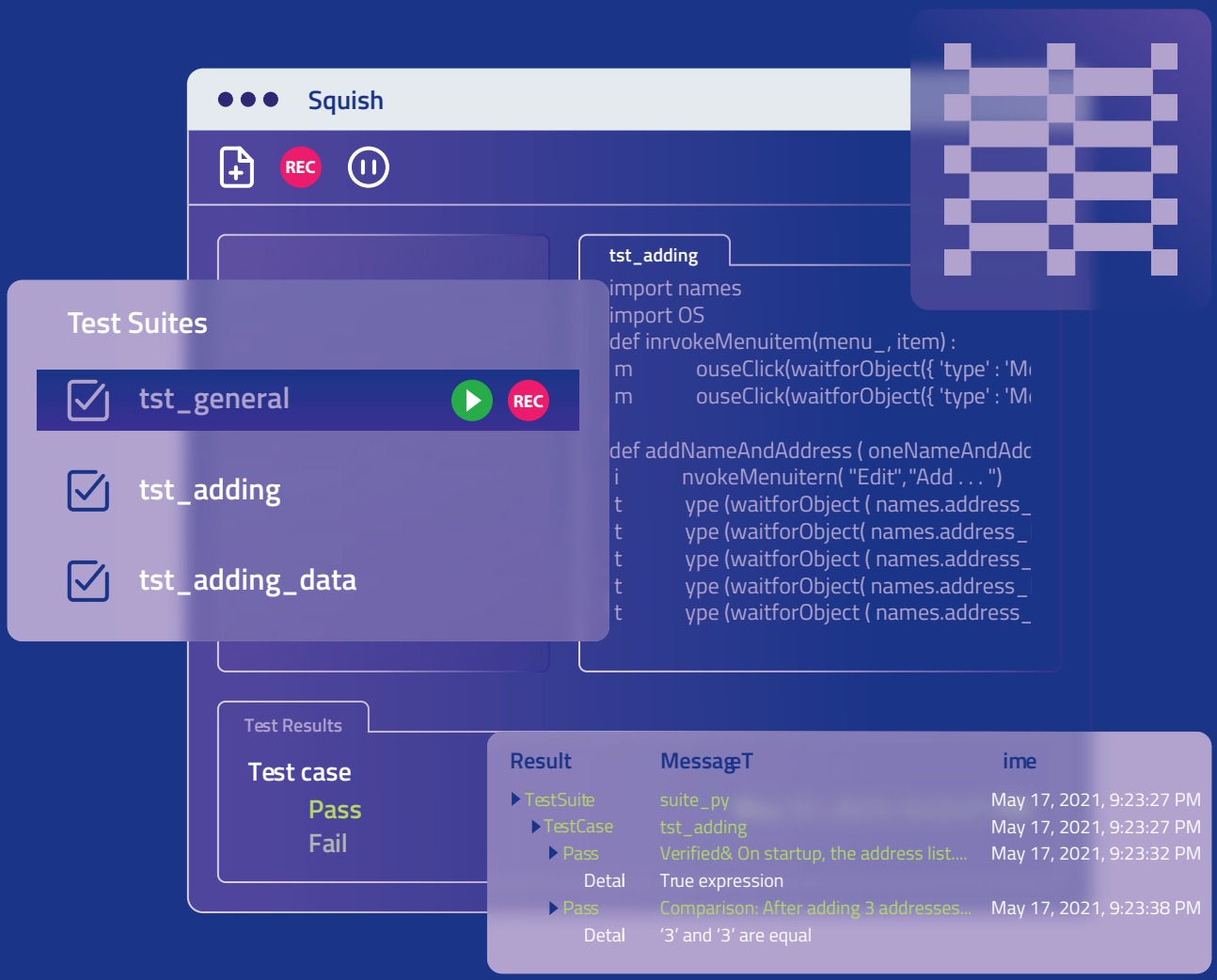
Qt Quality Assurance

WHITEPAPER

High Impact, Low Maintenance:
Test Automation Strategies

What is test maintenance? And what are the best practises when it comes to automated test maintenance? Learn all about strategies and practices to achieve low-maintenance tests in automated GUI testing with Squish.



High Impact, Low Maintenance: Test Automation Strategies

The goal of automating GUI testing is to enhance efficiency by reducing the time spent on manual testing and allowing our human testers to focus on more complex and critical tasks. But, how?

Before we discuss the best practices of automated test maintenance, let's examine the definition of test maintenance.

Understanding test maintenance

Regular maintenance of test scripts, whether manual or automated, is essential to keep them in sync with code changes. This proactive approach not only saves time and costs but also ensures high-quality tests.

For instance, if the identifying attribute of a UI element (such as Id, class, or Xpath) changes, the test must be updated accordingly to prevent failures.

However, without a strategic approach, test maintenance can become challenging and time-consuming. Teams may find themselves constantly fixing broken tests instead of creating new ones or enhancing existing ones.

This is where low-maintenance tests become invaluable.

What are low-maintenance tests?

Low-maintenance tests are designed to be stable and require minimal updates, even when the application under test undergoes changes.

These tests are structured to take a long time to break, meaning they can adapt to application changes without needing frequent and extensive modifications.

The goal is to create resilient test scripts and frameworks that demand less maintenance effort.

5 practical examples in achieving low-maintenance tests with Squish

Strategies and practices to achieve low-maintenance tests in automated GUI testing with Squish.

Here are some strategies and practices by which Squish achieves low-maintenance tests in automated GUI testing:

Practical example #1: Squish records low-maintenance test cases by default

Robust and stable mechanism for identifying GUI elements

The Squish object map and Object Names

Whenever Squish records a test case, it creates an Object Map and adds a symbolic name–a real name pair for every object the user interacts with.

When recording a test case or picking objects in the Spy, Squish automatically creates a name for each object accessed so that it can be identified later, for example, in a test script.

This name could be a multi-property (real) name for most toolkits. These lead to more flexible and reliable object identification in the face of application changes.

Symbolic vs Real Names

The Object Map associates each object name with a so-called symbolic name, which serves as a 'key' into the Object Map. Test scripts exclusively reference the object names by using symbolic names, and the symbolic names are automatically mapped to the referenced object names as a test case is executed.

The Squish object map is designed to make it easier to maintain test scripts when the application under test changes its object hierarchy or names.

Instead of repeating object names in multiple spots of the test script code base, the idea of the Object Map is to maintain a repository of all object names in which each object name is defined exactly once, centrally.

Changes to the application only require changing the single Object Map entry mentioning that object name. The symbolic names are independent of the AUT and, thus don't need any modification.

Hence, all test scripts remain unchanged, and the test cases replay as before since each symbolic name mentioned in the test scripts automatically maps to the new and updated object name.

Key takeaways:

- You know your AUT better than your tool does. Use your knowledge about your AUT to anticipate where AUT changes could happen.
- Increase the abstraction level further
  - Use different properties to identify an object
  - Reduce the number of properties that identify an object

Practical example #2: Increasing the abstraction level by editing Real Names

When recording tests with Squish, each element in the user interface is identified by a set of properties that are unique to that specific element. Properties used to identify elements could be visibility, text contents, parent window, etc.

After recording a test, it may be useful to examine the object map and examine the properties Squish has selected to uniquely identify the GUI elements.

Keep the following in mind when examining the object map:

- Squish may have chosen too many properties, thus making the object too unique. Imagine that you identify a button based on its text contents. This text then changes in the next version of the application under test. As a result, this change breaks your test.
  As a knowledgeable developer or tester familiar with the application being tested, you could have foreseen that the text might change in future versions and could have used various object properties to identify the button uniquely.
- Squish may have chosen too few properties, increasing the possibility of ambiguous real names. For example:
  A button is identified by its location in a specific menu. When a new version of the application under test adds a new button to the same view, suddenly, two buttons match the real name. As a tester/developer, you could have added more identifying properties to the original button's real name to ensure it would stay unique in future updates.

The second example shows that it can be useful to open the object map and examine the different real names to see whether they contain unnecessary properties that can be deleted or lack properties that should be added.

Wildcards can be used to match a real name's text property with multiple different text values for the specific case of text properties.

Key takeaways:

Your AUT changes will break your test at some point. To ease the pain of fixing these breaks, you can:

- Use the object map
- Extract functions
- Separate test logic from test data

Practical example #3: Reducing code repetition

The third practical example highlights the importance of not inlining real names into test scripts.

In most cases, no additional effort is needed because Squish typically uses symbolic names in test scripts when recording, thereby removing the actual names from the scripts.

When manual script writing is employed without the recording feature, it is important to utilize symbolic names corresponding to actual name entries in the object map.

In the test script, you should see:

```
tapObject(waitForObject(names.UIButton))
```

You should not see:

```
tapObject(waitForObject({"type": "UIButton",
"visible": 1, "window": {"type": "UIWindow",
"visible": 1}}))
```

By using only symbolic names in the script, less code repetition is achieved.

Key takeaways:
One major advantage of using object mapping in test scripts is that when changes or updates are needed for an object's properties, you only have to make the modifications in the object map rather than updating every instance of the object reference in the script code. This greatly simplifies maintenance and reduces the potential for errors in the test scripts.

Practical example #4: Extracting methods and sharing code among test cases

In this practical example, we'll look at how the IDE in Squish facilitates the process of creating and refactoring functions by providing tools for standard refactoring. This ensures that the code remains well-maintained.

Additionally, Squish allows for the creation of shared functions, centralizing changes in a single file and ensuring that calls to these functions in different test scripts remain unaffected.

These shared functions can be utilized by any test case within a specific test suite.

1. Screenshot Verification

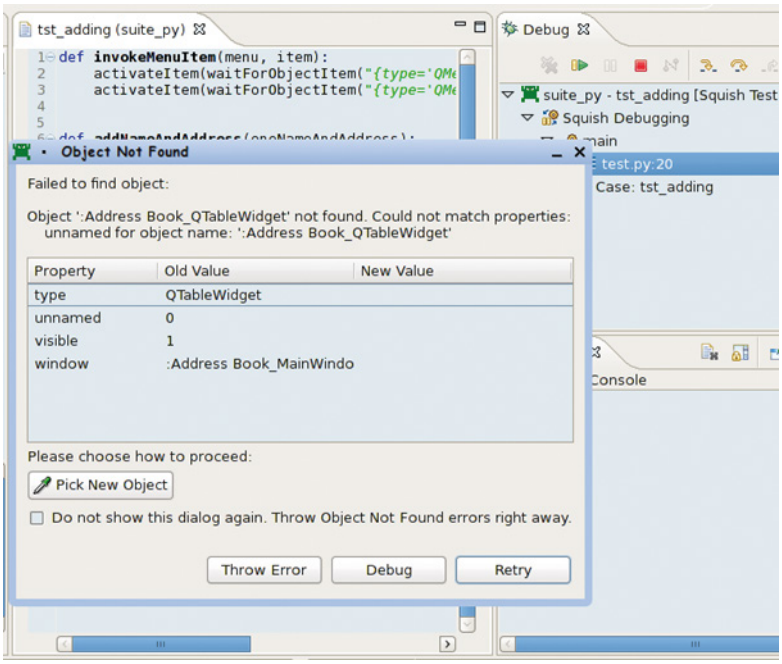
In Squish IDE, the user can use the new screenshot as the "Expected Image." If an application change causes the screenshot to fail, using the failed image as the "Expected Image" adjusts the test script for the new GUI.

Squish also allows the user to choose a different screenshot comparison mode depending on the use case, such as the threshold property, correlation, histogram, etc.

2. Retry mechanism in Object Not Found dialog

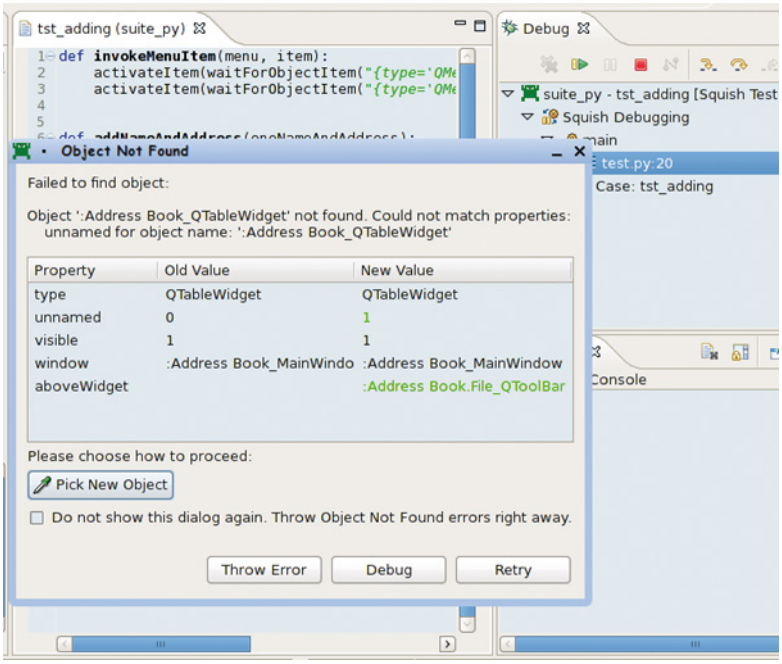
Upon "Object Not Found," Squish IDE allows you to pick a different object. Using the new object name, you can retry running the test script.

If a waitForObject command runs into the specified (or default) timeout during test execution, the Object Not Found dialog automatically opens.



The dialog shows the error message generated for the lookup error and the object name for which the lookup was executed. You can try to solve the error in the following ways.

Click the Pick New Object button to select an object from the AUT and compare its properties against those used for the name. This lets you investigate how the properties have changed and whether they caused the lookup error.



Click the Throw Error button to let the test execution continue and generate the appropriate error. Letting the execution continue at this point may end the test unless the test script catches the lookup error to recover from this itself.

Click the Debug button to close the dialog and open the corresponding name in the object map editor. The Squish IDE is now in the Test Debugging Perspective, so you can use all the usual debugging tools to examine the problem further.

Click the Retry button to re-execute the object lookup. It will use the old name if you did not pick a new object, which can be useful to determine whether the lookup error is triggered because the object takes longer to become ready than the default timeout. If you pick a new object, the new object name will be stored under the symbolic name in the object map, and the lookup will be performed with that new name.

3. Squish doesn't record absolute coordinates in the test script

This makes the recording independent of the absolute positions of the widgets on the screen. The recorded coordinates are not absolute and only specify where a click should occur relative to the top left corner of the widget.

4. Using Smart Wait Mechanisms:

Synchronization waitFor() waitForObject() waitForItem()

Squish has smart waiting mechanisms to handle asynchronous behavior. Instead of using hard-coded sleep statements, Squish provides a plethora of waitFor* API to synchronize the availability of objects to interact with.

The waitForObject(), waitForObjectExists(), and waitForObjectItem() functions default to waiting 20000 milliseconds (20 seconds) for the object they are given to be visible and ready to use. If they time out, they raise a catch-able exception.

Practical example #5 - Data-driven testing

In software testing, it can be beneficial to segregate test data from test logic. This separation can improve scalability and maintainability. It can lower the risk of programming errors, as the test logic remains untouched through changes to the test dataset. This segregation can be accomplished using external data sources like spreadsheets or databases.

The advantage of this approach is that when the test data changes, the external source can be updated without the need to modify the test script.

In Squish, importing a dataset into the system and utilizing it as the source of values inserted into tests is feasible. Squish supports importing data in formats such as .tsv, .csv, .xls, and .xlsx.

The import process can be executed using the Squish IDE or manually via a file manager or console commands. Both methods will be detailed, starting with utilizing the Squish IDE.

Best Practices for Automated Test Maintenance

These practices help ensure that the tests stay up to date, reduce the need for manual work, and maintain high-quality testing.

To keep automated tests running smoothly, following some best practices for upkeep is important. These practices help ensure that the tests stay up to date, reduce the need for manual work, and maintain high-quality testing.

Here are some important practices to keep in mind:

1. Implement version control for your test scripts

Version control systems are like a safety net for managing changes to test scripts. They help teams track modifications, work together effectively, and revert to previous versions if needed.

Essentially, version control ensures that any changes made to test scripts are recorded and can be checked or undone, making it easier to manage and coordinate everything.

By keeping track of changes and saving different versions of your test scripts, you can easily return to a stable version if problems arise. This means that every test script is managed in an organized way and can be updated with confidence.

2. Use codeless tools for test automation

Codeless test automation tools make it easier to create and maintain test scripts. These tools often have visual interfaces and record-and-playback features that reduce the need for deep programming knowledge.

The less technical the test, the easier it is for more team members to help with test creation and maintenance, which improves overall productivity and test coverage. This allows testers, business analysts, and subject-matter experts to create and manage automated tests without relying on specialized automation engineers.

In addition, different QA Engineers have different coding cultures and naming conventions. Visual tools can help avoid confusion and maintain test quality.

3. Continuous Integration/Continuous Delivery (CI/CD) integration

It's important to ensure tests are done automatically whenever the code is changed. It helps developers get quick feedback and reduces the chances of problems happening again. Automating tests in the development process makes sure that the tests always match the changes made to the code.

CI/CD tools are like helpful assistants that run tests whenever code is updated. Having a C⁄CD integration in place is instrumental in detecting issues early in the development process, thereby simplifying the resolution of defects before they escalate into significant problems.

4. Develop test data management strategies

Ensuring reliable and repeatable automated tests requires good strategies for handling test information.

This involves creating sets of high-quality test information that can be used repeatedly. Various methods can be employed to achieve this such as generating new test information, masking real information, or using dummy data to fulfill testing requirements without exposing private details, and maintaining consistency across different test environments.

Proper test data management eliminates inconsistencies, reduces the chances of test failures due to data issues, and provides predictable test conditions.

5. Set up automated monitoring and alerts

Monitoring test runs using tools can quickly identify and catch problems, making it easier to fix them immediately. Automatic alerts can help address issues before they escalate into major problems.

Setting up automated monitoring tools to track test results and performance numbers allows you to be notified if anything deviates from normal parameters—it's one way to stay ahead.

6. Collaborate with others

Teamwork is key to keeping your tests running smoothly. Encouraging communication and knowledge sharing among team members enables them to discuss issues, share solutions, and document best practices.

Regularly scheduled meetings or retrospectives focused on test maintenance between development and testing teams help identify improvement areas and foster a continuous improvement culture.

Enhancing the durability of tests

The two main takeaways from this paper

Squish creates robust, long-lasting tests that reduce the efforts required for maintenance. With strategic practices, you can increase the durability of your tests even further. The two main takeaways from this paper are:

1. **You can increase abstraction levels by:**

- Minimizing object properties - Use the minimal number of object properties necessary to identify GUI elements is crucial. This reduces the likelihood of tests breaking when changes occur in the application under test (AUT). Focusing on the essential properties that uniquely identify each object creates more resilient tests.
- Using Wildcards - Implement wildcards for object properties and values enhances the adaptability of tests. Wildcards allow for minor variations in property values, accommodating changes in the AUT without causing test failures. This flexibility is vital for maintaining robust test suites.
- Leveraging domain knowledge - You know the AUT better than any tools do. Utilize your in-depth understanding of the AUT to refine the object properties stored in the real name. Identify or remove irrelevant properties or modify them to more stable ones that uniquely identify elements. By doing so, you ensure that the tests are both precise and resilient to changes in the application's interface.
- Utilizing intelligent Wait functions - Replace hardcoded wait intervals with intelligent wait functions like waitForObject() to avoid unnecessary delays and reduces the potential for test failures due to timing issues. This function waits for an object to become available before proceeding, ensuring that the test only advances when the AUT is ready.

2. **You can ease the work of fixing a broken test by:**

- Using symbolic names to reference your real names - Use symbolic names to reference real names in your test scripts is essential for maintainability. Symbolic names act as aliases for the actual object properties, allowing you to update the real name in one place. The update is automatically reflected across all references in your test scripts, significantly reducing the effort required to fix tests when changes occur in the application.
- Extract repeated script code into a separate function - By modularizing your test scripts, you minimize redundancy and centralize common actions. When a change is necessary, updating the function in one place ensures consistency across all tests that utilize that function, making maintenance easier and less error-prone.
- Use the data driven approach - Separate test data from the test logic. Store test data in external files and use these files to drive test parameters. This approach minimizes changes to the test scripts themselves and reduces the likelihood of introducing errors during script updates. It also allows for easier scaling and updating of test data without modifying the underlying test logic.

These practices will significantly enhance the durability and maintainability of your automated tests. After all, isn't the goal of automating GUI testing to allow our human testers to focus on more complex and critical tasks?

Qt Quality Assurance

www.qt.io/squish