SiblingClassTestDetector: Finding Untested Sibling Functions

Qian Liang, Patrick Lam
University of Waterloo
Waterloo, Canada
{q8liang, patrick.lam}@uwaterloo.ca

***Abstract*—Methods which are not covered by a project's test suite are especially prone to exhibiting unexpected behaviours and being more challenging to maintain over time. Some methods are untested but have related implementations in sibling classes (sharing a common superclass) that are tested. Our goal is to help developers improve test suites by identifying such untested methods. We have implemented a prototype tool, SIBLINGCLASSTESTDETECTOR, which accepts programs along with their test suites, and outputs a set of Completable Candidates (CCs). We have applied our prototype tool to 17 open-source benchmarks and identified 107 CCs within these benchmarks. We have also manually produced two tests for Completable Candidates and submitted Pull Requests for these tests, one of which has been merged by developers.**

***Index Terms*—Hierarchy Analysis, Unit Testing, Test Coverage**

I. Introduction

Even well-tested software systems almost always fall short of 100% statement coverage. While testing does not guarantee desired system behaviour, and 100% statement coverage is arguably not a sensible goal, developers certainly have much less insight into the behaviour and the effects of changes on untested code as compared to tested code.

In object-oriented programming, code is organized into classes which belong to class hierarchies; for instance, *Collection* is a common ancestor of *List* and *Set*. Sometimes, the ancestor specifies a method that all subclasses must implement, such as *isEmpty()*. We have observed projects where some of the subclass method implementations are tested, but not others.

As sibling methods share the same specification, it is likely that a test case for one subclass's implementation will also work for another, possibly with minor changes (such as the identity of the class to create). The existing test implementation can serve as a template for a test for the untested method.

Our vision is to create a system which analyzes a software system, including the main code and the test suite, identifies untested methods whose siblings are tested, and automatically proposes tests for those untested methods. At this stage, we have implemented a prototype tool, SIBLINGCLASSTESTDETECTOR, which accepts programs, their test suites, and a list of untested methods (obtainable from a coverage tool such as JaCoCo); analyzes the program's class hierarchy; and outputs a set of Completable Candidates (CCs).

We have applied our prototype tool to 17 open-source benchmarks and identified 107 CCs within these benchmarks. We have also manually produced two tests for Completable Candidates and submitted Pull Requests for these tests, one of which has been merged by developers.

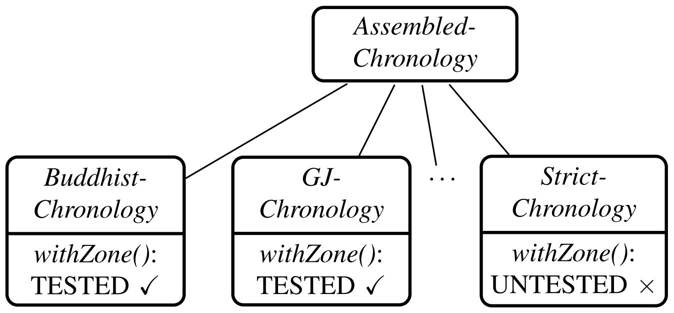


Fig. 1. *joda-time* contains superclass *AssembledChronology* and subclasses *BuddhistChronology*, *GJChronology*, *StrictChronology*, and others. Method *withZone()* is tested in *Buddhist-* and *GJChronology* but not *StrictChronology*.

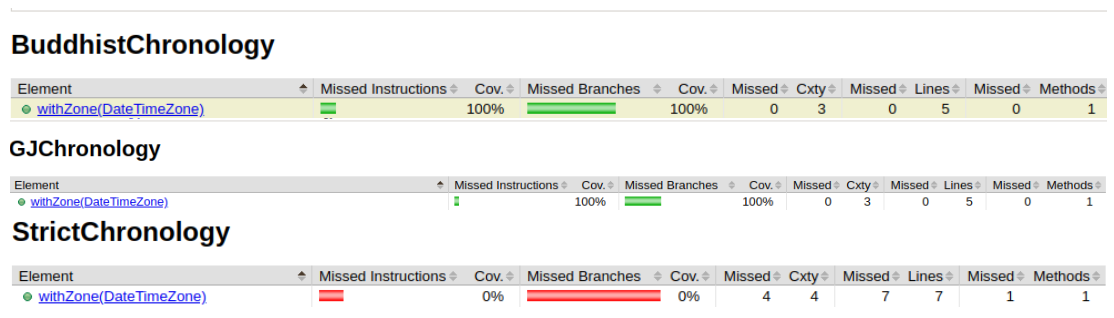
II. Motivating Example

In this section, we show how our SIBLINGCLASSTESTDETECTOR tool finds a Completable Candidate (CC) from the popular *joda-time* library (version 2.10.5). CCs are untested methods that have closely related tests.

Figure 1 depicts an example: abstract parent class *AssembledChronology* inherits a specification of method *withZone()* from its parent (not shown). Subclasses *BuddhistChronology*, *GJChronology*, *StrictChronology*, and others all implement method *withZone()*. Furthermore, *joda-time*'s test suite exercises the *withZone()* implementation in *BuddhistChronology* and *GJChronology* but not *StrictChronology*, as seen in the JaCoCo[1] reports excerpted in Figure 2. Based on this information, SIBLINGCLASSTESTDETECTOR identifies *StrictChronology.withZone()* as a Completable Candidate.

The next stage is to identify which tests exercise *withZone()* methods—a test-to-code traceability problem. In particular, we need to identify the tests that have *withZone()* as a focal method [1]—a method whose behaviour is the key point of interest in a unit test. We currently do this manually. Listing 1 shows the unit test with focal method *withZone()* from *BuddhistChronology*; a corresponding test exists for *GJChronology* which is identical except for referencing *GJChronology* in

[1]JaCoCo—Java Code Coverage Library: https://www.eclemma.org/jacoco/



BuddhistChronology

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
withZone(DateTimeZone)	🟩	100%	🟩	100%	0	3	0	5	0	1

GJChronology

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
withZone(DateTimeZone)	🟩	100%	🟩	100%	0	3	0	5	0	1

StrictChronology

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
withZone(DateTimeZone)	🟥	0%	🟥	0%	4	4	7	7	1	1

Fig. 2. JaCoCo coverage report showing that implementations of *withZone()* in *BuddhistChronology* and *GJChronology* are tested but that *StrictChronology.withZone()* is not.

place of *BuddhistChronology*. This is strong evidence that *StrictChronology* should also have a corresponding test. We currently identify the exercising *testWithZone()* test by using the `grep` tool on the source code of the test suite to find instances of the string *"withZone"*[2] We anticipate replacing this text-based search with an AST-based search for calls to *withZone()* from test methods.

```
public void testWithZone() {
  assertSame(BuddhistChronology.getInstance(TOKYO),
    BuddhistChronology.getInstance(TOKYO).withZone(TOKYO));
  assertSame(BuddhistChronology.getInstance(LONDON),
    BuddhistChronology.getInstance(TOKYO).withZone(LONDON));
  assertSame(BuddhistChronology.getInstance(PARIS),
    BuddhistChronology.getInstance(TOKYO).withZone(PARIS));
  // omitted 3 more similar asserts for space reasons
}
```

Listing 1. *withZone()* function that is implemented in multiple children classes of *AssembledChronology*, where the unit test case selected indicates the *withZone()* function covered in *BuddhistChronology*

In this example, our SIBLINGCLASSTESTDETECTOR tool parses JaCoCo XML output to obtain a list of untested methods. (JaCoCo defines an untested method to be one with 0% statement coverage.) It iterates over this list and reaches untested method *withZone()* belonging to class *StrictChronology*. This class has multiple concrete sibling classes. (In determining a class's siblings, we exclude potential siblings that are interfaces or abstract classes, as well as library classes.) Since *StrictChronology* has siblings including *GJChronology* and *BuddhistChronology*, our tool looks for sibling method implementations. This search uses the complete method signature (method name, parameter types, and return type) associated with *withZone()* to ensure unambiguity. Our tool verifies that each of the concrete sibling classes declares its own implementation of *withZone()*. (If some but not all siblings implement the method, our tool would report it as a Partial Completable Candidate, or PCC; we write (P)CC to denote a CC or a PCC). Lastly, our tool checks that at least one of these implementations is tested by ensuring that at least one implementation is not on JaCoCo's list of uncovered methods.

The eventual goal of this work is to automatically create new tests for Completable Candidates like *StrictChronology.withZone()*, based on tests for its sibling implementations.

[2]Admittedly, this example is a clean one, where *testWithZone()* tests exactly one method from the main program; it will be more difficult to identify focal methods in other cases—we plan to apply techniques from the literature there.

III. TECHNIQUE AND IMPLEMENTATION

In this section, we describe the technique that SIBLINGCLASSTESTDETECTOR implements to detect Completable Candidates (CCs), as well as how it identifies where new tests could be added. Our technique has three main stages.

1. The preprocessing stage collects functions not covered by unit tests; this pool is the source of CCs and PCCs.
2. Next, the SIBLINGCLASSTESTDETECTOR tool identifies untested methods whose siblings are tested.
3. (Ongoing) The last stage is to generate new tests for viable (P)CCs. Currently, we manually select the most promising (P)CCs and manually generate new tests. We identify potential heuristics which, when implemented, will allow our tool to locate promising tests and automatically generate sibling tests. We will manually review these tests and propose them as Pull Requests (PRs) for upstream developers.

A. Collecting Untested Functions

The first stage is a preprocessing phase, which collects data in the appropriate format for SIBLINGCLASSTESTDETECTOR to process. Our approach relies on Java projects that are built with Maven. The user must acquire the source code and is expected to sanity-check the code using `mvn clean test`, ensuring that the tests are executable. In our evaluation, we omitted projects that did not pass the sanity check, or that had any failed tests, from further consideration.

For the remaining projects, we collect coverage information. First, we add *jacoco-maven-plugin* and *maven-surefire-plugin* to their build setups. We then rerun `mvn clean test` and run the corresponding JaCoCo reporting command to generate the XML coverage report, as well as the HTML report for human visualization. Next, we run our Python script which takes the JaCoCo Coverage XML report as input and parses it using the *minidom* library. This script identifies all non-constructor methods with *missed* attribute equal to 1 (i.e. 0% statement coverage). For each identified non-constructor method m, our script stores its package name, class name, method name and method descriptor (an encoding of the method signature) in a CSV file.

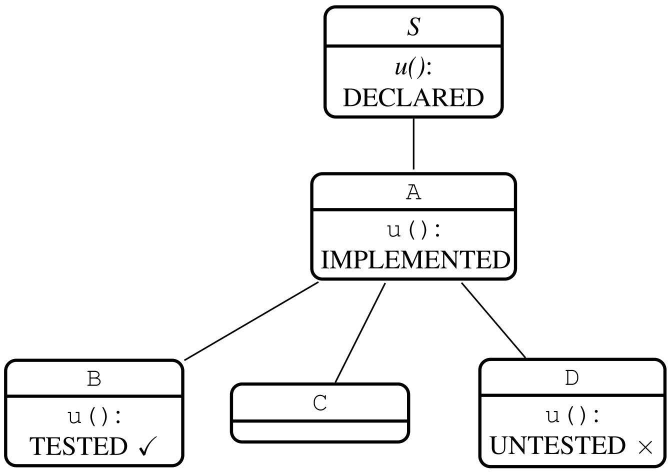


Fig. 3. A Partially Completable Candidate *D*. Superclass *S* declares abstract method *u()*. Direct subclass `A` implements `u()`. Classes {`B, C, D`} are direct subclasses of `A`. `B` has an implemented and tested `u()`, `D` has an implementated but untested `u()`, and `C` has no implementation of `u()`.

B. Computing (Partial) Completable Candidates

We implemented the SIBLINGCLASSTESTDETECTOR tool on top of the Soot [2] Java analysis framework. It takes as input the output of the preprocessing stage along with class files for the program under analysis (excluding test cases). The implementation uses the class hierarchy relationships declared in the classes under analysis to determine parents and siblings, and it assumes that it is given all relevant classes.

Let U be the list of untested methods collected from the preprocessing stage. SIBLINGCLASSTESTDETECTOR iterates through methods $u \in U$ and collects subsets of methods that satisfy properties $P1, P2, P3{-}\mathrm{CC}$ for completable candidates or $P1, P2, P3{-}\mathrm{PCC}$ for partially completable candidates. The notation $s :: u$ denotes a method with the same signature (method name, parameter types, return type) as u but instead belonging to class s.

We introduce the notion of Partial Completable Candidates (PCCs) to explore cases where some but not all siblings have an implementation of the method u (with the same signature). Figure 3 illustrates an example of a PCC where the sibling class `C` has no implementation of u. Contrast this to a CC, where all of the sibling classes implement u; for both CCs and PCCs, at least one of the implementations are tested.

We differentiate CCs and PCCs because the relative distribution of each provides information about the structure of a benchmark and perhaps about which test generation techniques might be best suited for particular candidate classes. We found that benchmarks have different proportions of CCs and PCCs.

Property 1. *The superclass* SUPERCLASS *of u's containing class C has more than one concrete, non-library direct subclasses (sibling classes).*

$$\text{let } \mathit{Siblings} := \{c \in \textsc{directSubclasses}(\textsc{SuperClass}) \mid c \neq C \ \wedge\ \textsc{isConcrete}(c)\ \wedge\ \neg\textsc{isLibrary}(c)\} \text{ in}$$

$$P1: \ \mathrm{sizeof}(\mathit{Siblings}) > 1.$$

$\textsc{isConcrete}(c)$ *holds when c is not an interface and not an abstract class.* $\textsc{isLibrary}(c)$ *holds when c does not belong to the benchmark that we are analyzing but rather one of its libraries.*

Property 2. *At least one of u's siblings is tested.*

$$P2: \exists s \in \mathit{Siblings}.\ s :: u \notin U.$$

Property 3. *Each sibling class s (for CC) or at least one sibling class s (for PCC) implements a method with the same signature as u.*

$$P3{-}\mathrm{CC}: \ \forall s \in \mathit{Siblings}.\ \textsc{implementsMethod}(s, s :: u);$$

$$P3{-}\mathrm{PCC}: \ \exists s \in \mathit{Siblings}.\ \textsc{implementsMethod}(s, s :: u).$$

The predicate IMPLEMENTSMETHOD *means that class s directly implements method m.*

The predicates DIRECTSUBCLASSES, ISCONCRETE, ISLIBRARY, and IMPLEMENTSMETHODS are provided by Soot.

C. Finding Promising Candidates & Generating Tests

We have identified two types of (P)CCs as promising candidates, described below. We are working on techniques for automatically finding them and generating tests.

1) Candidates with Corresponding Test Classes: Let method m be implemented in all sibling classes $\{B, C, D\}$. Let *B.m()* be tested. A promising candidate with corresponding test classes has a *BTest* class with test t which directly invokes focal method *B.m()*. For this type of promising candidate, t provides a potential template for generating sibling tests to cover the untested *C.m()* and *D.m()* methods.

Our current approach for locating candidates with corresponding test classes follows. The crux is locating tests with a given focal method. Our current approximation iterates on all test classes *ZTest*. We treat *ZTest* as a test for class *Z*, removing the "Test" suffix from its name. If we find focal method *B.m()* in *BTest* with sibling implementation *C.m()* that is a CC or a PCC, then we have found a corresponding-test-case promising candidate and generate a test for *C.m()* from *BTest*.

To generate tests, we would first (if needed) create the *CTest* and *DTest* classes, and then clone test case t from *BTest* to *CTest* and *DTest*, modifying declared types of variables and static class references, and otherwise preserving the method bodies and exception declarations. In the cases that we have looked at, this approach has worked, but it is of course subject to differences between B, C, and D, such as different methods or fields being present on the different siblings.

Because EvoSuite [3] integrates well with JaCoCo inside the Maven/Surefire configuration, we plan to first use EvoSuite (via commandline) with classes C and D as parameters to generate the corresponding *CTest* and *DTest*. We will then use Soot to collect the necessary information from test case t in *BTest*, and apply it to the generated test cases covering *C.m()* and *D.m()*, perhaps using the Comby tool[3].

2) Candidates with Focal Method Called via Superclass: Again, let m be a method in all of the sibling classes $\{B, C, D\}$, and let *B.m()* be tested. This time, $\{B, C, D\}$ have a common superclass S. For this type of promising candidate, a test case invokes *B.m()* as a focal method through dynamic dispatch using a receiver object of declared type S (i.e. *s.m()*). No test case invokes *C.m()* or *D.m()*.

Until now, we have only used program queries that are directly answerable from benchmark Java class files (e.g. information about sibling classes). However, to find this type of candidate, we need more sophisticated call graph information; for Java, this is tightly connected to pointer analysis.

To find called-via-superclass candidates, given that *B.m()* is a PCC or a CC, we look for edges in the call graph from some test case t that reaches method *B.m()*, where the call site in t has declared type S for the receiver object (*s.m()*). A call graph based on Class Hierarchy Analysis would show calls to *C.m()* and *D.m()* at *s.m()*. An object-sensitive approach [4] would potentially be able to construct a call graph guaranteeing that no D object reaches the *s.m()* callsite. We plan to manually investigate a number of called-via-superclass candidates before proposing a strategy for automatic test generation.

IV. Evaluation

We investigated the efficacy of our approach by applying it to a convenience sample of 17 open source Java benchmarks. Many of these benchmarks have a high number of GitHub stars, showing popularity; build successfully with Maven and have no failing tests; and have test suites with a range of code coverage percentages. Table I presents our results, including the numbers of completable condidates and partial completable condidates, as well as the number of untested non-constructor methods (0% statement coverage) and tested non-constructor methods in each benchmark, and the percentage of untested methods. Note that the benchmarks with the smallest proportions of untested methods, like *classmate*, *commons-lang3*, and *fastjson*, also had no (P)CCs, while Soot and FindBugs had the highest proportions of untested methods and many CCs and particularly PCCs. In all, 11 of the 17 benchmarks had some (P)CCs. This suggests that (P)CCs have promise for extending coverage of lightly-tested systems.

We had a closer look at *joda-time* and *soot*, representing low-coverage and high-coverage extremes. For *joda-time*, which had higher coverage (i.e. not many untested methods), the untested methods that do exist tend to cluster around a small number of classes. That is, many of the (P)CC methods share the same superclass. For instance, *org.joda.time.field.DecoratedDateTimeField* is the direct superclass for 3 out of the 18 CCs, and 8 out of the 11 PCCs. Such clustering of untested methods suggest that *joda-time* may be an especially suitable candidate for test generation. In any case, whether the (P)CCs in *joda-time* are intentionally untested needs to be investigated.

At first glance, *soot* appears to be poorly tested. However, more than half of the untested methods are from generated code not checked into Soot's git repository: Soot includes a generated Java parser. Package soot.JastAddJ is generated, with a total of 10,629 untested methods, accounting for $> 40\%$ of the untested methods. Hence, using the methods-covered metric indiscriminately leads to unwarranted conclusions. Excluding these methods, *soot* would be in the middle of the pack. The PCCs in *soot* arise because many of *soot*'s method

[3]Comby—Structural code search and replace for every language: https://comby.dev/

TABLE I
Counts of untested and tested non-constructor methods, percentage untested methods, and counts of Completable Candidates and Partial Completable Candidates among our 17 open-source benchmarks.

Benchmark	Version	# Untested Methods	# Tested Methods	% Untested Methods	# CCs	# Partial CCs
classmate	1.5.1	8	292	2.7	0	0
commons-collections	4.3	331	2336	12.4	4	4
commons-math	3.6.1	767	5276	12.7	3	4
commons-lang3	3.9	104	2561	3.9	0	0
fastjson	1.2.62	112	1448	7.2	0	0
findbugs	3.0.1	6976	1132	86.0	11	59
gson-parent	2.8.5	64	411	13.5	0	0
javacc	7.0.5	1787	582	75.4	9	0
jgrapht-core	1.3.1	329	1558	17.4	1	2
joda-time	2.10.5	241	2602	8.5	18	11
jsoup	1.10.1	114	604	15.9	0	0
ph-commons/ph-commons	9.3.9	2682	3731	41.8	8	16
plexus-utils	3.3.0	476	545	46.6	0	0
quartz-core	2.3.1	939	1298	42.0	3	2
soot	4.0.0	23516	2998	88.7	40	744
velocity-engine-core	2.1	477	1146	29.4	3	3
woodstox-core	6.2.0	551	1628	25.3	7	4
Total		39474	30148		107	849

implementations exist at least one level higher in the class hierarchy than the sibling classes where the untested method is located. Static analysis framework *findbugs* also exhibits this pattern—perhaps the pattern occurs often in this domain.

To evaluate the overall potential utility of (P)CCs, we manually generated tests for uncovered sibling methods in two benchmarks, *soot* and *ph-commons/ph-commons*, and submitted Pull Requests (PRs) for each. The PR submitted to *ph-commons/ph-commons* was quickly merged by the project authors, while the PR for *soot* still requires additional work to meet unrelated project requirements.

Potential Impacts and Future Work: We anticipate that our sibling test detection and generation approach can contribute to developer productivity. Most importantly, it can free developers from the burden of creating highly similar tests for related classes, while ensuring that these classes continue to be meaningfully tested with at least somewhat appropriate tests. Our tool can be used both when developers initially create new classes, as well as during routine maintenance—when a developer touches an untested class, they may benefit from generating related tests for it. Once we have created machinery for test generation, we will apply it and poll developers about the usefulness of generated tests. Finally, we intend to release our tool as open-source software.

V. Related Work

We discuss related work in the areas of code clones, refactoring tools, and test generation.

a) Code Clones: There is a broad consensus that code clones account for a significant amount of code, perhaps 5–10% of codebases [5]; Kapser and Godfrey investigate clones in the Apache web server [6], while Casazza et al identify clones in the Linux kernel [7]. With the increasing acceptance of unit tests, our work helps developers mitigate issues with clone maintenance by helping them ensure that clones are at least tested. Our approach does not detect clones; it simply combines information from the class hierarchy (i.e. which methods are implemented) with test coverage data to identify methods that are easily testable by adapting existing tests.

b) Refactoring: At the moment, our SiblingClassTestDetector tool is simply a detector—it recommends methods that could easily be testable, presumably with tests that are similar to already-existing ones. The easiest way to extend coverage to such methods is to clone existing tests. Such clones could then be reasonable candidates for refactoring. Meszaros [8] has written extensively about refactoring in the context of unit tests. Thus, an alternative to cloning tests would be to refactor them and to use the refactored tests for all of the sibling method implementations.

Due to its design, SiblingClassTestDetector may identify refactorable cloned methods in the main program (not tests). This is because it locates sibling method implementations which share the same specification. Such implementations may be at risk of inconsistent changes; Juergens et al [9] found that inconsistent updates to clones are frequent, and that such updates cause higher-than-usual rates of software faults.

Sibling methods may also be refactoring candidates. Clone detection has been extensively studied, surveyed by Roy et al. [10]. Identifying potential clones is a side-effect of this work; our goal is to identify methods that can easily be tested.

c) Test Generation: While the high-level goal is similar—to improve the quality of test suites—our work differs in focus from automatic test generation [11]. Malburg and Fraser's EvoSuite [3] uses a modified search-based approach over the space of test cases to identify potential new tests, and constraint solving to generate tests. The JQF approach by Padhye et al [12] combines fuzz testing and property-based testing to generate test cases. The FrUITeR framework [13] can be seen as a way of generating UI tests for target mobile apps from source apps, which is analogous to our approach of generating unit tests from those of sibling classes. We use domain knowledge about OO system design to find methods that are likely to be easily testable; the insight is that sibling methods should be testable using similar tests. We believe that information that is incidentally provided by developers can inspire the creation of useful tests, and we hope that the research community will follow.

VI. Conclusions

Based on our observations about OO systems and sibling classes, we presented an approach for identifying untested methods that are suitable for adaptation of existing tests from their siblings. We identified three conditions for methods to be (Partially) Completable Candidates and implemented them in our prototype SiblingClassTestDetector tool, which uses the output from the JaCoCo coverage measurement tool as well as the Soot program analysis framework. Our tool detected (P)CCs in 11 of 17 open-source Java benchmarks.

References

[1] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *SCAM*, 2015.

[2] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot—a Java bytecode optimization framework," in *CASCON*, 1999.

[3] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *ASE*, 2011, pp. 436–439.

[4] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *POPL*, 2011.

[5] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *RE*, 1995, pp. 86–95.

[6] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems: A case study," *J. Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[7] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Identifying clones in the Linux kernel," in *SCAM*, 2001, pp. 90–97.

[8] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE*, 2009, pp. 485–495.

[10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[11] G. Candea and P. Godefroid, *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*. Cham: Springer International Publishing, 2019, pp. 505–531.

[12] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," in *ISSTA*, 2019.

[13] Y. Zhao, J. Chen, A. Sejfia, M. S. Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "FrUITeR—a framework for evaluating ui test reuse," in *ESEC/FSE*, Nov 2020.