in pairs with effects, only 8-24% have the same effects, and 31-56% of submethods have more effects, and the effects of a large percentage of submethods cannot be inferred from the supermethod.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Empirical software validation*; *Maintaining software*;

KEYWORDS

Substitutability, Side-effects, Object-oriented Software Engineering

ACM Reference Format:
Jackson Maddox, Yuheng Long, and Hridesh Rajan. 2018. Large-Scale Study of Substitutability in the Presence of Effects. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236075

*This work was done when the author was at Iowa State University.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236075

1 INTRODUCTION

A huge amount of software has been written in Object-Oriented languages such as Java, C#, and C++. These languages frequently contain methods with side effects in order to perform work. Side effects can include a variety of operations such as throwing exceptions, acquiring a lock, and reading from a file. Inheritance is another frequently used feature, and is often used to organize and modularize code by subtyping a class and overriding methods.

With these two features, understanding how to safely use classes or APIs requires understanding both the methods' side effects and the side effects of any submethods that may override them. Doing this exhaustively is difficult for two reasons. First, determining a single method's side effects requires delving into the implementation of it and any other methods it may call. Secondly, inheritance hierarchies mean re-doing this process again for each overriding submethod due to polymorphism.

This process is highly simplified if we assume that the Liskov substitution principle [22, 23] holds. The principle says that a subtype should be able to replace a supertype without changing the supertype's properties such as its correctness or, as is this paper's concern, its side effects. Under this assumption we would not need to examine the submethods. This leads us to ask several questions. Is this principle being upheld in practice? If not, in what ways is the principle being violated? Are the violations problematic, and if so, how? This paper's focus is on answering these questions.

To see why violating substitutability is a problem, consider the following example.

```
1 class Services {
2   Map<Integer,Object> map;
3   synchronized void addService(int id, Object service) {
4     if (map.containsKey(id)) return;
5     map.put(id, service);
6   } }
```

From this implementation, we can infer that the `addService` method is thread-safe due to the `synchronized` keyword. It also does not throw exceptions so there is no error case to handle. We might then have a client of this class such as the one below:

```
1 Services s = App.getServicesList();
2 s.addService(0, service0);
```

The client retrieves a Services instance from a factory and adds a service 0. It can be executed multiple times. Now suppose the `App.getServicesList` method returned a subclass:

```
1 class TrackedServices extends Services {
2   @Override void addService(int id, Object service) {
3     if (map.containsKey(id)) throw new IllegalArgument();
4     map.put(id, service);
5     configFile.println(id + "=" + service);
6   } }
```

Immediately we see several unexpected behavioral differences. Firstly, if our client ran twice, it would now unexpectedly crash with `IllegalArgument` since the 0^{th} ID was added after the first run. Secondly, other clients may run concurrently and require thread

safety to avoid data races, which they do not get in this subclass. Finally, the subclass outputs to a configuration file that could change program-wide behavior. Clearly, this subclass both violates substitutability and causes problems for its clients.

Motivated by these problems, we aim to answer the following research questions.

RQ1: Can we accurately infer a subclass's effects based solely on the superclass's implementation or do the subclasses often violate substitutability?

RQ2: When violations occur, how do the two implementations differ and what common patterns arise throughout all violations?

RQ3: When violations occur, what is the impact? For example, it could be the source of a substitutability bug [32, 33] or a code smell.

This paper investigates these questions on real world Java projects hosted on GitHub. We present a comprehensive study comparing the effects of superclasses to their subclasses in these projects using the Boa infrastructure [14, 15, 17]. Boa allows us to examine the Java projects' ASTs to determine the side effects of a method and compare them to its super and submethods [16]. To generalize our findings, this study focuses on a broad range of effect kinds including exception, synchronization, and I/O, and method call effects.

We find that only 8-24% method pairs have the same effects, and 31-56% of submethods have *more* effects than their supermethod when considering the effect types independently. Finally, when analyzing method pairs in terms of these effects, we confirm the above: a large percentage of the effects of submethods cannot be inferred through the supermethod's implementation when at least one of them has an effect. An artifact including the source and results is made available at http://design.cs.iastate.edu/papers/FSE-18/.

The rest of this paper will discuss our methodology, results, threats to validity, the related work, future work, and conclude.

2 METHODOLOGY

Our substitutability study was conducted in two phases: automated analysis followed by manual inspection. For the first we analyzed Boa's September 2015 GitHub dataset, containing 380,125 Java projects and excluding forks [14]. Our analysis collected effect information on each non-abstract method, made scalable by using a lightweight syntax directed effect inference based on [37]. After this process, the analysis pairs each non-private, non-static, non-constructor method with the one they override, if any, forming many *method pairs* containing a *submethod* and *supermethod*. This pairing considers normal sub/superclass pairs and includes nested non-anonymous classes, but not default interface methods as the Boa infrastructure uses Java 1.7. Each pair is then categorized by comparing the effects of the two methods with each other. The exact categories used depend on the effect being examined. We then use the pair categorization in order to investigate the effect differences in method pairs, and later examine a few categories that suggest substitutability violations. Table 1 shows the counts of examined projects, classes, and more in the dataset.

To make our analyses more accurate, we added a few exceptions: Firstly, pairs containing abstract methods are ignored since there is no implementation to compare with. Files containing JUnit tests, identified by use of the `@Test` annotation, are also ignored in an effort to skip mock objects used only for testing. Finally, we found that many projects have duplicate class files identified by the package name plus the file name. In these cases we took the file that was last modified or, if both were modified in the same commit, we discard both. We assume the last modified file is the latest version of it, but if the duplicates are modified in the same commit it can indicate that they are separate, conflictingly named, parts of the program. The reasoning for this is that we want to avoid counting a class twice and to be sure that a given pair is correctly identified. While our results do not filter out small projects, we found that doing so does not change any of the trends in our results.

Table 1: Summary of the data studied

Metric	Count
# of projects	380,125
# of source files	20,302,663
# of classes	20,569,922
# of methods	149,294,833
# of method pairs	5,975,136
# of project AST nodes	6,619,264,814

The second phase consisted of studying cases sampled from the analysis output in order to better understand certain categories of method pairs. We are interested in seeing if certain categories suspected of substitutability violations are, in fact, problematic. In order to do this, the analysis output was first filtered based on the desired category and then randomized. From this, we randomly gather samples, discarding those pairs which cannot be found due to, for example, the repository having been deleted.

For each sample, we manually inspected and compared the implementations of the sub and supermethods to see if the submethod's effects cannot be inferred from the supermethod's body (a substitutability violation). Transitive effects from methods calls were also considered where possible. If a substitutability violation is discovered, we define a call sequence that, if executed, could trigger the effect difference. An example of such a violation would be a call sequence causing the submethod to throw an exception that the supermethod will never throw. The resulting set of violations was then examined in order to discover the most common patterns making them up. This case study procedure was conducted by the first author, and then the second author confirmed the labels given to each of the samples.

The side effects that this study focuses on are exceptions, synchronization, I/O operations and method calls. The reason for studying method calls as an effect is twofold. First, it is an important effect kind well explored [30]. Second, our effect inference tool ignores transitive effects from method calls so that it can scale up to very large datasets. Any transitive call analysis would also be very imprecise due to both the open world assumption [34] and dynamic dispatch [26]. The method call effect provides a well-understood technique for handling it by treating transitive calls abstractly in both the super and the subclass. For the rest of this section we detail how each of these side effects were analyzed.

In order to study exception effects, methods were assigned an exception effect if they contain a throw statement [28]. Our effect

inference then attempts to find the type of each thrown exception and if successful adds it to the method's set of thrown exception types. Otherwise the exception is given a generic `UNKNOWN` type and added to the set, but this occurred in only 4% of method pairs with an exception effect. Exceptions that may be thrown implicitly by *e.g.* accessing a null object or accessing an array out of its bounds are ignored as is standard in exception analyses.

For synchronization, we differentiate between acquiring and releasing locks. The analysis considers the `synchronized` keyword (either as a modifier or block) to be both acquiring and releasing a lock. We also consider Java's `Lock` interface, its implementations, and the `Semaphore` class. In order to include the fine-grain lock objects, the analysis examines each method call. From this it attempts to infer the type of the callee and when successful looks up the type and method in a table and assigns the appropriate effect, if any.

In methods with I/O operations, we distinguish between operations related to input and output. We consider a list of 52 Java standard I/O classes and interfaces, and assign an effect to each of these types' methods. In addition, operations on `System.out`, `System.err`, and `System.in` are added as special cases to the analysis. Similarly to synchronization, this analysis attempts to infer the type of object called and assign the appropriate effect.

Finally, for the method call effect, we consider each call expression to be a pair consisting of a category and the method name (standard in the intra-procedural phase of effect inferences [36]). These categories are `this`, `super`, and `other`. The `this` category means the callee is the object `this`, and similarly for `super` while the `other` category covers the remaining method calls. Each of these pairs are added to a set representing the method's call effect.

3 RESULTS

This section describes the results of our large scale study for each of the aforementioned side effects. Table 2 lists the number of methods in the dataset whose method bodies contain different combinations of explicit side effects. The table shows that about 32% of methods have no studied side effects, and at least 9% have side effects other than method calls.

Table 2: Effect kinds per concrete method

Exception	Sync	I/O	Call	# Dataset Methods
✗	✗	✗	✗	47,215,362
✗	✗	✗	✓	84,729,294
✗	✗	✓	✓	3,989,859
✗	✓	✗	✗	357,272
✗	✓	✗	✓	2,082,889
✗	✓	✓	✓	108,370
✓	✗	✗	✗	1,752,771
✓	✗	✗	✓	6,372,803
✓	✗	✓	✓	540,276
✓	✓	✗	✗	23,742
✓	✓	✗	✓	333,082
✓	✓	✓	✓	28,862
Total Methods				147,534,582

Table 3 shows data for submethod and supermethod pairs that have at least one effect. The submethod has more effects category means the submethod has "more" effects than the supermethod in at least one side-effect kind and none where the supermethod has "more". For example a submethod reading from a file where the supermethod does not would fit this category. The data shows that for a large number of cases (53%) the submethod has more effects compared to the supermethod.

Table 3: Effect comparison of pairs with effects

Submethod has more effects	2,680,336 (53%)
Supermethod has more effects	1,287,747 (25%)
Both have same effects	395,940 (8%)
All others	710,258 (14%)
Total Pairs	5,074,281

Next, we examine each side effect over all methods in the dataset, and then move onto examining pairs of sub and supermethods. For each pair, we compare the effects of the paired methods in order to see how many have substitutability violations, and verify this by examining a small subset of pairs in more detail.

3.1 Exceptions

We begin our examination of exception effects by first looking at how many individual methods explicitly throw an exception. These results are shown in Table 4, which categorizes methods by the number of different exception types they throw. In this table, all private methods are grouped together, including private constructors and static methods. The next 2 columns denote non-private static methods and constructors respectively, and the "other" column denotes all remaining methods.

Our results show that the vast majority (94%) of methods do not explicitly throw an exception. For those that do, most will only throw a single exception type regardless of the type of method. Proportionally, private and static methods throw exceptions more often, but there are far more methods in the "other" category throwing exceptions than those two combined.

These results are so far consistent with the previously published study on purity analyses [36, 39], although our study considers a thousand times more projects and, later on, other effect kinds.

Table 4: Number exceptions types thrown per method kind (NP means non-private, and Init means constructor)

# Types	Private	NP Static	NP Init	Other
0	14M (91%)	12M (92%)	14M (96%)	99M (94%)
1	1.2M (8%)	961K (7%)	509K (3%)	5.6M (5%)
2	86K (1%)	92K (1%)	39K (0%)	462K (0%)
3+	14K (0%)	12K (0%)	3017 (0%)	98K (0%)
Total	14,936,491	12,938,996	14,883,048	104,776,047

We now turn to examine method pairs where at least one of the methods contains a throws expression. Table 5 shows our comparison of the sub and supermethods for these pairs. Each pair's methods were compared by the set of exception types each throws. The pair was then placed into one of 6 categories depending on both how the submethod's set's cardinality compared to the supermethod's and whether one was a subset of the other. In each

category, *sub* refers to the set of exception types the submethod throws and *super* the set of exceptions the supermethod throws.

The pairs themselves are additionally split into an "Expected" and "Unexpected" grouping. The term "Expected" refers to pairs where the set of exceptions thrown in the submethod are covered by either the supermethod implementation, its throws clause, or both. Listing 1 provides an example of this. In this example, the submethod throws `IllegalArgument` and `IOException`. However, it is consistent with the supermethod because the supermethod throws `IllegalArgument` and the throws clause declares that it may throw `IOException`. Thus, it would be categorized as *Expected*.

```
1  class Super {
2    void m(Object x) throws IOException {
3      if (x == null) throw new IllegalArgument();
4    } }
5
6  class Sub extends Super {
7    @Override void m(Object x) throws IOException {
8      if (x == null) throw new IllegalArgument();
9      else throw new IOException();
10   } }
```

Listing 1: Example pair where submethod's exceptions are expected

Table 5: Exception pair types (s=sub, p=super)

Category	Pairs		
	Unexpected	Expected	**Total**
$s \subset p$	0 (0%)	350K (67%)	350K (48%)
$s \not\subset p, \lvert s\rvert < \lvert p\rvert$	5,399 (3%)	643 (0%)	6,042 (1%)
$s \supset p$	163K (79%)	61K (12%)	224K (31%)
$s \not\supset p, \lvert s\rvert > \lvert p\rvert$	3,203 (2%)	240 (0%)	3,443 (0%)
$s = p$	0 (0%)	103K (20%)	102K (14%)
$s \neq p, \lvert s\rvert = \lvert p\rvert$	34K (17%)	7,222 (1%)	41K (6%)
Total Pairs	205,844	521,880	727,724

Our results indicate that 38% of submethods may be unsafe substitutes of the superclass equivalent when considering only the method implementations. When including the throws clause, this drops to 28%, which is still a large percentage of these pairs. This finding motivates further study to understand whether these cases indicate substitutability violations. To do this, we start by examining what types of exceptions are most common in this situation.

Finding 1: In one quarter (28%) of the exception pairs, submethod effects cannot be inferred from the supermethod's body or throws clause. We will see similar results for the other three effect kinds.
Implication: Programmers and tools should consider submethod implementations when analyzing a method's effects.

Figure 1 shows exceptions most commonly thrown by pairs where $|sub| > |super|$. Notice that several of these exceptions are related to preconditions such as `IllegalArgument` and `IllegalState`. This may indicate that these submethods have stronger preconditions than their supermethod, which is problematic from the viewpoint of supertype abstraction [22].

Another top exception, `UnsupportedOperation`, is intended for an operation a class cannot support. Submethods throwing this exception likely indicate a violation of substitutability in which a client may expect a class to support an operation that it does not

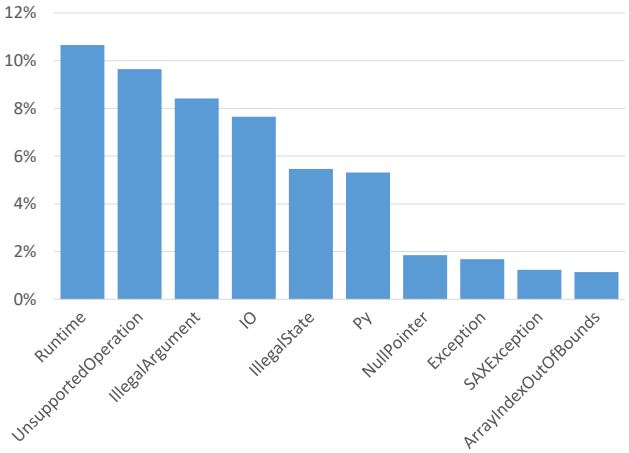

Figure 1: Top 10 exceptions thrown when the submethod throws more types (Exception suffix omitted)

due to being a subclass instance. The case study investigates these two groups in more detail.

Finally, `RuntimeException`, the most common, is a very generic exception to throw, and is used for a variety of different purposes. In most cases, a subclass of this exception should be thrown instead, so that clients catching the exception have some idea of what the particular error is. Thus, throwing this exception likely indicates a code smell. Even when normalizing Figure 1 by project (*i.e.* each project can only count 1 towards an exception), `RuntimeException` is still the top exception type. This indicates that these kinds of pairs are not specific to a few large projects, but many.

3.1.1 Are Exception Substitutability Violations Problematic? To understand this, we start by examining exceptions related to preconditions whose submethods throw more types of exceptions than the supermethod. This group is further filtered so that the method pair must throw at least one of `IllegalArgumentException`, `NullPointerException`, `IllegalStateException`, or `IndexOutOfBoundsException` in addition to the array and string variants of the out-of-bounds exception. From this set, we sample 50 method pairs that we examine for substitutability violations. In the context of exceptions, this means that we can construct a call sequence that would cause the submethod to throw an exception while the supermethod either throws a different or no exception. *We find that 80% of these cases indicate a violation.*

In general, most of the violations have at least one of 3 common patterns. The first is brittle parameters [33] either via `instanceof` (16 cases) or restricting the allowed range of values (8 cases). The supermethods generally do not have any explicit parameter restrictions in comparison, and do not throw the precondition-related exceptions the submethods throw. The second pattern consists of state-related exceptions, almost completely characterized by `IllegalStateException` being thrown, encompassing 11 cases. The general pattern for these is that the submethod checks to ensure the object state is valid while the supermethod has no such checks in place, mostly due to not having the extra state that the subclass has. For the third group (7 cases), the sub and supermethods have the same constraints, but handle it differently. For example

a supermethod may just return on a null argument while the submethod instead throws `NullPointerException`.

This characterization leaves 6 violations, 4 of which fit into multiple of the above patterns, and 2 that fit into none of the above. We will now examine some of the violations in more detail.

The first pair (from [1]) we examine is a case of a brittle parameter shown in Listing 2, and is from the Java Swing library. The superclass `InputMap` accepts any other `InputMap` or subclass of it as a parent in the `setParent` method. One may reasonably expect that, given any object of that type, we can assign the parent to be any other object of the same type. However, this assumption does not hold with a subclass. Instead, `ComponentInputMap` only accepts other instances of that subclass as parent. Clearly we cannot rely on the static type to tell us what the potential effects of `setParent` are, nor do we intuitively expect it to ever throw an exception.

```
1  class InputMap {
2    public void setParent(InputMap parentMap) {
3      parent = parentMap;
4    } }
5  class ComponentInputMap extends InputMap {
6    public void setParent(InputMap parentMap) {
7      if (parentMap != null &&
8          !(parentMap instanceof ComponentInputMap))
9        throw new IllegalArgumentException();
10     ...
11   } }
```

Listing 2: Preconditions added by the subclass

Our second example (from [2]) shown in Listing 3, is a case where the submethod has a state-related precondition via a switch statement. The non-constant static variables referenced in these methods are all set by the client of these classes. While the superclass may throw `IndexOutOfBoundsException` if `tag` is set incorrectly, the submethod may instead throw `IllegalStateException` if it cannot handle a certain constant object's tag. What makes this example interesting, is that clients of the superclass in the project avoid calling the submethod by using `instanceof` to check the runtime type of their `Attribute` variable. However, checking the other samples shows that this pattern does not appear, indicating that it is rare for a client to check runtime types explicitly before calling a method when it contains a substitutability violation.

```
1  abstract class Attribute {
2    public String toString() {
3      return Constants.ATTRIBUTE_NAMES[tag];
4    } }
5  class ConstantValue extends Attribute {
6    public String toString() {
7      Constant c = constant_pool.getConstant(constantvalue_index);
8      switch (c.getTag()) {
9        case Constants.CONSTANT_LONG: buf = ...; break;
10       ...
11       default: throw new IllegalStateException(...);
12     } } }
```

Listing 3: Different handling of the same precondition

Our results show that only in one case do clients use reflection (via `instanceof`) [25] to avoid an exception being thrown by a subtype method. Most clients do not handle substitutability violations explicitly by checking the object runtime type and that is an indication that violations can lead to mostly latent bugs.

We also sampled 20 method pairs where the submethod throws more exceptions than its supermethod and also throws `UnsupportedOperationException`. *All 20 of these pairs violated substitutability.* When examining the submethods, we find 4 patterns. For 5 examples, the exception is used to restrict changes and notify the client. This is to either make the subclass immutable, or to enforce a specific container size in one case. For another 4 examples this exception is thrown due to the submethod being more specialized and unable to support an operation as stated in either Javadoc or the exception message itself. Thirdly, in only 1 case, the submethod uses `UnsupportedOperation` as a "to do" that was never completed. Finally, in the 9 remaining examples no explanation is given through method documentation or as an exception message, making it difficult to ascertain why the submethod is unimplemented.

Finding 2: Program patterns where the submethod appears to have more effects than the supermethod are often (80%) indicators of substitutability violations.
Implication: Code smell detection tools can accurately warn about substitutability violations from submethods that explicitly throw exceptions not found in the supermethod.

3.1.2 Are Developers Documenting Exception Substitutability Violations? We found that the majority of the time the answer to this question was no, though a number of cases were documented.

We examined the context in which these method pair implementations exist in order to see if developers are aware of and document substitutability violations. In 6 cases, the supermethod's Javadoc declares that the method may throw the submethod's exceptions where 2 of these supermethods include a throws clause for the unchecked exceptions. However, for the remaining it is impossible to infer the exception side-effect from the supermethod alone. In 5 cases the subclass Javadoc alone states either the exception or at least the precondition. The Javadocs in these submethods indicate potential points of code smell where the supermethod's Javadoc simply was not updated. Finally, while Javadocs do not provide information about either the exception or preconditions in the remaining cases, 18 at least provide a exception message though 6 do not, providing no information on why the exception occurred.

Beyond looking at Javadocs and exception messages, we found evidence that some of the method pairs indicate problems, or bugs. In one case the Javadoc of both the sub and supermethods is incorrect for the submethod's implementation. For example, in a repository clone [3] of the JFreeChart library, the Javadoc for `VectorRenderer.findDomainBounds` states that the null argument is permitted, and indeed it is in the supermethod. However, the submethod throws `IllegalArgumentException` if the argument is null, indicating that a mistake may have been made in the submethod's implementation. A similar situation was found in a repository [4] that contains a copy of the Java Swing library. Specifically the `TitledBorder.getBaseline` method whose Javadoc indicates that the submethod expects its supermethod Javadoc to say that `NullPointerException` may be thrown (via the `{@inheritDoc}`), but it does not. One other sample has a similar inconsistency between the sub and supermethod documentation.

We also found two other cases where the submethod's precondition is likely valid in the supermethod. However, the supermethod does not check that the precondition is true. For example in one repository [5], a clone of GNU Classpath, the `BandedSampleModel.getDataElement` (submethod), checks the bounds of the arguments

specifying an (x, y) coordinate. However, the supermethod does not though some other methods in the superclass do bounds checking.

3.2 Synchronization

Next, we examine synchronization effects, either via one of the Java standard library locks or using the `synchronized` keyword. Methods may have no locking effect, acquire or release a lock, or do both. No distinction is made between what object is used for the lock. This is because many classes may use multiple different lock objects to ensure more fine-grained thread-safety, so it will not necessarily provide useful information when examining these methods. However, we group these methods and pairs by how fine-grained their locks themselves are. That is, whether they use a lock object (finest), synchronized blocks (fine), or just the synchronized method modifier (coarse).

First we look at individual methods in Table 6. The columns represent the aforementioned lock grouping based on how fine grained the lock is, and the rows represent whether the method acquires, releases, or both acquires and releases a lock. Note that the "Block" column, due to how synchronized works in Java, always acquires and releases the lock. Secondly, the "Modifier" column represents both methods that use only the synchronized modifier and those who have no synchronization effects. The entries with "N/A" indicate that the entry is not applicable for the group.

The vast majority of methods with synchronization use the keyword, likely due to ease of use. These methods split almost evenly between using synchronized blocks and the method modifier. For those using a lock object from the standard library, most both acquire and release locks, indicating that they might not rely on other methods to release or acquire the necessary locks. However, 15% of methods with lock objects seem to expect a different method to acquire or release a lock, indicating more complex locking scenarios.

Table 6: Synchronization types per method

Type	Lock	Block	Modifier
None	N/A	N/A	145M (99%)
Acquire	8,488 (8%)	N/A	N/A
Release	7,134 (7%)	N/A	N/A
Both	94K (86%)	1.4M (100%)	1.5M (1%)
Total	109,408	1,373,460	146,051,714

Next is to examine the method pairs with synchronization shown in Table 7. Similar to the previous table, method pairs are grouped based on how fine-grained their locks are with the exception that "Modifier" does not include pairs without synchronization. In the table categories, a method `m` having "more" of an effect than the other, `m'`, indicates that `m'` has no effect and `m` does or `m'` acquires or releases a lock and `m` does both. The same category similarly means that the sub and supermethods have the same kind of synchronization effect, and different means one method acquires a lock while the other releases it. This last row is only applicable to locks since the synchronized keyword includes both acquire and release.

The different category would likely indicate a bug, but since no pair fit into this category, we cannot examine this idea further. However, a significant percentage of pairs have a supermethod with synchronization and a submethod that does not, and may indicate the presence of substitutability bugs. Consider for example, a supermethod that is thread-safe, but a submethod that is not. If a client expects the superclass, but gets a subclass, it may assume it is thread-safe and use the object among multiple threads. This leads to race conditions and visibility bugs. Visibility bugs occur when a thread caches a value it updated without making it available to other threads, leading to stale values and inconsistent object state.

Table 7: Synchronization pair types

Category	Pairs			
	Lock	Block	Modifier	**Total**
Super more	1,741 (51%)	33K (38%)	24K (25%)	58K (31%)
Sub more	1,366 (40%)	36K (41%)	44K (47%)	82K (44%)
Same	288 (8%)	18K (21%)	27K (28%)	45K (24%)
Different	0	N/A	N/A	0
Total	3,395	86,821	94,912	185,128

3.2.1 Are there Synchronization Substitutability Violations and what Kinds? To understand this, we sampled 50 pairs where the supermethod has more synchronization effects than its submethod out of 58K cases. The idea behind this is that, perhaps synchronization in the subclass has been forgotten, causing a client that believes an object with the static type of the superclass to be thread-safe when it is not. Will two threads that operate safely on the superclass cause a potential race condition or state visibility bug when operating on the subclass? We assume that the two threads initially obtained a properly published instance of the object in question. We also assume the first thread knows of the subclass object, while the second thread is only aware of the superclass.

Out of the selection, 15 or 30% have a substitutability violation caused via a race condition or value visibility problem. Of these cases, 7 of them are violations that will cause visibility problems of some sort, such as the fields of the subclass getting out of sync across threads. Then in 4 cases, two threads calling the submethod can end up doing different things due to a visibility issue caused during execution of the call sequence. 3 of the violations from the aforementioned cases require one of the threads to be aware of the subclass and call subclass-specific methods. Then for 3 cases the violation leads to a race condition such as when manipulating a thread unsafe collection or other object. Finally, the last violation is due to the supermethod being synchronized but only throwing `UnsupportedOperation` whereas the submethod is unsafe.

```
1  class PreparedStatement {
2    void setNCharacterStream(
3        int parameterIndex, Reader reader, long length) {
4      synchronized(checkClosed()) { ... }
5    } }
6  class JDBC4ServerPreparedStatement
7  extends ServerPreparedStatement /*extends PreparedStatement*/ {
8    void setNCharacterStream(
9        int parameterIndex, Reader reader, long length) {
10     ...
11     // both calls supposedly thread-safe
12     BindValue binding = getBinding(parameterIndex, true);
13     setType(binding, MysqlDefs.FIELD_TYPE_BLOB);
14     binding.value = reader;
15     ...
16   } }
```

Listing 4: Reference visibility race condition

Let us now examine two of these substitutability violations. The first (from [6]) as shown in Listing 4 contains a supermethod that enters a synchronized block, does some work, and exits. In comparison, the submethod has no explicit synchronization. Instead, it calls multiple methods that are already thread-safe (*e.g.* `getBinding()` and `setType()`). However, after this it updates a `BindValue` structure that is stored in its superclass's state without synchronization. So if two threads both call this method on the same subclass instance, they may see different versions of the same or other `BindValues`. Whether this leads to a bug depends on what the threads do (or do not do) next. For example if thread 1 attempts to retrieve the binding thread 2 modified and thread 2 has not entered a lock by then, thread 1 may receive outdated information.

The second example (from [7]) is a case of unsafe operations on collections and shown in Listing 5. In the superclass, most methods are marked as synchronized except for several setters (not shown). However, the subclass has no synchronization for its `HashSet _prefixes`. In a case where two threads have an instance of the subclass and one knows the runtime type, that thread may add to the collection and call the `addABox` method. However, when the second thread calls the sub method as well, it may see an inconsistent or outdated version of the collection that could cause a call that would otherwise pass the filtering to fail.

```
1  class LinkFilterDefault {
2    synchronized void addABox(Node[] nx, int i) {
3      addUri(nx, i);
4    }
5    synchronized void addUri(Node[] nx, int i) { ... } }
6  class LinkFilterPrefix extends LinkFilterDefault {
7    void addPrefix(String prefix) { _prefixes.add(prefix); }
8    void addABox(Node[] nx, int i) {
9      boolean found = false;
10     for (String prefix: _prefixes) // a HashSet in sublass
11       if (nx[i].toString().startsWith(prefix)) found = true;
12     if (found) super.addUri(nx, i);
13   } }
```

Listing 5: Unsafe operations

Now that we have examined the violations, let us look at the cases with no synchronization violation. In 11, the submethod is either trivial (returns, empty method, throws exception) or delegates the call to the supermethod with no changes in logic. For 5 more cases, the submethod delegates calls to an object that either itself delegates calls or is thread-safe. In 7 cases, the submethod conducts only thread-safe operations on state (such as calling a thread-safe method), and while it may be possible to use them in a way that causes strange results, they do not appear to contain race conditions. Then in 8 cases the supermethod has very specific locking behavior, and alone may not be intended to be thread-safe. Finally, 1 case's safety depends on something a user would implement and 2 contain methods throwing `UnsupportedOperation`.

3.2.2 Do Developers Document Synchronization Substitutability Violations? Of the violations, in two cases the thread-safe superclass was from the AWT library and the unsafe subclass from the Swing library. Since the Swing library was built on top of the AWT library and, unlike AWT, did not attempt to be thread-safe, we consider these violations to not be bugs. Note that this determination required outside knowledge to understand the thread-safety of the particular subclass. In fact, we found no documentation of the thread-safety properties of any pair of the classes we studied in their Javadocs. This suggests a problem where a client of a superclass may assume certain operations are thread-safe whereas the subclass may have different ideas. From these cases, subtle race conditions or value visibility problems could appear that might not be discovered until much later.

For the remaining pairs with violations, it is more likely that the thread-safety of the subclass was not on the implementer's mind as there is no indication of locks or explicitly thread-safe collections.

Finding 3: There is sparse to no documentation on class thread safety guarantees, including between a sub and supertype. It can only be inferred from general knowledge of the project or examining the implementation.
Implication: Tools are needed first to help document synchronization guarantees and second warn on mismatches between the sub and supermethods based on these guarantees for large concurrent programs.

3.3 I/O

In this section, we categorize methods based on whether they have input, output, both, or no I/O operations. Table 8 shows how common different I/O operations are among all the methods analyzed. Similarly to exceptions and synchronization, the majority do not contain I/O operations. It turns out that the number of methods with output operations is about 2.7 for private methods, 3.6 for non-private constructors, 4.3 for non-private statics, and 4.2 for the rest times the number of methods with input operations.

Table 8: I/O effects per method (NP means non-private, and Init means constructor)

IO Type	Private	NP Static	NP Init	Other
No I/O	14M (94%)	12M (92%)	15M (99%)	102M (98%)
Output	664K (4%)	878K (7%)	94K (1%)	2.0M (2%)
Input	205K (1%)	107K (1%)	21K (0%)	351K (0%)
Both	79K (1%)	124K (1%)	6,973 (0%)	201K (0%)
Total	14,936,491	12,938,996	14,883,048	104,776,047

In order to investigate why there were so many more write operations across all types of methods, we examined what output I/O objects were used most. This was done by, for each type, counting the number of methods that do an I/O operation on an instance of it. We also make a distinction between any I/O objects and the standard `System.out` and `System.err` objects. As Figure 2 shows, `System.out` is the most commonly used by far followed by `PrintWriter` combined making up over 65% of output. Even if we ignore the percentage of methods with console output, making the assumption they only used `System.out` or `System.err`, writes will still outnumber reads in all 4 cases.

Now we discuss the comparison between method pairs with I/O effects displayed in Table 9. Like with synchronization, the categories for one method `m` having "more" I/O effects than the other (call `m'`) refers to either `m` having an I/O effect and `m'` not or `m` having both read and write while `m'` either write or reads.

Method pairs are grouped into one of three categories in the table based on what kind of I/O objects they use. If either method uses console I/O such as the `Console` class or `System.out` they are placed in the "Console" category. Otherwise if either uses file I/O

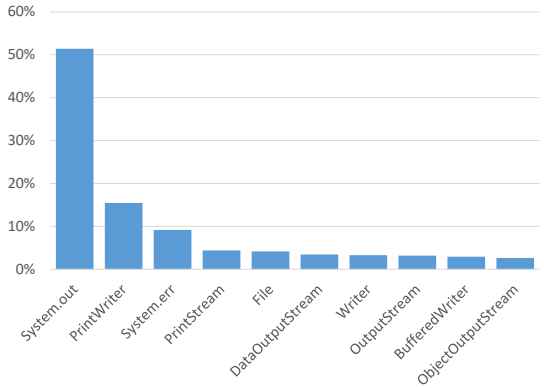

Figure 2: Top 10 I/O Output Types

objects then they are placed in "File". Finally, the "Bus" category signifies streams, readers, and writers, referring to how these objects may be used to move bytes from one stream to another. This last category includes pairs with file and console I/O if the streams are connected to a file or the standard output/input streams. However, this is not necessarily the case for all of these objects.

Table 9: I/O pair types

Category	Pairs			
	Console	File	Bus	**Total**
Sub less	123 (0%)	20 (0%)	70K (54%)	70K (37%)
Sub more	36K (75%)	11K (91%)	34K (26%)	82K (43%)
Same	12K (25%)	1077 (9%)	26K (20%)	39K (21%)
Different	77 (0%)	7 (0%)	116 (0%)	200 (0%)
Total	48,440	12,314	129,858	190,612

The table shows a marked difference between console and file I/O with the rest, where the first two pairs have a majority (75% - 91%) of submethods with more I/O effects. These types of pairs may indicate substitutability violations such as being able to crash the submethod but not the supermethod (or with different exceptions) due to an I/O error, or to cause unexpected output in the submethod. We investigate these types of pairs further in the I/O case study.

Finding 4: The majority (75%-91%) of method pair console and file-based I/O is located in submethods.
Implication: Tools that infer a call's potential I/O effects will benefit from examining methods overriding the callee.

3.3.1 What kinds of I/O Substitutability Violations are Present? As before, we gather 50 samples where the submethod has more I/O effects than the supermethod of which there are 82K pairs. There are two types of substitutability violations we consider. First being if the submethod throws an exception caused by an I/O error that in the supermethod causes a different or no exception to be thrown. Secondly whether there is an output difference (*e.g.* the submethod outputs debugging info where the supermethod outputs nothing)?

Overall, 37 or 74% of the selected cases has at least one of these violations. Of these, 9 are due to differing behavior from an I/O operation throwing an exception. The remaining 28 are due to output differences, 21 from directly using `System.out` or `System.err`.

For those examples in which a violation is caused by a thrown exception, all but 1 are due to a pattern of the submethod containing an implementation while the supermethod is trivial (simple return, throws an exception, or empty). In two of these cases we can cause the exception to occur by calling a shutdown or similarly named method. For the rest, the difference can be seen by providing an illegal file name or already closed stream to operate on.

The example Listing 6 (from [8]) is one of the I/O violations due to a thrown exception. The supermethod only closes the stream and then throws a particular exception, also indirectly outputting to `System.out` if the call to `close()` throws. However, the submethod, during its execution, outputs to the provided stream. Now when comparing the sub and supermethods in the case where the stream `os` has already been closed, we observe the following difference: The supermethod will simply output the error to `System.out`, but the submethod will throw `IOException` when it writes to the stream.

Of course, any client of these methods will need to handle both checked exceptions that the method declares. However, anyone who uses the superclass's implementation as a point of reference may be misdirected without also examining subclass implementations. The problem is exacerbated by the other examples where instead the supermethod is trivial, which could be seen as a hint to look elsewhere for the true effects of the method.

```
abstract class ImageParser {
 void writeImage(BufferedImage src, OutputStream os, Map params)
     throws ImageWriteException, IOException {
   try { os.close(); }
   catch (Exception e){ Debug.debug(e); } // output to System.out
   throw new ImageWriteException(...);
 } }
class GifImageParser extends ImageParser {
 void writeImage(BufferedImage src, OutputStream os, Map params)
     throws ImageWriteException, IOException {
   ...
   os.write(0x47);
   ...
 } }
```

Listing 6: I/O failure causing an exception in subclass

In many other cases the submethod instead produces (from the client's perspective) undesirable I/O output. One example (from [9]) is Listing 7 where the supermethod simply updates several class fields. The submethod operates similarly, but its operations can throw an exception, in which case it catches the exception and disables some checks. However, notice that in addition to this, the submethod outputs a message via `System.err`.

```
abstract class Sampler {
  void setInput(Image inImage) {
    this.inImage = inImage;
    this.inWidth = inImage.getWidth();
    ...
  } }
class Clip extends Sampler {
  void setInput(Image inImage) {
    ...
    try { ...
      inImgScaler = inImage.getWCS().getScaler();
      // inverse() can throw TransformationException
      inImgScalerInv = (Scaler) inImgScalar.inverse();
    } catch (TransformationException e) {
      System.err.println(...);
      pixelCheck = false;
      straddleCheck = false;
    } } }
```

Listing 7: I/O output difference

A client of the supertype will not notice that output had been produced. This subtype's output would cause corruption in console-based application output where the console is the user interface or is used to print results. A similar situation would occur in applications using logging facilities rather than writing to the console.

Similar patterns abound in the other related output violations. In 6 cases the output is due to detecting some sort of failure or announcing a warning. A more common pattern (10 cases) are cases where the console output appears to consist of debug-related statements. That is, statements that do not indicate errors nor necessarily useful information for a user. A single case contains both debugging and failure messages. Finally, the remaining 4 come from a console-based projects for which the output is neither apparent debugging or failure statements.

This leave 7 cases that do not use `System.out` or `System.err`, but still present output differences. In only one of these, the subclass transforms the input via another stream, but the remaining examples generally consist of straightforward string output to either a stream or writer (which doesn't throw `IOException`) object.

For those examples that turned out not to be violations, 7 are due to I/O operations that will never throw an exception. These I/O operations also either not result in actual I/O occurring or produce the same output as the supermethod when considering transitive calls. An example of this is writing to a non-subclassed StringWriter, which simply builds a string for its client. Then there are only 2 occurrences of `System.out` statements hidden behind a final debug flag set to false. Then 3 extend an existing Java stream class, which relies on abstract method calls to work. Finally, the last case contains what appears to be an output difference, with no apparent way to trigger it from the code provided.

3.3.2 Are these I/O Substitutability Violations Problematic? When we examine the 9 I/O violations caused by an exception, we note that in each of them, the exception effect can be inferred from the supermethod signature through the supermethod throws clause (in addition to the Javadoc in 2 cases). This suggests that exceptions thrown from I/O substitutability violations are not necessarily a problem, at least in Java with checked exceptions such as `IOException`. However, violations due to output differences paint a different story.

In general, the biggest problems would likely be caused by the submethod corrupting output unexpectedly. For example, 5 submethods appear to output debug or informational messages. To provide an example, an Android app[10] has a class `ChapterReader` whose methods use `System.out` that outputs some informational messages when certain methods are called. Interestingly the class also uses the standard Android `Log` class for other messages. Further examination shows that this class is one of two in the repository that use `System.out` while `Log` is used the rest of the time. This is an inconsistency in logging with the norm of the project, indicating that the use of `System.out` is a code smell.

In 12 cases, the subclass I/O difference appears to be intentional, either due to it being a console-based program or due to the subclass's purpose indicating the intention (*e.g.* VerboseObject *vs.* Object). 6 other cases use `System.out` or `System.err` solely for reporting problems or warnings, though it may still surprise a developer aware only of the superclass implementation.

Finding 5: Whether I/O substitutability violations are problematic are often situation-dependent.
Implication: Tools intended to warn about I/O substitutability violations should be able to take into account the method pair's context.

3.4 Method Calls

This section analyzes method calls as effects, starting with individual methods in Table 10. The table show that unlike the other effects most (66%) methods have at least one call, and 30% call 3 or more different methods. The reason for considering method calls is that a method's side effects includes the side effects that occur during both direct and indirect calls. For example, a submethod calling another that throws an exception that the supermethod directly throws in its body. In this case the pair is *not* a substitutability violation due to the method call effect.

Table 10: Method calls per method (NP means non-private, and Init means constructor)

# calls	Private	NP Static	NP Init	Other
0	2.3M (15%)	2.8M (21%)	4.8M (33%)	39M (38%)
1	2.5M (16%)	3.3M (26%)	6.4M (43%)	23M (22%)
2	2.1M (14%)	2.2M (17%)	1.5M (10%)	12M (12%)
3+	8.1M (54%)	4.7M (36%)	2.2M (15%)	30M (28%)
Total	14,936,491	12,938,996	14,883,048	104,776,047

Next we briefly examine pairs with method calls in Table 11. The categories seen in this table are similar to those in Table 5, but with sets of method calls rather than exception types thrown. 18% of these pairs are cases where transitive effects of submethods can be inferred from their supermethod. However, that leaves the remaining 82% of pairs in which this is not the case. This strongly indicates that, similarly to our findings for other effects, relying on the supermethod implementation is not helpful for determining effects. In general, this table shows that method calls in sub/super pairs have a wide variety of differences leaning towards the submethod calling more and different methods.

Table 11: Method call pairs (s=sub, p=super)

$s \subset p$	524K (10%)
$s \not\subset p, \lvert s\rvert < \lvert p\rvert$	743K (15%)
$s \supset p$	1.9M (38%)
$s \not\supset p, \lvert s\rvert > \lvert p\rvert$	903K (18%)
$s = p$	398K (8%)
$s \neq p, \lvert s\rvert = \lvert p\rvert$	516K (10%)
Total Pairs	5,002,299

Table 12 considers only pairs where the submethod has more effects in at least one category than the supermethod and no effects where the supermethod has more. This table effectively lists pairs that are very likely to be substitutability violations even when side effects occurring via method calls are taken into account. Under these conditions we see a large number of method pairs, backing our results in previous sections. For example, approximately 75K sync pairs qualify, which is close to Table 7's 82K pairs.

Table 12: Pair kinds where submethod have more effects

Exception	Sync	I/O	Call	# Pairs
✗	✗	✗	✓	2,301,578
✗	✗	✓	✓	71,683
✗	✓	✗	✗	5,302
✗	✓	✗	✓	54,627
✗	✓	✓	✓	2,830
✓	✗	✗	✗	18,567
✓	✗	✗	✓	201,071
✓	✗	✓	✓	11,881
✓	✓	✗	✗	656
✓	✓	✗	✓	11,418
✓	✓	✓	✓	723
Effect Pairs				2,680,336

Finding 6: Even considering method calls, a significant number of pairs contain submethod that has more effects. This in particular indicates violations for exception and I/O effects.
Implication: While transitive calls can be important, examining only the bodies of a pair of methods is a good simplification to help find many substitutability violations.

3.5 Threats to Validity

3.5.1 External validity. As in all mining studies, representativeness and dataset quality affect the study's validity. We noted, *e.g.*, that a number of repositories included the Java Standard Library and Hadoop codebases. Since we use Boa's 2015 dataset, the study is also limited to Java projects with GitHub repositories. However, because the dataset contains over 380K Java projects, we believe this is not a problem. In our case study, we randomly selected examples and narrowed this selection down to specific types of cases. Because of the limited number of cases we were able to analyze, we cannot make broad generalizations. However, it should still bring to light some common patterns where substitutability violated.

3.5.2 Internal validity. Notwithstanding bugs, our analysis is precise in that if an effect is found in a method, then that method can have such an effect as we define it, ignoring infeasible paths. However, it is incomplete due to three reasons. First the open world assumption: we did not have access to standard or external library ASTs unless included in the project. Second, in order to scale the analysis, we do not consider inter-procedural effects from method calls and instead consider method calls as a separate effect to mitigate this. This allows us to take into account method calls without attempting to find the correct method implementation to add the call effects to the caller. Thirdly, the type of a thrown exception or object whose method is being called cannot always be determined by the analysis. For exceptions, we mark these types as `UNKNOWN` which happened very rarely. This means that side effect comparisons determined by the analysis between method pairs may be imprecise if either methods' effects are incomplete. We believe our mitigation strategies have helped to reduce these problems' impact.

4 RELATED WORK

There have been many studies on and tools introduced to manage side effects, commonly targeting exceptions and synchronization.

There have been many studies [12, 13, 20, 21, 29, 35] examining different ways developers handled exceptions. Many of the results are similar to ours. Kechagia *et al.* found 19% of a set of 4,900 crash traces in Android applications were avoidable if exceptions had been documented [20]. Another study [29] showed Java checked exceptions were often ignored. A third study [35] showed that about 20% of bug reports in a number of large projects are related to exceptions. Others [19, 27, 31, 38] examined lock usage and/or concurrency bugs. In comparison, we focus on whether substitutability is upheld and examined a much larger set of projects.

Long, Liu and Rajan introduced language features [24–26], such as effect reflection, to promote the effect analyses precision for dynamically dispatched method calls. These works assume that the effects of the subclasses and superclasses are different. The finding of our large scale study greatly supports their assumption.

To our knowledge, there have been very few studies on substitutability. Pradel *et al.* created a tool [32] to automatically test for exception and synchronization violations by random testing of method pairs. This was used to show that many widely used Java classes violate substitutability in ways that, for 30% of cases, lead to crashes. This work has previously confirmed that developers do care about exception and synchronization substitutability violations. Another work [11] examined `cxf`, a web framework, and found many problematic violations. Gordon *et al.* introduced a tool to reason about GUI usage [18]. They found developers frequently created unsafe subtypes by overriding UI-safe methods with ones that are unsafe outside of the UI thread, and that documentation on thread safety was scarce. Our study was larger scale, considered more side effects, confirms some of these previous findings, and produced some of our own. We also considered I/O and method call effects, which have not been studied by previous work.

In summary, previous work has examined various side effects or developed tools to manage them, but most do not consider substitutability and/or are small scale studies. This study in contrast examines 380K Java projects; focuses on substitutability; and includes exception, synchronization, I/O, and method call side effects.

5 CONCLUSION AND FUTURE WORK

This paper describes the first large-scale empirical study of how inheritance and side effects interact in real world Java projects. Our study is comprehensive and general. It is based on four different effect kinds and studied more than 380K projects. Our results show that a large portion of method pairs that have effects violating substitutability. We have also discussed various patterns of these pairs, and their potential impacts on supertype clients.

In the future it would be interesting to extend our study into other effects such as memory read/write [37]. We could also create IDE extensions that warn developers about effect substitutability violations, or create other tools to help manage them.

ACKNOWLEDGMENTS

The authors would like to thank Hoan Nguyen, Robert Dyer, Kathryn Stolee, and Loránd Szakács for their comments on our work. This work was supported by the US National Science Foundation under grants CCF-14-23370, CNS-15-13263, and CCF-15-18897.

REFERENCES

[1] 2018. https://github.com/MIPS/mips-src. Accessed: 03/09/2018.
[2] 2018. https://github.com/hkff/JInterfaceHack. Accessed: 03/09/2018.
[3] 2018. https://github.com/JSansalone/JFreeChart/. Accessed: 03/09/2018.
[4] 2018. https://github.com/andreagenso/Mora-Ormj/. Accessed: 03/09/2018.
[5] 2018. https://github.com/sandeep-datta/gcc/. Accessed: 03/09/2018.
[6] 2018. https://github.com/yvens47/Portfolio. Accessed: 03/09/2018.
[7] 2018. https://github.com/researchstudio-sat/ldspider4won. Accessed: 03/09/2018.
[8] 2018. https://github.com/bedatadriven/appengine-export. Accessed: 03/09/2018.
[9] 2018. https://github.com/jankotek/asterope-kotlin-prototype. Accessed: 03/09/2018.
[10] 2018. https://github.com/AntennaPod/AntennaPod/. Accessed: 03/09/2018.
[11] Diana Allam, Hervé Grall, and Jean-Claude Royer. 2013. The Substitution Principle in an Object-Oriented Framework for Web Services: From Failure to Success. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services (IIWAS '13)*. ACM, New York, NY, USA, Article 250, 10 pages. https://doi.org/10.1145/2539150.2539192
[12] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K. Roy, and Kevin A. Schneider. 2016. How Developers Use Exception Handling in Java?. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 516–519. https://doi.org/10.1145/2901739.2903500
[13] Keith Ó Dúlaigh, James F. Power, and Peter J. Clarke. 2012. Measurement of Exception-handling Code: An Exploratory Study. In *Proceedings of the 5th International Workshop on Exception Handling (WEH '12)*. IEEE Press, Piscataway, NJ, USA, 55–61. http://dl.acm.org/citation.cfm?id=2666990.2667003
[14] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844
[15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2015. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 7 (December 2015), 34 pages. https://doi.org/10.1145/2803171
[16] 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
[17] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013. 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
[18] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. JavaUI: Effects for Controlling UI Object Access. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 179–204. https://doi.org/10.1007/978-3-642-39038-8_8
[19] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What Change History Tells Us About Thread Synchronization. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 426–438. https://doi.org/10.1145/2786805.2786815
[20] 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
[21] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 484–487. https://doi.org/10.1145/2901739.2903497
[22] Gary T. Leavens and William E. Weihl. 1995. Specification and Verification of Object-oriented Programs Using Supertype Abstraction. *Acta Informatica* 32, 8 (August 1995), 705–778. https://doi.org/10.1007/BF01178658
[23] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (November 1994), 1811–1841. https://doi.org/10.1145/197320.197383
[24] Yuheng Long, Yu David Liu, and Hridesh Rajan. 2015. Intensional Effect Polymorphism. In *Proceedings of the 29th European Conference on Object-oriented Programming (ECOOP'15)*.
[25] Yuheng Long, Yu David Liu, and Hridesh Rajan. 2016. First-class Effect Reflection for Effect-guided Programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 820–837. https://doi.org/10.1145/2983990.2984037
[26] Yuheng Long and Hridesh Rajan. 2016. A Type-and-effect System for Asynchronous, Typed Events. In *International Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA, 42–53. https://doi.org/10.1145/2889443.2889446
[27] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323
[28] Daniel Marino and Todd Millstein. 2009. A Generic Type-and-effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/1481861.1481868
[29] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. 2016. Analysis of Exception Handling Patterns in Java Projects: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 500–503. https://doi.org/10.1145/2901739.2903499
[30] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*. Springer-Verlag, London, UK, UK, 114–136. http://dl.acm.org/citation.cfm?id=646005.673740
[31] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. 2015. A Large-scale Study on the Usage of Java's Concurrent Programming Constructs. *Journal of Systems and Software* 106, C (August 2015), 59–81. https://doi.org/10.1016/j.jss.2015.04.064
[32] Michael Pradel and Thomas R. Gross. 2013. Automatic Testing of Sequential and Concurrent Substitutability. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 282–291. https://doi.org/10.1109/ICSE.2013.6606574
[33] Michael Pradel, Severin Heiniger, and Thomas R. Gross. 2012. Static Detection of Brittle Parameter Typing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 265–275. https://doi.org/10.1145/2338965.2336785
[34] R. Reiter. 1987. Readings in Nonmonotonic Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter On Closed World Data Bases, 300–310. http://dl.acm.org/citation.cfm?id=42641.42663
[35] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2901739.2901757
[36] Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 199–215. https://doi.org/10.1007/978-3-540-30579-8_14
[37] Jean-Pierre Talpin and Pierre Jouvelot. 1994. The Type and Effect Discipline. *Inf. Comput.* 111, 2 (June 1994), 245–296. https://doi.org/10.1006/inco.1994.1046
[38] Rui Xin, Zhengwei Qi, Shiqiu Huang, Chengcheng Xiang, Yudi Zheng, Yin Wang, and Haibing Guan. 2013. An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, Washington, DC, USA, 100–109. https://doi.org/10.1109/ICSM.2013.21
[39] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. 2007. Dynamic Purity Analysis for Java Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 75–82. https://doi.org/10.1145/1251535.1251548