New Trends and Ideas

Snapshot testing in practice: Benefits and drawbacks☆

Victor Pezzi Gazzinelli Cruz [a],[*], Henrique Rocha [b], Marco Tulio Valente [a]

[a] *Department of Computer Science, Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil*
[b] *Department of Computer Science, Loyola University Maryland, Baltimore, United States*

ARTICLE INFO

ABSTRACT

In recent years, snapshot testing has gained attention as a novel testing technique. Even though it is used by well-known companies, there is a gap in the scientific literature concerning the tradeoffs of this testing technique. Therefore, in this paper, we investigate the adoption of snapshot testing in practice through a grey literature review, aiming to reveal tradeoffs, best practices, and tools commented by practitioners. Our findings show that snapshots are simple to create and can help in preventing regressions; however, they may swiftly become fragile if used improperly. We also envision an opportunity for further research on snapshot testing, for example by mining and analyzing project samples from public repositories.

Contents

1. Introduction

Snapshot testing is a new form of testing that has gained popularity in recent years (Panchuk, 2022). Essentially, it can be used as a User Interface (UI) test that compares the current state of the interface to a previously stored state (or snapshot). By adopting snapshots tests, developers aim to detect unexpected UI changes without devoting a major effort designing and implementing tests with complex and detailed assertions (Nakazawa, 2016). The significance of snapshot testing in practice can be illustrated by the number and importance of the software companies that are using this type of test. For example, Airbnb's iOS app has approximately 30,000 snapshot tests, which is three times greater than the number of unit tests in the app. As other examples, Spotify and Shopify apps have more than 1000 snapshot tests

---

each one (Smiley, 2022). Additional examples of companies using snapshot tests include Uber, Lyft, and Robinhood.

Nonetheless, as with any testing technique, snapshot testing is not a silver bullet. Despite knowing that, we still lack curated information on the real benefits and drawbacks of this novel testing technique. This information is important to better support developers when deciding whether they should or not adopt snapshot testing in their projects. It can also help developers that are already using snapshot tests to better embrace this type of testing.

However, to the best of our knowledge, there are no previous studies in the literature focused on investigating the benefits and drawbacks of snapshot tests. For example, we ran a Google Scholar search and could not identify any peer-reviewed papers addressing this type of testing explicitly.

Thus, in this paper, we aim to explore the benefits and drawbacks of snapshot testing by means of a grey literature review, which is a recommended research methodology to study emerging programming technologies (Kamei et al., 2021; Zhang et al., 2020). More specifically, by using a search engine we initially gathered 100 recent documents on snapshot testing, including blog posts, video transcripts, and GitHub discussions. After applying a set of filters, we selected 50 documents for detailed reading, review, and analysis. Our goal was to answer five research questions:

- RQ1: What are the main benefits of snapshot tests?
- RQ2: What are the main drawbacks of snapshot tests?
- RQ3: What are the best practices when using snapshot tests?
- RQ4: What architectural components are tested using snapshot testing?
- RQ5: Which libraries and tools are commonly used for implementing snapshot tests?

Our key findings can be summarized as follows:

1. Snapshot tests are mostly adopted due to their easiness of implementation, which allows developers to write more tests and consequently prevent regressions in their projects.
2. The most common drawback is the fragility of snapshot tests, leading to the necessity of constant updates of the "golden standard" snapshot.
3. Practices such as treating snapshots as code that is subjected to code reviews, and the usage of detailed textual descriptions can mitigate the mentioned downsides.
4. Snapshot testing is mostly used in frontend components.
5. Jest is the most popular tool for implementing such tests.

The remainder of this paper is organized as follows. In Section 2, we present background information on snapshot testing. In Section 3, we present the design of our study. We detail and discuss our key results in Section 4. In Section 5, we list threats to validity. Finally, we present related work in Section 6 and our conclusions in Section 7.

2. Snapshot testing

Snapshot testing is a type of golden master testing, meaning that the first time the test runs, it just takes its result (golden master) as a snapshot. After that, every time the test runs it compares the current state of the component under test to the stored snapshot. If the two instances are identical, the test passes. Otherwise, it fails. Fig. 1 shows the general workflow for snapshot testing.

When a snapshot test fails, the testing framework highlights and displays the differences between the snapshot and the current state. There are two possible causes for such differences:
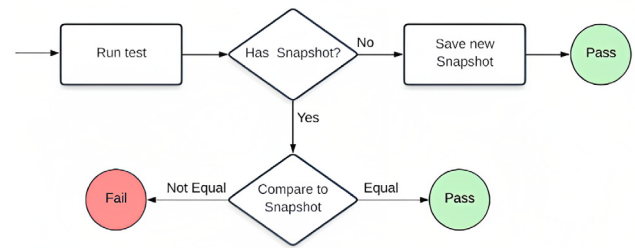

Fig. 1. The general workflow for snapshot testing.

Fig. 2. Facebook logo icon.

1. The difference is indeed a consequence of a bug introduced in the code (true positive).
2. The difference is caused by an intended change in the code, for example, to support a new requirement or a changing one (false positive).

When a false positive happens, developers should update the snapshot, introducing a new "golden standard" based on the current state, and thus resolving the failed test. Testing frameworks such as Jest have the option of encoding snapshots into a file, making it simpler to check the tests on further runs, and also serving as an artifact to be reviewed as part of a code review process.

Suppose we are using React[1] to implement a web component to display a small icon for a social media link.[2] Initially, the only social media we had was Facebook and we hard coded it into our React component (Listing 1). In this component, we have a link (line 3) with the Facebook logo (line 4). Fig. 2 shows the expected display for that component.

```
1 export default function SocialMediaIcon(){
2    return(
3        <a href="http://www.facebook.com">
4            <i className='bi bi-facebook'/></i>
5        </a>
6    );
7 }
```

Listing 1: `SocialMediaIcon` React component

React allows the reuse of our components in other parts of a project. Listing 2 shows an example of using `SocialMediaIcon` in another component called `MainFooter`.

```
1 export default function MainFooter(){
2    return(
3        <div>
4            <SocialMediaIcon />
5        {/* Other footer components */}
6        </div>
7    );
8 }
```

Listing 2: (Re)using `SocialMediaIcon` in another component

To guarantee that our component does not change without our knowledge, we create a snapshot test, as shown in Listing 3.

---

[1] https://reactjs.org/

[2] This example is inspired by the snapshot test example in the Jest webpage: https://jestjs.io/docs/snapshot-testing.

```
- Snapshot   - 2
+ Received   + 2

  <div>
    <a
-      href="http://www.facebook.com"
+      href="http://www.undefined.com"
    >
      <i
-        className="bi bi-facebook"
+        className="bi bi-undefined"
      />
    </a>
  </div>
```

Fig. 3. Snapshot testing diff result example.

```
1 test('Main Footer renders correctly', () => {
2   const visualComponent = renderer
3     .create(<MainFooter />).toJSON();
4   expect(visualComponent).toMatchSnapshot();
5 });
```

Listing 3: Snapshot test for `MainFooter` component

The first time this test runs, a snapshot file is created, exporting a syntax tree of HTML tags, elements, and attributes representing the state of the component under test. From now on, any change to this component will result in a failing test.

Suppose, for example, that some days later, another developer adds another social media link besides Facebook. Particularly, the developer changes the `SocialMediaIcon` component to receive the name as property (Listing 4).

```
1 export default function SocialMediaIcon(props){
2   return(
3     <a href={`http://www.${props.name}.com`}>
4         <i className={`bi bi-${props.name}`}></i>
5     </a>
6   );
7 }
```

Listing 4: Refactored `SocialMediaIcon` component

Now to properly use the modified component, one must call `<SocialMediaIcon name=''facebook'' />`. However, suppose the developer forgets to update the `MainFooter` component, which will therefore invoke the `SocialMediaIcon` component without the name property. In JavaScript, by omitting a property, its value becomes undefined.

After implementing the change, the developer runs the test suite. The snapshot test (Listing 3) will fail and Jest will produce a diff result as the one in Fig. 3. This warns the developer that he/she forgot to update one of the pages, and the developer can now fix the issue.

3. Study design

To the best of our knowledge, we still lack scientific studies examining the benefits and drawbacks of snapshot testing. However, the grey literature – which consists of blogs, forums, videos, etc – already has a good number of documents about this technique. Therefore, we decided to undertake a grey literature review in order to address five research questions:

- **RQ1:** What are the main benefits of snapshot tests?
- **RQ2:** What are the main drawbacks of snapshot tests?
- **RQ3:** What are the best practices when using snapshot tests?
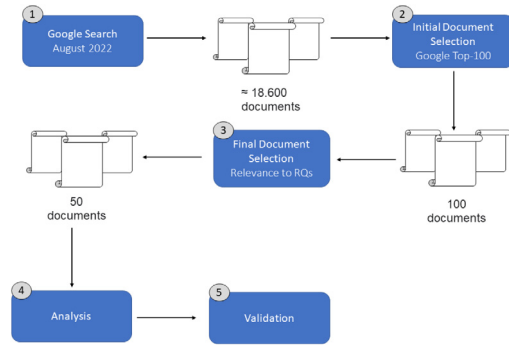- **RQ4:** What architectural components are tested using snapshot testing?

Fig. 4. Overview of the grey literature review steps.

Table 1
Source of the selected documents.

Source	Documents
Reputable Author	38 (76%)
Reputable Blog	5 (10%)
Official Documentation	4 (8%)
Forum Discussion	3 (6%)

- **RQ5:** Which libraries and tools are commonly used for implementing snapshot tests?

Fig. 4 highlights the steps we followed in the grey literature review.

As can be seen, we split our work into five steps:

Google Search: Before executing a search for documents on Google, it is important to calibrate the query strings (for instance, by combining synonyms or removing terms that may impact the results). After conducting this calibration, we decided to use the following query: ("snapshot testing") AND ("benefits" OR "limitations").

Initial Document Selection: Since about 18,600 documents were retrieved by the search, a full review of them is impracticable. Therefore, and as is common in grey literature reviews, it is necessary to restrict the number of articles subjected to in-depth reading and analysis. Thus, we concentrate on the first 10 pages of search results, i.e., the top-100 documents returned by Google.

Final Document Selection: The first author read these 100 documents to narrow down the list to only those that are relevant to the proposed RQs. In three cases, he also discussed the article with the second author before including it in the final list. In the end, 50 articles, which we refer to as D1 to D50, were selected for analysis. All these articles are from authors with at least two years of experience in software development (we used LinkedIn to check this information) or from reputable blogs (e.g., The Software House), from the official documentation of testing frameworks (such as Jest) or from well-known forums (e.g., Stack Overflow and GitHub Discussions). Table 1 shows the distribution of the documents among these categories. A single source was attributed to each document (the first source that applies, following the order they are listed in the table). For this reason, the sum of the percentages is 100%.

Analysis: The documents selected in the previous step were analyzed by the first author in order to answer the proposed research questions. Essentially, he read each document in detail and whenever he found a discussion about benefits (RQ1), drawbacks (RQ2), best practices (RQ3), architectural components (RQ4), or tools (RQ5) he created a label to denote this discussion (or reused a previously created one) and applied it to the document under

Table 2
Main benefits of snapshot testing.

Description	Documents
Easy to implement	27 (54%)
Prevents regression	19 (38%)
Serializable	9 (18%)
Alternative to UI tests	6 (12%)
Easier and safer refactorings	5 (10%)

Table 3
Drawbacks of snapshot testing.

Description	Documents
Fragility	14 (28%)
Lack of Context	11 (22%)
Large snapshots	8 (16%)
Manual Verification	6 (12%)
Flaky behavior	3 (6%)

analysis. For example, documents commenting on how easy it is to implement snapshot tests received an *Easy to Implement* label, which was used to answer RQ1.

Validation: The labels proposed by the first author were revised and also validated by the second author. Basically, for each research question, he checked whether the documents associated with each label indeed discuss the label's theme. In total, he agreed with 224 classifications. But he also proposed associating 28 labels to documents and also removing 34 labels from documents. In RQ2, the second author also suggested renaming an existing label and adding a novel one. These suggestions were checked and then accepted by the first author.

4. Results

4.1. What are the main benefits of snapshot tests?

Table 2 shows the main benefits of snapshot tests and the number (and percentage) of documents commenting on each one.

The two key benefits are easiness of implementation and the prevention of regressions, the former appearing in 27 documents (54%) and the latter in 19 documents (38%). Indeed, traditional automated tests can require a significant amount of time to design, implement, and define all assertions. On the other hand, snapshot tests make a single default assertion about the state rather than testing particular function results or side effects (as usual for example with unit testing). D3 stated, "*Snapshots with their simplicity can speed up the creation of unit tests (...) snapshot is a single line in comparison to the traditional method*".

As unit tests, snapshot tests are also implemented to prevent regressions. However, the benefit, in this case, connects mainly with the easiness of detecting such regressions. Snapshot testing stores the state of artifacts in a way that can be compared to the current version when changes are made, highlighting the differences between the two. For example, D37's author mentioned that "*it's a really, really easy way to see how your changes are impacting some output*".

The next benefit points out that snapshots enable programmers to compare and check any serializable value, as commented in 9 documents (18%). For example, D40's author states: "*the format of the snapshot is flexible, it can be inlined into the test as a string or stored as a file — it can be a DOM tree, a JSON blob or even an image*". This distinguishes them from traditional tests, where considerable effort is usually required to define the test inputs and outputs.

Our second-to-last benefit, as reported by 6 documents (12%), regards the use of snapshot testing as a faster alternative to end-to-end tests. According to D33, "*(...) I mostly avoid UI tests as they are too clumsy to write and time-consuming to run. Snapshot tests are better for SwiftUI, and very fast*".

4.2. What are the main drawbacks of snapshot tests?

Table 3 shows the main drawbacks of snapshot tests as uncovered in our review.

Snapshot tests need a single assertion, which makes them simple to build, but at a price. The most significant issue found in 14 documents (28%) report that these tests are highly dependent on the outcome of what is being tested, which makes them rather fragile. Therefore, every time a new increment is added to the code, the test will probably fail, resulting in a false positive. D5 in particular stated that "*a snapshot just tells you what the component looked like before and what it looks like now. The decision whether you've fixed a bug, or introduced one, is entirely on you*". Another issue related to the fragility of snapshot tests is the practice of blindly updating the test results, i.e., commonly developers accept the current state of the application without a thorough review. For example, D34 highlights that "*when tests fail, it is very easy to update the snapshots without fixing the code and understanding the failure reason*".

The lack of context when writing a snapshot test is mentioned in 11 documents (22%). Poor descriptions lead to the absence of context while making assertions about snapshots. According to D6, "*we've found that in practice, snapshot tests end up making assertions that aren't clearly represented in the test output. (...) It's not immediately clear whether something is actually wrong, especially for developers who are unfamiliar with the component*".

Another important issue refers to the usage of large snapshots, as commented in eight documents (16%). Large snapshots usually require developers to have a deep knowledge of what is being tested and the associated logic, which makes merge requests and code reviews a more complex task. For example, D3 mentions that "*conflict resolution can be a major hassle — resolving merge conflicts in snapshots is a cumbersome task, especially if the snapshot is considerably large*".

Our second-to-last drawback, as reported by 6 documents (12%), regards the manual verification of snapshot results. For example, D29 mentions that "*the biggest problem with manual verification is that it requires a human in the loop. This slows down the process, and is also potentially error-prone.*".

A final problem refers to the flaky behavior of snapshot tests, mentioned in three documents (6%). Non-deterministic data can result in different values being used every time the test runs. For example, D14 mentions explicitly that "*one of the most frequent (issues) were non-deterministic JSON outputs, for example, when generating random UUIDs. We were frequently facing failures*".

4.3. What are the best practices when using snapshot tests?

Table 4 outlines the best practices developers follow in order to reduce the downsides of snapshot tests. The two key guidelines are the usage of code reviews and treating snapshots as part of the application code, the former appearing in 13 documents (26%) and the latter in 11 documents (22%). Since snapshot testing highlights the differences between two versions of the same artifact, they can be used in code review to compare the differences between merges. D9 in particular states: "*through this process, our team has become more strict on code reviewing these types of tests. We look at exactly what has changed and why. We tend to write a snapshot test as a catch-all for shared React components*".

Due to their fragility, snapshot results should be treated as part of the code, i.e., they should be committed and changed the

Table 4
Best practices to reduce downsides of snapshot testing.

Description	Documents
Use snapshot tests during code reviews	13 (26%)
Treat snapshot results as code	11 (22%)
Write small snapshots	7 (14%)
Detailed test descriptions	3 (6%)

Table 5
Components tested using snapshot tests.

Component	Documents
Frontend	28 (56%)
Mobile	14 (28%)
Backend	4 (8%)

Table 6
Libraries and frameworks for snapshot testing.

Library/Framework	Documents
Jest	27 (54%)
SwiftSnapshotTesting	7 (14%)
FB/iOSSnapshotTestCase	5 (10%)
Enzyme	5 (10%)
React Testing Library	5 (10%)
Cypress	4 (8%)
KotlinSnapshot	3 (6%)
AndroidTestify	2 (4%)
Jetpack Compose	2 (4%)
ScreenshotTestsForAndroid	2 (4%)
Shot	2 (4%)
Mocha	1 (2%)

same way as the source code. D1 states that "*(...) all snapshot files should be committed alongside the modules they are covering and their tests. They should be considered part of a test, similar to the value of any other assertion*".

Our second-to-last benefit, as reported in seven artifacts (14%), regards writing small and focused snapshots to narrow the scope of the tests and to allow developers to check and understand changes more quickly.

As a final guideline, in three documents (6%) we found suggestions to use detailed test descriptions, in order to facilitate the understanding of the test's context by other developers.

4.4. What architectural components are tested using snapshot testing?

Table 5 outlines the architectural components where snapshot tests are commonly used.

As expected, snapshot tests are mostly used in the frontend (28 documents, 56%). In fact, as we mentioned in RQ1, they are seen as an alternative to traditional end-to-end tests. For example, D15 mentions the following about the test of React-based JavaScript frontends: "*snapshot testing has been created due to the need for an easier way to write tests for React components. Many React developers reported that they spend more time writing tests than the actual component. Therefore, snapshot testing allows React developers to quickly generate tests using its simple syntax*".

Snapshot tests also have an important role in mobile development, particularly to check the behavior of displayed layouts or views across multiple devices and orientations. One of such use case is described by D13: "*within our test suite, we utilized a well-crafted loop to iterate through all combinations of the five cell parameters, so that we could quickly and exhaustively generate views for all valid combinations. (...) As the tooling and frameworks mature and experience with the technique grows, snapshot testing will become more reliable and is already a worthy addition to any mobile project*".

Finally, snapshot testing is less prevalent in the backend (four documents, 8%). However, in these components, they can be used to check warning and error messages, as explained by D28's author "*(...) it's a really common case that you want to write a test to ensure that a good error or warning message is logged to the console... Before snapshot testing I would always write a silly regex that got the basic gist of what the message should say, but with snapshot testing it's so much easier*".

4.5. Which libraries and tools are commonly used for implementing snapshot tests?

Table 6 lists the libraries and tools used to support snapshot tests. Jest is the most prevalent testing framework, being mentioned in 27 documents (54%). Typically, Jest is used to write snapshot tests for the frontend components of JavaScript-based software. The second most used tool is SwiftSnapshotTesting, which is mentioned in seven documents (14%).

4.6. Key findings

We can summarize our findings as follows. Snapshot tests are easy and simple to implement (RQ1). Despite that, they are effective in preventing regressions. Particularly, they are more common in frontend components and mobile apps (RQ4). On the other hand, fragility is the key concern among practitioners, i.e., the tests can falsely fail with frequency (RQ2). Regarding best practices when using the technique, it is recommended to include snapshots in commits (RQ3). Snapshots can also help developers to understand the changes subjected to code review. Finally, Jest is the most popular tool for implementing such tests (RQ5).

5. Threats to validity

Due to its own nature, our findings rely on documents that were not subjected to peer review. However, we carefully selected the articles used for answering the proposed research questions, in order to reduce the risks of considering low-quality content. In particular, we selected only articles authored by experienced software engineers or sponsored by known software companies. The analysis of the documents, in order to answer the proposed research questions, was also a manual process and therefore subjected to different results. However, the documents were carefully analyzed by the first author and then reviewed, discussed, and validated by the second author.

The search query's structure represents another possible threat to the validity of our results. For example, there is always a chance that important articles were not included in our selection due to missing keywords. However, to mitigate this threat, we conducted exploratory tests with several queries, in order to select the one with the best combination of keywords. Furthermore, we only considered documents written in English. Therefore, we might have missed important and valid documents in other languages.

6. Related work

To the best of our knowledge, there are currently no academic papers focusing on snapshot testing. Consequently, we divided our related work into two categories: (I) testing, and (II) grey literature reviews.

As mentioned, snapshot testing ensures that the current state matches a certain golden standard. The first reference to this type of testing used the term characterization testing, which was a name coined by Feathers (2004). Characterization tests attempt

to describe how a test object acts and the goal is to keep this object from changing by accident.

Recently, UI testing has become more popular in both industry and academia. Ran et al. (2022) presents a framework, called VTest, for industrial visual testing. They argue that automated UI testing needs to be both applicable on multiple platforms and able to handle non-standard UI elements. The authors describe their experience in employing their framework in a popular mobile application and report an 87% activity coverage which is better than other state-of-the-art tools.

Borjesson and Feldt (2012) presented a comparative study between two UI testing tools, Sikuli[3] and a commercial tool. They compared both tools according to their qualities, how they assist automation of previously manually conducted test cases, as well as the time to design new tests and their execution times. The results show that there are only minimal differences between both tools and, more significantly, that UI testing is a promising technology for automated system testing.

On grey literature review papers, Raulamo-Jurvanen et al. (2017) conducted a review on the criteria used by practitioners to choose the right test automation tool for their projects. They concluded that there is an agreement about the most significant selection criteria for test automation tools, taking into consideration factors such as cost and community support, but that criteria are not methodically utilized by practitioners. Garousi and Mäntylä (2016) presented a multi-vocal literature review, i.e., a review where formal literature is combined with grey literature to answer questions about the whens and whats of software testing automation taking into account multiple factors such as economic viability, software maturity, and regression testing. The authors also provide guidance for the industry as well as a practitioner-friendly checklist.

Sánchez et al. (2022) expanded previous research on mutation testing tools by analyzing and classifying new tools and by mining public repositories to characterize the adoption of such tools in GitHub projects. The authors identified that Infection (PHP), PIT (Java), and Humbug (PHP) are the most widely used mutation tools in recent years. Mutation testing is mostly used to support real software development, followed by teaching, learning, and research projects.

7. Conclusion

Snapshot tests are an emerging software testing approach that rapidly became widely adopted by many and important software organizations. In this paper, we reviewed a total of 50 grey literature documents on snapshot tests. According to our findings, snapshot tests are easy to implement and at the same time, they can be effective to prevent regressions. They are also serializable, which makes their result comprehensible and subject to code reviews. For this reason, they are seen as an alternative to UI tests implemented using tools such as Cypress and Playwright.

However, the simplicity and easiness of implementation come at the price of being fragile and therefore subjected to false positives. To circumvent these problems, it is important that developers treat snapshots as code (for example, they should include their results in pull requests). It is also recommended to write small and focused snapshot tests including detailed and clear descriptions.

For future work, we plan to better characterize the usage of snapshot testing, for example by mining and analyzing a large sample of GitHub projects. In our review, we also missed documents discussing the best guidelines to set up the first version of the snapshot of a given test, which has particular importance in

the case of non-deterministic or flaky tests. Therefore, studying and defining these guidelines is also an interesting line of future work.

The complete list of the documents considered in our review is publicly available.[4]

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The complete list of the documents considered in our review is publicly available at https://github.com/VictorGazzinelli/snapshot-testing.

Snapshot Testing: Benefits and Drawbacks (Original data) (Github)

Acknowledgments

References

Borjesson, E., Feldt, R., 2012. Automated system testing using visual GUI testing tools: A comparative study in industry. In: 5th International Conference on Software Testing, Verification and Validation. ICST, pp. 350–359. http://dx.doi.org/10.1109/ICST.2012.115.

Feathers, M., 2004. Working Effectively with Legacy Code. Prentice Hall.

Garousi, V., Mäntylä, M.V., 2016. When and what to automate in software testing? A multi-vocal literature review. Inf. Softw. Technol. 76, 92–117. http://dx.doi.org/10.1016/j.infsof.2016.04.015.

Kamei, F., Wiese, I., Lima, C., Polato, I., Nepomuceno, V., Ferreira, W., Ribeiro, M., Pena, C., Cartaxo, B., Pinto, G., Soares, S., 2021. Grey literature in software engineering: a critical review. Inf. Softw. Technol. 138 (1), 1–26.

Nakazawa, C., 2016. Jest 14.0: React tree snapshot testing. Retrieved June 27, 2022 from https://jestjs.io/blog/2016/07/27/jest-14.

Panchuk, A., 2022. Snapshot testing: Example and its benefits. Retrieved June 27, 2022 from https://ec2-3-132-53-78.us-east-2.compute.amazonaws.com/blog/snapshot-testing-example-and-its-benefits/.

Ran, D., Li, Z., Liu, C., Wang, W., Meng, W., Wu, X., Jin, H., Cui, J., Tang, X., Xie, T., 2022. Automated visual testing for mobile apps in an industrial setting. In: 44th International Conference on Software Engineering (ICSE), Software Engineering in Practice Track. SEIP, pp. 55–64. http://dx.doi.org/10.1145/3510457.3513027.

Raulamo-Jurvanen, P., Mäntylä, M., Garousi, V., 2017. Choosing the right test automation tool: a grey literature review of practitioner sources. In: 21st International Conference on Evaluation and Assessment in Software Engineering. EASE, pp. 21–30. http://dx.doi.org/10.1145/3084226.3084252.

Sánchez, A.B., Delgado-Pérez, P., Medina-Bulo, I., Segura, S., 2022. Mutation testing in the wild: Findings from GitHub. Empir. Softw. Eng. 27 (6), http://dx.doi.org/10.1007/s10664-022-10177-8.

Smiley, K., 2022. Testing strategies. Retrieved June 27, 2022 from https://github.com/MobileNativeFoundation/discussions/discussions/6.

Zhang, H., Zhou, X., Huang, X., Huang, H., Babar, M.A., 2020. An evidence-based inquiry into the use of grey literature in software engineering. In: 42nd ACM/IEEE International Conference on Software Engineering. ICSE, pp. 1422–1434. http://dx.doi.org/10.1145/3377811.3380336.

Victor Pezzi Gazzinelli Cruz is an MSc student at the Computer Science Department of the Federal University of Minas Gerais, Brazil. His research interests are software engineering, software testing, software maintenance, and evolution.

Henrique Rocha completed his Ph.D. at UFMG, Brazil (2016), and worked as a postdoc fellow at Inria, France (2017–2018) and the University of Antwerp, Belgium (2019–2021). Currently, he is an assistant professor at the Computer Science Department at Loyola University Maryland, USA (since 2021). His research

---

³ http://sikulix.com

⁴ https://github.com/VictorGazzinelli/snapshot-testing

interests include software testing, software maintenance, blockchain-oriented software engineering, and empirical software engineering.

Marco Tulio Valente received his Ph.D. degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG. In 2020, he published the book "Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade" (in portuguese), which is widely used by Brazilian universities.