Increasing the Effectiveness of Automatically Generated Tests by Improving Class Observability

Geraldine Galindo-Gutierrez[†], Juan Pablo Sandoval Alcocer[*], Nicolas Jimenez-Fuentes[*]
Alexandre Bergel,[‡], Gordon Fraser[§]
[*]Department of Computer Science, School of Engineering, Faculty of Engineering,
Pontificia Universidad Católica de Chile, Chile
[†]Centro de Investigación en Ciencias Exactas e Ingenierías, Universidad Católica Boliviana, Bolivia
[‡]RelationalAI, Switzerland
[§]University of Passau, Germany

***Abstract*—Automated unit test generation consists of two complementary challenges: Finding sequences of API calls that exercise the code of a class under test, and finding assertion statements that validate the behavior of the class during execution. The former challenge is often addressed using meta-heuristic search algorithms optimising tests for code coverage, which are then annotated with regression assertions to address the latter challenge, i.e., assertions that capture the states observed during test generation. While the resulting tests tend to achieve high coverage, their fault finding potential is often inhibited by poor or difficult observability of the codebase. That is, relevant attributes and properties may either not be exposed adequately at all, or only in ways that the test generator is unable to handle. In this paper, we investigate the influence of observability in the context of the EvoSuite search-based Java test generator, which we extend in two complementary ways to study and improve observability: First, we apply a transformation to code under test to expose encapsulated attributes to the test generator; second, we address EvoSuite's limited capability of asserting the state of complex objects. Our evaluation demonstrates that together these observability improvements lead to significantly increased mutation scores, underscoring the importance of considering the class observability in the test generation process.**

***Index Terms*—Observability, Automatic Test Generation, Mutation Analysis**

I. Introduction

Testing is a key aspect of the software development cycle, but it is also considered one of the most time-consuming tasks for developers [1], [2]. To support developers in writing unit tests to verify the behavior of their code, research has therefore proposed different approaches and tools to automatically generate unit tests [3]–[5]. These tools produce tests consisting of sequences of statements that satisfy predefined testing goals, such as maximizing the code coverage. While studies have demonstrated that automatically generated tests succeed in achieving high code coverage, they also showed that these tests are limited in their effectiveness at revealing faults [6]–[10]. One factor influencing this is a poor choice of test assertions, which often fail to establish a relationship between the methods being tested and what they check [11].

An important reason contributing to this issue is the observability of the class under test, i.e., the ability to access (and check) the internal state of an object. For example, consider the `DbConnectionBroker` class (Listing 1), which is responsible for handling database connections in the *biblestudy* open source project. When testing the `release` method, a test generator will struggle to create an appropriate assertion as the method neither has a return value to check, nor is the private `maxConnections` attribute publicly accessible for use in an assertion. A recent study [11] revealed that in most cases (69%) when automatically generated tests fail to produce appropriate assertions, the class under test lacks methods to verify the effects of the behavior being tested.

Even if the class under test allows inspecting the internal state, test generators may be unable to make use of that infor-

```
public class DbConnectionBroker{
  private int maxConnections = 0;
  //...
  public void release(DbConnectionAttributes attrs) {
    if (attrs.getIndex() >= 0 &&
      this.attributesArray[attrs.getIndex()] != null) {
        if (attrs.getIndex() < this.min) {
          this.attributesArray[attrs.getIndex()].free();
          --this.maxConnections;
        } else {
          this.disconnect(attrs);
        }
    } else {
        this.disconnect(attrs);
    }
  }
}
```

Listing 1. The DbConnectionBroker class does not allow verifying if the `release` method works as expected.

```
public class HtmlViewerPanel extends JPanel {
  private URL _currentURL;
  ...
  public synchronized void goForward(){
    if (_historyIndex > -1 &&
        _historyIndex < _history.size() - 1) {
      displayURL(_history.get(++_historyIndex));
    }
  }
  private void displayURL(URL URL) {
    ...
    _currentURL = url;
    ...
  }
  public URL getURL() {
    return _currentURL;
  }
}
```

Listing 2. The internal state of HtmlViewerPanel objects is accessible but current test generators cannot make use of that information.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00156

mation. For example, consider the `HtmlViewerPanel` class in Listing 2, taken from the open source project *squirrel-sql*: The method `goForwards` moves the content of an HTML viewer forward to the next URL, which is stored in the attribute `_currentURL`. While the method again does not provide a return value on which an assertion can be generated, in this example there actually is a means to access the internal state, as the `getURL` method allows retrieving the currently stored URL. However, the popular EvoSuite unit test generator [3], like other tools, will nevertheless fail to produce an appropriate assertion as it only considers getter functions that return primitive datatypes (e.g., integers or Booleans).

Without appropriate assertions, tests may miss faults [12]. The urgency of this problem has been demonstrated by a recent study [13] that found that low observability led to inconsistent results in EvoSuite, as measured by the mutation score (i.e., a proxy metric estimating the fault finding potential of tests). Additionally, although various studies have shown that observability is an aspect that makes test creation difficult [14], [15], little is known about the extent to which limited observability negatively impacts test generation algorithms. In this paper, we therefore aim to assess the impact of class observability on the effectiveness of EvoSuite's automatically generated tests; the central research question consequently is: *To what extent does the observability of the class under test impact the effectiveness of the generated tests?*

In order to answer this research question, we designed a study where we automatically increased class observability through two EvoSuite extensions addressing two types of observability problems. The first extension increases the visibility of class attributes by automatically injecting public accessors into private and protected attributes. The second extension improves EvoSuite's ability to inspect the states of composed objects (i.e., non-primitive types) by generating assertions that recursively check their states. We then conducted an extensive experiment in which we generated tests for a well-known set of 100 complex classes from the SF-110 dataset [16]–[18] using EvoSuite, both with and without our extensions, separately and together. This study seeks to contribute to the literature on how test-generation tools can produce more effective tests by increasing class observability.

In detail, this paper makes the following contributions:

- We propose an observability transformation that exposes internal object states to automated test generators.
- We propose an extension of existing assertion generation techniques for asserting on states of complex objects.
- We empirically evaluate the effects of observability on a sample of 100 open source Java classes.

Our study reveals that increasing the visibility of class attributes alone significantly increased the mutation scores of the generated tests for 26 out of the 100 classes. When we both increased visibility and enabled EvoSuite to recursively assert composed objects, the mutation score improved significantly for 26 classes. These results demonstrate that observability has a substantial and significant impact on the effectiveness of assert generation and the mutation score. However, we recommend further research on techniques that improve the observability and testability of classes to determine when and how to apply such strategies effectively.

II. Observability Challenges in Test Generation

Software testability, defined as the ease of testing a software artifact, plays an important role for testing. High testability simplifies test creation, whereas low testability can significantly increase the time and effort required in the development of unit tests [19]–[21]. Various factors have been shown to induce negative testability, such as controllability [22], complexity [23], cohesion/coupling [24], understandability [25], and inheritance [26], with observability being particularly important and subject of extensive research [14], [27].

Observability is the ability to monitor a program's behavior through its outputs [28], therefore, it represents an important aspect in verifying correct input processing [14], [29]. In classes with limited observability, creating unit tests and especially their assertions, becomes a challenge specifically for test generation tools. A previous study showed that limited observability of class attributes may lead to inconsistencies in automatically generated tests, particularly inconsistencies between the test target goals and the generated assertions [11]. For example, resulting assertions may not be related to the intended behavior being tested. In the following paragraphs, we describe two root causes of these inconsistencies.

A. Limited Object State Visibility

Listing 1 showed that if the class under test lacks mechanisms to verify that an object reaches the expected state, it becomes difficult to create effective assertions, even manually. This issue arises because, without access to the internal state, it is challenging to confirm whether the methods have performed as intended. For instance, consider the `ScriptOrFnScope` class shown in Listing 3, and the two resulting tests automatically generated by EvoSuite for it shown in Listing 4.

```
class ScriptOrFnScope {
  private int braceNesting;
  private ScriptOrFnScope parentScope;
  private ArrayList subScopes;
  private Hashtable identifiers = new Hashtable();
  private Hashtable hints = new Hashtable();
  private boolean markedForMunging = true;
  private int varcount = 0;
  ...
  void preventMunging() {
    if (parentScope != null) {
      markedForMunging = false;
    }
  }
  void munge() {
    if (!markedForMunging) { return; }
    ...
    for (int i = 0; i < subScopes.size(); i++) {
      ScriptOrFnScope scope =
        (ScriptOrFnScope) subScopes.get(i);
      scope.munge();
    }
  }
}
```

Listing 3. Excerpt of ScriptOrFnScope class.

```
public void test09()  throws Throwable  {
  ScriptOrFnScope scope0 = new ScriptOrFnScope((-870), (
       ScriptOrFnScope) null);
  ScriptOrFnScope scope1 = new ScriptOrFnScope((-870),
       scope0);
  scriptOrFnScope1.preventMunging();
  scriptOrFnScope1.munge();
  assertFalse(scope0.equals((Object)scope1));
}
public void test10()  throws Throwable  {
  ScriptOrFnScope scope0 = new ScriptOrFnScope((-870), (
       ScriptOrFnScope) null);
  scope0.preventMunging();
}
```

Listing 4. Generated tests for ScriptOrFnScope class.

EvoSuite generates regression assertions for a given test by tracing its execution, and then elaborating all possible assertions using basic heuristics. For example, for each primitive return value it will add an assertion that compares the return value to the one observed during the execution, and for all non-primitive objects contained in the test it will create assertions comparing the values returned by all getter functions with the observed values. Thereby, a getter is simply identified as a parameterless method that returns a primitive value and has no side-effects. In a second step, EvoSuite will then iteratively insert mutations into the code under test and filter out those assertions that can successfully reveal a difference between the executions on the original code and the mutated code. The result is a minimized set of assertions supposed to check only attributes relevant to the execution [30].

As the `ScriptOrFnScope` class provides no public accessors or other means to check the state of variables used by the methods under test, specifically the `preventMunging` and `munge` attributes, none of the assertions generated using this approach can reveal a difference in behavior induced by the mutants. This may result in a test without any assertions at all, like `test10`. To avoid filtering *all* assertions, EvoSuite implements some heuristics that try to select arbitrary but plausible assertions, such as assertions at the end of the test case like the assertion in `test09`. This assertion attempts to compare the states of two objects, but this comparison is ineffective because the class `ScriptOrFnScope` does not override the `equals` method. Consequently, the `equals` method only checks if the two variables reference the same object, which is not the case. As a result, unless an exception is thrown during execution, the test will always pass independently of the presence of faults, and it provides no meaningful validation.

B. Asserting Composed Object States

Even when a class contains methods that provide visibility of its internal state, the complexity escalates when creating assertions for non-primitive objects, which are objects that have other objects as attributes. Currently, EvoSuite generates five types of assertions: primitive assertions, comparison assertions, inspector assertions, field assertions, and string assertions. Some of these assertions, such as inspector and field assertions, help to assert the final state of an object after test execution. For instance, they can inspect a private primitive attribute through an inspector method (e.g., an accessor) or directly access the primitive field if it is public.

However, these types of assertions do not check the state of non-primitive objects (other than comparing against `null`), such as objects that have other objects as attributes. As a result, it is not possible to detect bugs related to the attributes of inner objects. For example, consider the class `HtmlViewerPanel` in Listing 2 and its method `goForwards`, which helps users navigate in the HTML viewer by moving forward to the next URL. During the mutation testing step of assertion generation, EvoSuite injects bugs into the if condition within the method `goForwards`. When the if condition is met, the `URL` object is updated through the method `displayURL`. Although the class has a public accessor for the attribute `_currentURL`, EvoSuite's assertions are unable to validate if the `URL` to which the HTML viewer moves forward is correct. To do so would require checking if the state of the `URL` object is correct. However, since the URL object is an attribute of `HtmlViewerPanel`, it is necessary to assert the state of the inner objects.

C. Motivation

Automatically generating tests for classes with restricted observability presents a formidable challenge. Most test generation tools focus on achieving high code coverage, often overlooking the testability of the class under test. Especially observability greatly influences the formulation of effective assertions and is directly reflected in the (low) mutation scores. Our examples illustrate that even during manual test creation it may be challenging to generate feasible assertions in some scenarios. Moreover, recent research highlights a disconnection between the assertions and the branches targeted in tests due to limited observability [11]. Our research is motivated by the goal of understanding how observability impacts the effectiveness of generated tests and to what extent. Therefore, we hypothesized that increasing the observability of the class under tests can lead to a better performance of the generation tool, mainly on mutation analysis.

III. Studying and Increasing Observability

To assess the impact of class observability on EvoSuite's effectiveness, we implemented two complementary extensions. This section provides a detailed description of each extension.

A. Open Object State Visibility Extension

This extension increases the observability of the class under test by automatically injecting public accessors for all private or protected attributes that do not already have an accessor. We inject these accessors at the bytecode level. The injected accessors simply put the attribute on the stack and include a return instruction. The method names of the accessors are a concatenation of the string `get`, the attribute name, and the keyword `_Injected` at the end. This keyword helps us to differentiate between the accessors that already exist and those that we injected. To detect whether an attribute already has an accessor, we search its class for a public method containing only one statement that returns the attribute.

When this extension is activated, EvoSuite runs a preprocessing algorithm before starting the generation. This algorithm iterates over all classes in the system and injects public accessors. Our goal is to open object state visibility, allowing EvoSuite direct access to all attributes in the system. Since these accessors are part of the bytecode, EvoSuite would consider them when computing target goals or during mutation testing. We modify EvoSuite so it considers these injected accessors only during the assertion generation algorithm, which is a post-processing step. However, the injected accessors are not considered for mutation injection or as part of the coverage goals. Listing 5 shows a generated test for the class `ScriptOrFnScope` using this extension. The generated test uses an injected getter `getMarkedForMunging_Injected` in the assertion generation (in bold). This helps to increase the mutation score of the class `ScriptOrFnScope` by 5.40%.

```
public void test10()  throws Throwable  {
  ScriptOrFnScope scope0 = new ScriptOrFnScope(-870, (
      ScriptOrFnScope) null);
  ScriptOrFnScope scope1 = new ScriptOrFnScope(-870,
      scope0);
  scope1.preventMunging();
  scope1.munge();
  assertFalse(scope1.getMarkedForMunging_Injected());
  assertNotSame(scope1, scope0);
}
```

Listing 5. Example of generated test that use an injected getter (in bold).

Note that this EvoSuite extension is not intended as a solution for the observability problem, but as a means to assess its impact. Encapsulation is important in object-oriented programming, as it reduces dependencies on internal object data. We further discuss this point in Section VI.

B. Recursive Inspector Assertion Extension

This extension adds a new assertion type to EvoSuite called a *recursive inspector assertion*. It builds on the existing Inspector Assertion in EvoSuite. Inspector assertions create an assertion for each inspector method found in the object under analysis. EvoSuite considers an inspector method as one that takes no parameters, has no side effects, and returns a primitive data type. This excludes methods that return an object. In our case, we extend this functionality to also consider inspector methods that return an object. We focus on methods that take no parameters, and consist only of functionality of returning a class attribute that is not primitive.

Our extension generates assertions that contain a chain of inspector methods that are executed over the object under analysis. The recursion stops either when an inspector method is found that returns a primitive value, for which an assertion is added, or when a configurable threshold of the recursion depth is reached. The algorithm inspects all attributes that have a public accessor, and if the attribute is an object the algorithm will inspect their attributes too, recursively. Listing 6 shows a test for the `HtmlViewerPanel` class (Listing 2) that uses this extension. Note that the generated test has three recursive inspector assertions. For instance, the second assertion evaluates the URL protocol using two accessors `getURL` and `getProtocol` in a chain.

```
public void test15()  throws Throwable  {
  ...
  HtmlViewerPanel panel0 = new HtmlViewerPanel(uRL1);
  panel0.gotoURL(uRL0);
  panel0.goBack();
  panel0.goForward();
  assertEquals("file://some/fake/but/wellformed/url",
      panel0.getURL().toExternalForm());
  assertEquals("file", panel0.getURL().getProtocol());
  assertEquals("some", panel0.getURL().getHost());
}
```

Listing 6. Example of recursive inspector assertions (in bold)

Note that after assertion generation, EvoSuite performs a mutation-based assertion minimization, which removes assertions not relevant to the mutation testing analysis. As a consequence, EvoSuite will only keep recursive inspector assertions that are considered useful for detecting mutants.

IV. Experimental Setup

This section details the empirical study for evaluating the impact of class observability on EvoSuite effectiveness.

A. Research Questions

Our main research question is: *To what extent does the observability of the class under test impact the effectiveness of the generated tests?* As we have shown in Section II, the limited access to private/protected class attributes and mutants that affect composed objects can hinder test generation algorithms. To assess the impact of these limitations, we aim to answer the following research questions empirically:

- **RQ1 Encapsulation**: *To what extent does the limited access to private/protected attributes in the system impact EvoSuite's effectiveness?*
- **RQ2 Composed Objects**: *To what extent does checking the state of composed objects impact EvoSuite's effectiveness?*
- **RQ3 Combined Effectiveness**: *To what extent does the combination of the two EvoSuite extensions impact EvoSuite's effectiveness?*

RQ1 investigates how encapsulation affects EvoSuite's effectiveness. To address this question, we compare EvoSuite's performance, in terms of code coverage and mutation score, before and after modifying the visibility of all class attributes in the system. **RQ2** and **RQ3** examine the impact of the ability to inspect the state of composed objects on EvoSuite's effectiveness. Specifically, these questions evaluate recursive object assertions with and without restricted access to non-public attributes. **RQ3** specifically aims to determine the extent to which encapsulation limits our proposed recursive assertion generation technique.

B. Subject Java Classes

Our evaluation uses 100 classes from the SF110 dataset, widely used in unit test-generating tool evaluations [31]–[33]. The complete dataset contains over 23K classes from 110 projects. However, studies have shown that the dataset contains many trivial classes [34]. Therefore, we decided to use a subset of classes previously used in other studies [11], [18], [35]. This

subset was filtered using a cyclomatic complexity threshold of 3 to ensure that the generated tests are not solely composed of accessors. This filtering guarantees a wider variety of explored branches and generated mutants while keeping the time required to run our experiments manageable.

C. Baseline for Comparison

As in previous studies, we use the DynaMOSA algorithm as a baseline for comparison [11], [18], [36]. DynaMOSA uses a multi-objective technique known as the Many-Objective Sorting Algorithm (MOSA) with dynamic selection; this technique has proven to generate tests with higher coverage. We generate unit tests for our subjects following the recommended hyperparameters of previous studies. We also extended the time of the search budget to 180 seconds and the assertion timeout to 600 seconds, as suggested by Panichella et al. [32].

D. Treatments

To answer our research questions, we compare the baseline results with three treatments based on the observability extensions described in Section III.

1) *Injected Getters*: The first treatment involves a preprocessing step before generating tests, which injects public accessors for attributes that are protected or private.
2) *Recursive Assertions*: The second treatment allows EvoSuite to generate assertions that recursively check the state of composed objects, using only accessors already defined in the original source code.
3) *Combined Extensions*: The third treatment combines both extensions, injecting public accessors, which can later be used for generating recursive inspector assertions.

Note that in all cases, the injected code is not considered when computing the test goals and mutation score. Our goal is to generate tests with the same test goals and analyze how many additional mutants the treatments help to kill using the same set of mutants from the baseline. For recursive assertions in the second and third treatments, we use a threshold of $n = 3$. This means that the generated assertion checks recursively all the primitive attributes of the composed object up to a maximum depth of three levels.

E. Experimental Protocol

To answer our research questions, we compare a baseline of generated tests with the treatments previously defined. We consider the following aspects of the conducted experiment for comparing test suites generated for the same class.

1) Collected Metrics: For coverage metrics, we compare the *statement, branch, and resulting test suite's coverage*, with the latter being a summary of the eight coverage metrics considered during EvoSuite's generation. By construction, our EvoSuite extensions should not impact coverage, but we compute these metrics only to support this claim. However, our primary focus is on mutation analysis metrics. We analyze the *mutation score* (i.e., the percentage of mutants detected by a test suite) to compare if the use of a treatment yields a higher percentage of mutants detected. To determine if a detected mutant is related to the extensions used, we manually examine the details of *killed mutants*. For mutation analysis, we use the EvoSuite mutation engine and their default mutation operators (delete call, delete field, insert unary operator, replace arithmetic operator, replace bitwise operator, replace comparison operator, replace constant, and replace variable) [30], [37], [38], this allowed us to control the use of our extensions for mutation analysis in a single workflow. In total, all classes under analysis contain 26,177 mutants, varying between 3 and 2,944 for each class.

2) Data Collection: We generated tests for each of the treatments and the baseline using 30 randomly generated 13-digit seeds to ensure reliable detection of statistical differences. In consequence, we had to generate tests for each class using four different configurations 30 times each, making a total of 120 generated test suites for each class and 12 thousand test suites overall. We stored all the data outputs and metrics in text files for each configuration and statistically analyzed them. Each treatment generation of 30 seeds per class took around a day and a half to complete. All experiments were executed on a computer with an M1 processor with 20 cores and 64 GB RAM. In total, the experiment took 6 days of computation time.

3) Statistical Analysis: After applying all treatments to the studied subjects, we compared the generated tests from each treatment against the baseline generated tests in terms of coverage and mutation score. As the Shapiro-Wilk test suggested that the data is not normal, we used a one-tailed Wilcoxon test [39] to identify statistically significant differences between the baseline and the treatments. Specifically, we compared the 30 mutation scores and coverage metrics of the 30 generated test suites for each class, with each test suite generated using a different seed. We aim to conclude if the collected metrics of the treatments were significantly higher than the baseline, using a threshold value of 0.05. After comparison, for each class that shows a statistically significant difference, we manually analyze the source code of the generated tests to confirm if they use an injected accessor or a recursive inspection assertion, and if the mutation score increment is directly related to our EvoSuite extensions.

Finally, we measure the impact of each treatment using the Vargha-Delaney ($\hat{A}_{12}$) statistic [40] to gauge the effect size. We interpret the $\hat{A}_{12}$ statistic, taking as reference the value 0.5. If the value is 0.5, we conclude that the treatment used does not affect the generated tests. However, a value greater than 0.50 suggests that in further generations, the use of the strategy has a greater probability of obtaining better metrics. Consequently, a value lower than 0.50 indicates the opposite.

F. Threats to Validity

Threats to *construct validity* concern the relationship between theory and experimentation. The evaluation of the proposed extensions is based on effectiveness metrics commonly used in the literature: coverage metrics and mutation score [32], [33], [41], [42]. Coverage metrics represent the quantity of code executed by the tests and mutation score

creates artificial bugs and measures how many are detected by the test suite. Although these artificial bugs do not necessarily represent all possible bugs in the code, mutation score is considered as a reasonable estimation of effectiveness to evaluate a test suite [43], [44].

Threats to *internal validity* arise from factors that might influence the obtained results. To reduce the randomness and to ensure the sufficiency of data for statistical tests, we generated tests 30 times per class using different seeds. For generation, we used parameters suggested in previous studies that have proven to generate tests with higher coverage. Another threat comes from the additions caused by our extensions. When using injected getters, we modified EvoSuite to ignore the injected methods in the mutation analysis (i.e., to not create mutants for these methods) and from the generation goals. This avoids variations in the number of mutations or tests. In consequence, the results in each treatment are comparable as they are analyzed using the same set of mutants.

Threats to *external validity* are related to the generalization of our results. Because of the high number of test generations needed for the experiment, we limited the studied classes to 100 Java classes from the SF-110 dataset. Although reduced, this dataset contains classes from a variety of 32 projects and several domains. The use of these 100 classes is also comparable with previous studies that use the same selection to study EvoSuite improvements and test quality [11], [18]. While the *injected getters* implementation is applicable to different test generation tools, the *recursive assertions* implementation aligns with the functionality and limitations of EvoSuite. Asserting the state of composed objects remains a relatively unexplored field in test generation, which we aim to expand on in future work.

Threats to *conclusion validity* arise from the relationship between treatment and results. For analyzing the effectiveness of our extensions using the mutation score of the generated test suites, we use appropriate statistical tests and 30 repetitions to ensure enough data to use these tests. For each comparison, we performed a Wilcoxon test using $alpha = 0.05$ and used the Vargha-Delaney effect size. We concluded only from statistically significant results.

V. Results

This section summarizes the results after generating tests for the selected subjects using the three defined treatments and the baseline 30 times using different seeds. A replication package containing the results in detail is available online.[1]

A. Code Coverage

It is important to note that our proposed EvoSuite extensions are designed to enhance the visibility of the classes under test, thereby mitigating observability limitations during assertion generation. This enhancement is directly correlated with the mutation score. Consequently, by construction, the coverage of the tests generated by both the baseline and the treatments remains unchanged. Our results confirm this assumption as the treatments do not exert a direct impact on the coverage metrics, and there are no significant differences between the treatments. The average resulting test suite's coverage for the 100 classes is 62.13% where the `ElGamalKeyParameters` class has the highest resulting coverage on the set, 97.20%, 3 classes were not covered by any test generated (`FileNodeModel`, `OpenPreviousDialog`, `PreviewDialog`) and 9 classes of the set reached full branch coverage.

[1]https://figshare.com/s/46bfa8872f668fafdf66

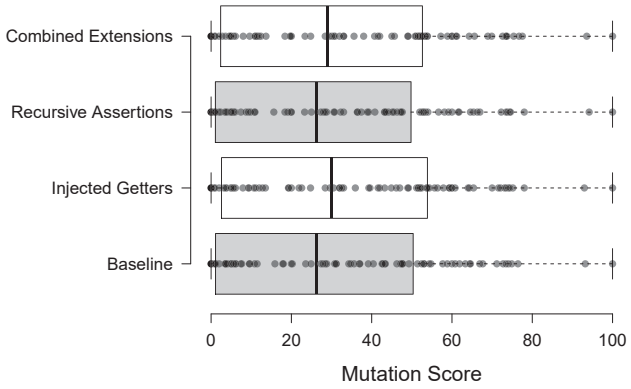

Fig. 1. Mutation score of the 100 classes in the different treatments compared to the baseline

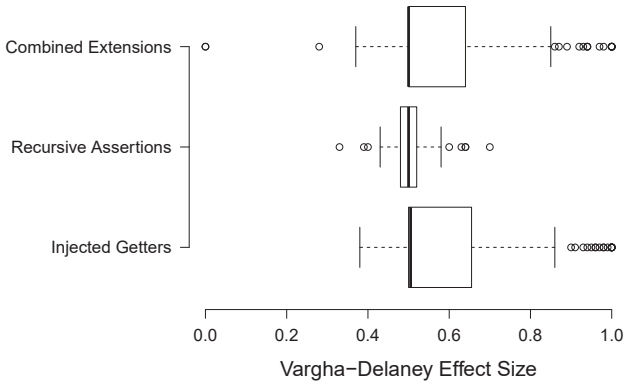

Fig. 2. Vargha-Delaney size effect of the 100 classes in the different treatments

B. Baseline Mutation Score

We generated tests for the 100 classes under analysis using the DynaMOSA EvoSuite algorithm, to take these results as baseline for comparison. EvoSuite produced 3000 test suites, 30 per class. On average, the generated test suites reached an average mutation score of 28.62%. For 22 classes, the generated test suites did not detect any mutants in any of their 30 generations, and for only one class, the generated test suites killed all the injected mutants.

C. Object State Visibility

To answer our first research question (RQ1), we employed our first extension, which introduces public accessors to private and protected attributes that previously lacked such accessors.

Upon analyzing the 100 classes under study, we observed that only 60 classes were augmented with at least one injected public accessor. Therefore, the remaining 40 classes either did not contain any private or protected attributes, or their attributes were already equipped with public accessors. Consequently, the bytecode of these classes remained unchanged. On average, the generated test suites with the injected accessors contained 132 more assertions than the baseline.

We conducted a manual analysis and observed that the generated tests for 40 classes did not contain any assertions using an injected getter. Consequently, their mutation scores did not experience any variation. No tests were generated for three classes with injected accessors, and consequently, these classes were not further considered in the analysis. The remaining 57 classes contain injected getters in the assertions of their generated tests. Through statistical analysis, we found that 26 of these 57 classes have a significant increase in their average mutation score when using injected getters. Table I summarizes these 26 classes and the difference between the mutation score and the number of mutants killed. By using the injected accessors, the 26 classes improved their mutation score from 0.13% to 49%, with one class killing 61 additional mutants. Therefore, 45.61% of the classes that used injected accessors significantly increased their mutation score and detected more mutants.

Figure 2 illustrates the variation of the effect size for the 100 classes studied. We found that 49 classes have an $\hat{A}12$ value greater than 0.50, but only 26 of these classes show a statistically significant increase in the mutation score. The $\hat{A}12$ values for these classes range from 0.56 to 1. For 36 classes, although they have a positive effect size ($\hat{A}12 > 0.5$), the difference in mutation scores is not significant, with $\hat{A}12$ values ranging from 0.503 to 0.603. The remaining 51 classes do not present a significant improvement and have a $\hat{A}_{12}$ of 0.5 or close to it, indicating that the treatment used does not impact the mutation score.

We conducted a manual review of these 26 classes with a significant increment in the mutation score and confirm that the mutation score increment is due to the use of injected getters in the assertions. In total, 435 new mutants were detected, with an average of 16 additional mutants per class. We were able to confirm that on each of the 26 classes, the injected accessors helped detect mutants related to class initialization, variable value changes, and conditionals. The 26 classes contained 177 injected getters. We found that only 76 helped detect at least one new mutant. On average, each of the 26 classes used only 47% of their injected getters. In most cases, no test in the suite covers the lines where the field linked to an injected accessor is modified. However, if we consider all the injected getters in the 100 selected classes, only 76 of 346 (22%) injected getters were helpful.

TABLE I
RESULTS FOR SIGNIFICANT CLASSES (P-VAL < 0.05) USING INJECTED GETTERS CONSIDERING MUTATION SCORE (MS) AND NUMBER OF KILLED/DETECTED MUTANTS(#KM).

Class	Baseline		Injected Getters		ΔMS	$\hat{A}_{12}$
	MS	#KM	MS	#KM		
AlphabeticTokenizer	37%	110	49.2%	140	12.2%	0.86
AxisServlet	5%	10	12%	25	7%	1.00
AZMessageDecoder	40.4%	149	47%	167	6.6%	0.91
AZOtherInstanceImpl	71.2%	81	74.1%	84	2.9%	0.72
CalEventModelImpl	66.9%	560	71.6%	598	4.6%	0.97
DbConnectionBroker	27.4%	64	42%	97	14.6%	1.00
DHTRouterNodeImpl	37%	105	44.9%	129	7.9%	0.99
DirEntry	47.6%	35	51.5%	37	3.5%	0.81
DLFileEntryModelImpl	67.7%	929	72.1%	990	4.4%	0.98
EngineImpl	7.6%	17	9.4%	19	1.8%	0.66
GetRevision	43.3%	20	46.1%	21	2.8%	0.95
HL7CheckerImpl	9.6%	12	12.8%	16	3.2%	0.96
HtmlViewerPanel	42.6%	129	59.3%	180	16.7%	1.00
JoomlaOutput	47.6%	44	59.3%	55	11.7%	0.96
NoenFormatter	27.9%	8	52.0%	15	24.1%	1.00
NotificationEventComparator	47.7%	32	56.4%	37	8.7%	0.65
OrganizationModelImpl	61.9%	393	65.4%	417	3.5%	0.94
Parser	3.6 %	114	4.9%	145	1.3%	0.93
PDFProcessorImpl	7.4%	39	7.9%	41	0.5%	0.73
PollsChoiceModelImpl	47.2%	270	51.2%	294	4%	0.90
PortalExecutorFactoryImpl	18%	12	32.6%	22	14.6%	0.98
PreLaunchHelperImpl	15.9%	26	19.3%	33	3.4%	0.80
RDResumeHandler	0.9%	3	1%	7	0.1%	0.57
SACPluginActivator	1.2%	1	13.5%	6	12.3%	1.00
ScriptOrFnScope	46.3%	27	51.7%	30	5.4%	1.00
TestJava	0%	0	49%	18	49%	1.00

Finding 1. Increasing the visibility of class attributes improved EvoSuite's mutation score between 0.13% and 49% in the tests generated for 26 classes. A manual review confirmed that these additional mutants killed were directly associated with assertions utilizing the injected accessors. This underscores the importance of considering observability in the test generation process.

The generated tests for 49 classes show no variation in the mutation scores, despite containing injected public accessors. In 18 classes, despite having between 1 and 13 injected public accessors, EvoSuite did not use them in the assertion generation process. We analyzed these classes and found that for 11 of them, EvoSuite generated only test methods that ended in exceptions, resulting in no assertions being used. For 7 classes, EvoSuite deleted the assertions that used the injected accessors during the mutant-based minimization process because they were unrelated to any mutants.

In the remaining 31 classes of the 49 without changes in mutation score, we found that the generated tests included injected getters in the assertions. However, they showed no changes in their mutation scores, as the generated assertions killed the same mutants as the baseline, showing no improvements. For instance, in the class `ConnectionConsumer`, the 25 detected mutants in the baseline remained after applying the treatment. It is likely that in these cases, the injected getters are related to attributes that are not associated with the undetected mutants, the generated tests do not even execute the undetected mutants due to low coverage, or the mutants related to these

TABLE II
RESULTS FOR SIGNIFICANT CLASSES (P-VAL < 0.05) USING RECURSIVE ASSERTIONS CONSIDERING MUTATION SCORE (MS) AND NUMBER OF KILLED/DETECTED MUTANTS(#KM)

Class	Baseline		Rec Assertions		ΔMS	$\hat{A}_{12}$
	MS	#KM	MS	#KM		
AZOtherInstanceImpl	71.2%	81	73.8%	84	2.6%	0.70
HtmlViewerPanel	42.6%	129	44.2%	131	1.6%	0.63
isc_stmt_handle_impl	93.2%	331	94.2%	333	1%	0.64
ShoppingCategoryWrapper	17.9%	9	19.9%	11	2%	0.64

attributes are already killed by other assertions.

These observations on the use of the injected getters can be seen in the Vargha-Delaney distribution where some classes with a higher effect size have few instance variables while others show no effect despite numerous injected accessors, for example, `NoenFormatter` class is injected with only one getter and reports an $\hat{A}_{12}$ value of 1.00 while the `MacawStateEditor` class contained 13 injected getters but only reaches a $\hat{A}_{12}$ value of 0.50, suggesting further research is needed to determine the conditions under which a getter is beneficial.

Finding 2. The increment in the mutation score is not directly related to the number of injected getters, as not all injected getters may provide benefits. This identifies an area for further research to develop strategies for determining which attributes require accessors.

D. Asserting Composed Objects

To understand to what extent the lack of observability is due to EvoSuite's inability to assert on composed objects (RQ2), we now consider our second EvoSuite extension, which recursively checks the states of composed objects using a threshold. Note that treatment two only assesses the impact of recursive assertions without injecting getters to the code under tests. After generating tests for the 100 classes, we manually checked the 30 test suites generated for each class, one for each seed, and found that 33 of the 100 checked classes at least one generated test suite used a recursive inspection assertion. On average, the generated test suites that allowed asserting composed objects contained 218 more assertions than the baseline. Therefore, we manually analyzed the assertions in these 33 cases, of which 18 were included in the 57 classes that were injected with getters in the previous treatment.

We compared mutation scores and found that only in four cases the difference in mutation score is statistically significant. This represents 12.12% out of the 33 classes that use recursive inspection assertions. The increment of mutation score ranges from 0.96% to 2.63%, and a maximum of 3 new mutants are detected. We manually reviewed the newly detected mutants for each class and found a direct relation between recursive assertions and mutants changing the values of internal objects or the initialization of a composed object; this means that asserting the composed object was related to mutants injected to change the internal value of an element. Table II summarizes the mutation scores, the number of killed mutants, and the effect size of these four classes.

Figure 1 shows the comparison between the mutation scores on the baseline and using recursive assertions. The tendency between the boxplots does not reveal a significant impact of the recursive assertions on the mutation score; for most classes, the effect size remained at 0.5, implying the same performance as the baseline. This low impact is also observed in Figure 2 compared to the previous treatment. For the 100 classes under analysis, only 29 classes had a $\hat{A}_{12}$ value slightly higher than 0.50, varying between 0.51 and 0.7. Of these 29 classes, only 4 show a statistically significant increase in the mutation score. The remaining 71 classes do not experience increases in their mutation scores, and their Vargha-Delaney values are 0.5 or close to it.

Finding 3. The generation of recursive inspection assertions had a slight impact on mutation scores, improving them between 0.96% and 2.63% for only four classes. Our manual review confirms that this increment is directly related to the generated recursive assertions. This shows that asserting complex objects can impact mutation scores in specific cases, but further research is needed on alternative assertion generation methods, as a considerable portion of mutants remain undetected.

E. Open State Visibility and Asserting Composed Objects

Our findings show that encapsulation impacts the effectiveness of generated tests, particularly the generated assertions. Our proposed assert generation technique is not exempt from this. To understand the degree to which encapsulation limits recursive assertion generation, we evaluate the effectiveness of asserting recursively composed objects when all attributes of the code under test are visible, which we can simulate by using both extensions at the same time. After generating tests for the 100 classes using a combination of our previously implemented extensions, we observe that 61 of them use one or both extensions, 7 use only injected getters in their test suites, 15 use only recursive assertions, and 39 use both extensions, 9 use the extensions combined in their assertions, and 30 separately. Therefore, we focus on the assertions generated for these 61 classes using 30 different seeds. On average, the generated test suites resulting from the use of the combined extensions contained 1,286 more assertions than the baseline.

Statistical analysis shows that in 26 of these 61 cases the mutation score increased significantly. Of these 26 classes, 7 only use injected getters, no class uses recursive assertions without using injected getters of the composed objects, and 19 use both extensions in their assertions, 2 classes combine the extensions in the assertions and 17 use them separately. Table III details the mutation score, number of killed mutants, and the effect size of each one of these 26 classes.

We conducted a manual review to determine the relation between the assertions generated for the 26 significantly improved classes and the new detected mutants. We found

TABLE III
RESULTS FOR SIGNIFICANT CLASSES (P-VAL < 0.05) USING COMBINED EXTENSIONS CONSIDERING MUTATION SCORE (MS) AND NUMBER OF KILLED/DETECTED MUTANTS(#KM)

Class	Baseline		Combined Extensions		ΔMS	$\hat{A}_{12}$
	MS	#KM	MS	#KM		
AlphabeticTokenizer	37%	110	49.1%	141	12.1%	0.86
AxisServlet	5%	10	12%	25	7%	1
AZOtherInstanceImpl	71.2%	81	73.7%	85	2.5%	0.71
CalEventModelImpl	66.9%	560	68.9%	587	2%	0.89
DbConnectionBroker	27.4%	64	42%	97	14.6%	1
DHTRouterNodeImpl	37%	105	45%	130	8%	0.97
DirEntry	47.6%	34.5	52%	37	4.4%	0.85
DLFileEntryModelImpl	67.7%	929	69.9%	956	2.2%	0.79
EngineImpl	7.6%	17	10.9%	19	3.3%	0.69
HL7CheckerImpl	9.6%	12	12.4%	15	2.9%	0.94
HtmlViewerPanel	42.6%	129	51.9%	180	9.3%	0.87
JoomlaOutput	47.6%	44	52.8%	49	5.1%	0.93
NewScheduler	39.2%	43.5	45.7%	52	6.5%	0.67
NoenFormatter	27.9%	8	53.9%	15	26%	1
NotificationEventComparator	47.7%	32	77.5%	37	29.8%	0.69
OrganizationModelImpl	61.9%	393	65.6%	418	3.7%	0.94
Parser	3.6%	114	4.9%	148	1.3%	0.94
PDFProcessorImpl	7.4%	39	8%	40	0.6%	0.74
PollsChoiceModelImpl	47.2%	270	51.6%	294	4.4%	0.92
PortalExecutorFactoryImpl	18%	12	33%	22	15%	0.98
PreLaunchHelperImpl	15.9%	26	19.9%	33	4%	0.8
RDResumeHandler	0.9%	3	1%	7	0.1%	0.57
SACPluginActivator	1.2%	1	13.7%	7	12.5%	1
ScriptOrFnScope	46.3%	27	51.6%	30	5.3%	1
TestJava	0%	0	49%	18	49%	1
Version	53.1%	139	53.8%	142	0.8%	0.6

a direct relation between the assertions generated and the mutants detected, in most cases the generated tests contained in the same test assertions generated using both extensions, instead of only one. For instance, consider the generated test for `AZOtherInstanceImpl` on Listing 7 where different assertions on the same test use both extensions. For example, the assertion that contains the `get_UDP_non_data_portKeyInjected` is related to mutants created to simulate bugs during object construction. The two last assertions are related to test the internal state of the composed object of the class `InternalAddress`. This check is useful because of mutants dedicated to change the assignment of the attribute. The classes with significant improvement found for combined extensions correspond to the classes found with the use of only injected getters excepting two `AZMessageDecoder` and `GetRevision` where their generated tests contained unstable assertions. However, two new classes had significant improvement in their mutation score, `NewScheduler` and `Version`.

The increment on mutation score in these classes ranges from 0.6% to 49% and a maximum of 51 new mutants were detected. Figure 1 shows the comparison between the mutation score of the generated test suites using both extensions, using only one and not using any. Note that in all cases, the extensions do not lower the mutation score of the classes under test. The figure also reveals a notable impact of the extensions used.

Compared to the exclusive use of injected getters, the addition of recursive assertions raised the mutation score by a maximum of 21.13% in the `NotificationEventComparator` class. However, this addition did not increase the mutation score of the other classes, which has an average increase of 0.30% on the mutation score.

The combination of both extensions improved the mutation score across 26 classes, with 19 classes using both recursive assertions and injected getters. However, 24 classes showed similar improvements with only injected getters, indicating a minimal additional benefit from asserting complex objects. These findings suggest that, while recursive assertions are beneficial, they do not fully address the limitations of assertion generation. Further research is required to explore alternative methods to enhance assertion generation.

Finding 4. The combination of both extensions improved the mutation score from 0. 6% to 49% in 26 classes, with 19 classes using both recursive assertions and injected getters. While recursive getters are beneficial, they do not fully address the limitations of assertion generation. Further research is required to explore alternative methods to enhance assertion generation.

For the Vargha-Delaney $\hat{A}_{12}$ statistic, 42 of these 61 classes showed a value greater than 0.50, suggesting a 68.85% chance of higher mutation scores as a result of using the combined extensions. Figure 2 shows the comparison of $\hat{A}_{12}$ between treatments. The figure shows a higher concentration of classes surpassing the 0.5 value. The classes that used the extensions have on average a 0.70 $\hat{A}_{12}$ value, a higher effect than the average of the significative classes of separate extensions. However, no clear relation between the use of the injected getters and the effect size is found.

Finding 5. Although the number of significant classes has increased using extensions combined rather than separately, the recursive assertions do not demonstrate a higher effect on the mutation score. The results using injected getters remark the importance of class attributes observability for generating more effective test suites.

```
public void test00()  throws Throwable  {
  ...
  AZOtherInstanceImpl aZOtherInstanceImpl0 = new
       AZOtherInstanceImpl("yYl_u{kvr}", "`BfD",
       inetAddress0, inetAddress0, 744, 744, (-385),
       hashMap0);
  assertEquals(744,
    aZOtherInstanceImpl0.getUDP_Port_());
  assertEquals((-385),
    aZOtherInstanceImpl0.getUDP_non_data_port_Injected());
  ...
  assertEquals("127.0.0.1",
    aZOtherInstanceImpl0.
         getInternalAddress().getHostAddress());
  assertFalse(
    aZOtherInstanceImpl0.
         getInternalAddress().isMCSiteLocal());
}
```

Listing 7. Test with injected getters and recursive assertions.

VI. Discussion

A. Encapsulation

Our results show that injecting public accessors enhances the observability of internal states, thereby improving mutation testing effectiveness by exposing class attributes for verification. However, manual analysis revealed that not all injected accessors were beneficial. Among the 26 classes with significant improvements in the mutation score, 177 accessors were injected, but only 76 were used by EvoSuite to detect additional mutants. This highlights that while increased accessibility can be advantageous, its use in production code must be carefully managed to avoid violating encapsulation principles, which could compromise modularity and security.

Maintaining a delicate balance requires strategic decisions regarding public accessors. Developers should selectively implement public accessors to maximize testing coverage without compromising class structure. Future research should focus on developing techniques that guide developers in identifying attributes that need accessibility, helping them make informed decisions on injecting public accessors. This approach would optimize the trade-off between enhanced testability and preserving encapsulation principles.

B. Recursive Assertions

To assert the state of composite objects, we proposed a direct variation of the EvoSuite inspector assertion that assesses the states of these objects through consecutive method accessor calls. This strategy complements our extension, which facilitates the automatic introduction of accessors. Nevertheless, it is important to acknowledge that alternative methods, such as incorporating functions like `equals` or `hashCode`, could further enhance class observability. A significant advantage of employing recursive inspector assertions is their ability to simplify the debugging process: if a test fails, the problematic attribute with an unexpected value is easily identified.

Our results show that recursive inspector assertions may be useful in certain cases where the effects of artificially injected mutants have side effects on composed objects. We also observed that the use of these assertions did not lead to a noticeable variation in performance (e.g., test generation time). However, further work is needed to evaluate the potential of recursive inspector assertions, particularly with real bugs and a wider range of classes. This could help identify the cases in which these types of assertions are most useful. An alternative approach is to understand how manually created tests kill mutants similar to the ones that remain undetected and learn from them to create new strategies for generating assertions.

C. Observability vs. Propagation

Our experiments demonstrated that the observability of internal states has a significant effect on the effectiveness of generated unit tests, but we also found that this is not simply due to a lack of the ability to assert on complex object states. Besides guiding developers in improving the observability of their code and revisiting assertion generation techniques, there is an orthogonal aspect to consider: The process of how faults lead to software failures has been studied intensively. In particular, the RIP model [45]–[47] describes that faults need to be **R**eached (i.e., executed), cause a state **I**nfection, which then needs to **P**ropagate to an observable output; the PIE model (Execution, Infection, and Propagation) describes the same process. A recent extension of the RIP model [48] includes a final requirement *Revealability*.

Our investigation of assertion generation targets this last step of *Revealability*. We based our experiments on the common practice of automatically generating unit tests for code coverage; therein lies a potential problem related to observability: Optimizing tests for code coverage only ensures that *Reachability* is satisfied, but whether or not *Infection* happens is coincidental, and generated tests may require further calls to ensure propagation of infected internal states to allow generating adequate assertions. Consequently, it will be important for future work to also consider optimization goals for test generation that go beyond code coverage [38], [49].

We did not find a clear relation between code coverage and the improvement in mutation score in any of our experiments. In the first treatment, the average coverage achieved by the 26 classes with mutation coverage improvement ranges between 23.17% and 96% where only 8 classes had an average coverage $< 70\%$ and only 2 classes exceeded 90% coverage. However, classes such as `SACPluginActivator` and `JoomlaOutput` report a high mutation score increase ($> 11\%$) reaching 49.9% and 68.83% average coverage respectively. In contrast, 7 classes with an average coverage $> 80\%$ had a mutation score improvement $< 5\%$. In the second treatment, all the 4 classes with significative increase had an average coverage $> 80\%$ with similar mutation score improvement. Finally, the third treatment presented similar results as the first one, without directly relating the coverage with the improvement in mutation score.

D. Assertion Minimization

The recursive inspector assertion strategy may lead to a large number of assertions for each composed object and even more for each generated test. Furthermore, it may introduce indirect testing, a well-known test smell, as these assertions may end up testing methods that do not belong to the class under test. In such cases, the EvoSuite mutation-based assertion minimization algorithm plays an important role by reducing the number of generated assertions using mutation analysis, retaining only those directly related to a mutant. Consequently, the remaining assertions are considered useful for detecting mutants injected into the class under test. However, as previous studies have shown, EvoSuite-generated tests, even without our extensions, may contain indirect testing smells among other types of test smells [17], [18]. Therefore, further research is needed to assess whether recursive assertions or increased observability might decrease the quality of the generated tests, particularly by increasing the number of test smells present in the generated code.

VII. Related Work

Controllability and observability are widely studied factors [14], [22]–[27], [50], [51] in research on testability. Controllability involves the ease with which testers can manipulate state and input to achieve specific test conditions and control execution flow. On the other hand, observability involves the ability to observe and verify the outputs and behaviors of the unit under test. This section summarizes related work aimed at improving testability.

Diverse approaches have been proposed to improve the testability of the code under test. A common method involves performing automatic transformations on the program components to enhance their testability. These transformations target various source code patterns that are known to affect testability and reduce the effectiveness of test generator tools. For instance, they address boolean flags that may be present in conditionals [52], [53] or loops [54], as well as nested conditionals [55] and unstructured control flows [56]. Additionally, these transformations can also be performed at the bytecode level [57].

Refactorings can significantly enhance the testability of program components. Previous studies indicate that class design and code quality impact testability. Cinnéide et al. [58] used a search-based refactoring tool [59] to perform various refactorings on the class under test. Practitioners developed tests before and after refactoring, suggesting improved testability, though the results were not definitive. Reich et al. [19] analyzed 200 pull requests aimed at improving testability, identifying ten beneficial refactoring patterns. Common patterns included method extraction for overriding or invocation, widening method access, and class extraction for invocation.

Improving observability is another method to enhance testability. Several studies focus on tracking execution to inject assertions for checking constraints [60], pre/post conditions, or class invariants [61], [62]. For instance, Kansomkeat et al. [22] proposed injecting observability probes to track variable states and insert assertions to verify them. Mao [63] uses Aspect-Oriented Programming (AOP) to verify component invariants by introducing a tracing aspect to gather preconditions of method executions, aiding in regression testing. Schuler and Zeller use dynamic slicing to determine checked coverage, the statements with an actual contribution to the test oracles [12]. Finally, Zhu et al. proposed 19 code observability metrics with a strong correlation with the mutation score [64].

In contrast to previous work, our study evaluates the impact of observability on test generation algorithms. We conducted an experiment that exposed encapsulated attributes and enabled state assertions of complex objects. Our results show that improved observability significantly increases mutation scores. This highlights the need for further research to understand how observability affects the effectiveness of test generation.

VIII. Conclusion

This paper introduces two complementary EvoSuite extensions designed to enhance class observability by publicly exposing all class attributes and enabling EvoSuite to assert the states of composite objects. We implemented these extensions to evaluate their impact on test generation tools, particularly EvoSuite. Our empirical study, conducted using a well-known set of 100 complex Java classes from the SF110 dataset, demonstrated that increased observability through our extensions led to a significant improvement in the mutation score for generated tests for 26 classes. This underscores the importance of class observability in the test generation process. In future work, we plan to study the effects not only in a mutation testing scenario but also on real faults, to explore heuristics to identify variables that significantly benefit from being accessed by generated tests, as well as further improved assertion generation techniques. We also aim to extend our study using other mutation engines (e.g., PIT [65]) with varied mutation operators, and study the mutants detected vs. the mutants reached by the test suite.

Acknowledgment

Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this research. We also thank the Programa de Inserción Académica 2022, Vicerrectoría Académica y Prorrectoría, at the Pontificia Universidad Católica de Chile.

References

[1] A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *2008 The Third International Conference on Software Engineering Advances*, USA, Jul. 2008, pp. 252–257.

[2] P. Straubinger and G. Fraser, "A survey on what developers think about testing," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 80–90.

[3] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, New York, NY, USA, Sep. 2011, p. 416–419.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, USA, Nov. 2007, p. 75–84.

[5] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[6] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Delft University of Technology, NLD, Tech. Rep., 2001.

[7] G. Meszaros, *xUnit Test Patters*. Boston, MA: Addison-Wesley, 2007.

[8] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, USA, Nov. 2019, p. 193–202.

[9] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[10] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi, "Test smell detection tools: A systematic mapping study," in *Evaluation and Assessment in Software Engineering*, New York, NY, USA, Jun. 2021, p. 170–180.

[11] G. Galindo-Gutierrez, M. N. Carvajal, A. Fernandez Blanco, N. Anquetil, and J. P. Sandoval Alcocer, "A manual categorization of new quality issues on automatically-generated tests," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE, 2023, pp. 271–281.

[12] D. Schuler and A. Zeller, "Checked coverage: an indicator for oracle quality," *Software testing, verification and reliability*, vol. 23, no. 7, pp. 531–551, 2013.

[13] L. Ma, C. Zhang, B. Yu, and H. Sato, "An empirical study on the effects of code visibility on program testability," *Software Quality Journal*, vol. 25, pp. 951–978, 2017.

[14] V. Garousi, M. Felderer, and F. Kılıçaslan, "A survey on software testability," *Information and Software Technology*, vol. 108, pp. 35–64, Apr. 2019.

[15] M. M. Hassan, W. Afzal, M. Blom, B. Lindström, S. F. Andler, and S. Eldh, "Testability and software robustness: A systematic literature review," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2015, pp. 341–348.

[16] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, Dec. 2014.

[17] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[18] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. Hellendoorn, "Test smells 20 years later: Detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, 2022.

[19] P. Reich and W. Maalei, "Testability refactoring in pull requests: Patterns and trends," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1508–1519. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00131

[20] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Boston, MA: Addison-Wesley Professional, 2009.

[21] J. Voas and K. Miller, "Software testability: the new verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, 1995.

[22] S. Kansomkeat and W. Rivepiboon, "An analysis technique to increase testability of object-oriented components," *Softw. Test. Verif. Reliab.*, vol. 18, no. 4, p. 193–219, dec 2008.

[23] R. Poston, J. Patel, and J. Dhaliwal, "A software testing assessment to manage project testability," *ECIS 2012 - Proceedings of the 20th European Conference on Information Systems*, vol. 219, 01 2012.

[24] B. Lo and H. Shi, "A preliminary testability model for object-oriented software," in *Proceedings. 1998 International Conference Software Engineering: Education and Practice (Cat. No.98EX220)*. Los Alamitos, CA, USA: IEEE, 1998, pp. 330–337.

[25] J. Gao and M.-C. Shih, "A component testability model for verification and measurement," in *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, vol. 2. Los Alamitos, CA, USA: IEEE, 2005, pp. 211–218 Vol. 1.

[26] Y. Zhou, H. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, and B. Xu, "An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems." *SCIENCE CHINA Information Sciences*, vol. 55, no. 12, pp. 2800–2815, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/chinaf/chinaf55.html#ZhouLSZLCX12

[27] V. Chowdhary, "Practicing testability in the real world," in *2009 International Conference on Software Testing Verification and Validation*. Los Alamitos, CA, USA: IEEE, 2009, pp. 260–268.

[28] R. V. Binder, "Design for testability in object-oriented systems," *Commun. ACM*, vol. 37, no. 9, p. 87–101, sep 1994. [Online]. Available: https://doi.org/10.1145/182987.184077

[29] S. Almugrin, W. Albattah, and A. Melton, "Using indirect coupling metrics to predict package maintainability and testability," *Journal of Systems and Software*, vol. 121, 03 2016.

[30] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, march-april 2012.

[31] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, USA, Jul. 2015, p. 338–349.

[32] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[33] ——, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[34] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA, Jul. 2015, p. 1367–1374.

[35] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri, "Random or genetic algorithm search for object-oriented test suite generation?" *Software Testing, Verification and Reliability*, vol. 28, no. 4, pp. 1660–1660, 2018.

[36] A. Panichella, J. Campos, and G. Fraser, "Evosuite at the sbst 2020 tool competition," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, USA, Jul. 2020, p. 549–552.

[37] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[38] ——, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, pp. 783–812, 2015.

[39] W. Conover, *Practical nonparametric statistics*, 3rd ed., ser. Wiley series in probability and statistics. New York, NY [u.a.]: Wiley, 1999. [Online]. Available: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+24551600X&sourceid=fbw_bibsonomy

[40] A. Vargha and H. Delaney, "A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics - J EDUC BEHAV STAT*, vol. 25, 06 2000.

[41] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[42] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 360–369.

[43] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921.

[44] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 163–171.

[45] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, 1990.

[46] R. A. DeMillo, A. J. Offutt *et al.*, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[47] H. Du, V. K. Palepu, and J. A. Jones, "Ripples of a mutation—an empirical study of propagation effects in mutation testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[48] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Transactions on software Engineering*, vol. 18, no. 8, p. 717, 1992.

[49] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 212–222.

[50] V. Terragni, P. Salza, and M. Pezzè, "Measuring software testability modulo test quality," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 241–251. [Online]. Available: https://doi.org/10.1145/3387904.3389273

[51] L. Guglielmo, L. Mariani, and G. Denaro, "Measuring software testability via automatically generated test cases," *IEEE Access*, vol. 12, pp. 63 904–63 916, 2024.

[52] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, p. 1359–1366.

[53] S. Wappler, A. Baresel, and J. Wegener, "Improving evolutionary testing in the presence of function-assigned flags," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*

(TAICPART-MUTATION 2007). Los Alamitos, CA, USA: IEEE, 2007, pp. 23–34.

[54] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, p. 108–118, jul 2004. [Online]. Available: https://doi.org/10.1145/1013886.1007527

[55] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 3, jun 2009. [Online]. Available: https://doi.org/10.1145/1525880.1525884

[56] R. Hierons, M. Harman, and C. Fox, "Branch-coverage testability transformation for unstructured programs," *The Computer Journal*, vol. 48, 05 2005.

[57] Y. Li and G. Fraser, "Bytecode testability transformation," in *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer, Sep. 2011, pp. 237–251.

[58] M. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. Los Alamitos, CA, USA: IEEE, 2011, pp. 437–443.

[59] M. O'Keeffe and M. Cinneíde, "Search-based refactoring: An empirical study," *Journal of Software Maintenance*, vol. 20, pp. 345–364, 09 2008.

[60] N. Pan and E. Song, "An aspect-oriented testability framework," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, ser. RACS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 356–363. [Online]. Available: https://doi.org/10.1145/2401603.2401682

[61] Y. Singh and A. Saha, "Improving the testability of object oriented software through software contracts," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, p. 1–4, jan 2010. [Online]. Available: https://doi.org/10.1145/1668862.1668869

[62] L. Briand, Y. Labiche, and H. Sun, "Investigating the use of analysis contracts to improve the testability of object-oriented code," *Softw., Pract. Exper.*, vol. 33, 06 2003.

[63] C. Mao, "Aop-based testability improvement for component-based software," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 2. Los Alamitos, CA, USA: IEEE, 2007, pp. 547–552.

[64] Q. Zhu, A. Zaidman, and A. Panichella, "How to kill them all: An exploratory study on the impact of code observability on mutation testing," *Journal of Systems and Software*, vol. 173, p. 110864, 2021.

[65] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707