froglogic

Adaptive Automated GUI Testing

Producing Test Frameworks to Withstand Change

Abstract

Although QA and Development managers may see many challenges in creating an automated GUI testing framework, it is possible to make a test framework that is both effective and maintainable. Challenges a QA or Development manager may consider include:

- How can I ensure the framework provides additional value?
- Can automated GUI testing work with the kind of software I test?
- How can I ensure the test framework effectively detects regressions?
- How can I ensure the framework remains useful as my product evolves?
- Will ongoing maintenance take too much time?

Solutions exist for all of these questions, but it does require strategic planning, participation from all departments, and buy-in from all team members.

This article will describe and give examples of some considerations to help design a test framework that produces adaptive tests and enables teams to keep up with the evolution of today's software industry.

Trend

QA managers faced a wide range of challenges in recent years:

- Influx of rapid application development methodologies (e.g. Agile, extreme programming)
- Expanding product lines, often on multiple technology platforms
- New QA automated testing frameworks and technologies

Some of these trends have been both good and bad for QA. For example, Agile places significant emphasis on testing. According to Belatrix, an industry expert in Agile development and testing, "Agile development integrates testing into the development process, verses having it as a separate phase. Testing therefore is an integral part of the core software development and actively participates though out the software coding process."* However, both Development and Quality Assurance departments have scrambled to adapt to these new methodologies. Many have rushed their testing cycles while also attempting to verify multiple packages concurrently.

At the same time, many new QA testing frameworks have become available. This provides teams with new, powerful tools. However, these new tools and methodologies come with new expectations, especially for automated GUI testing.

Incorporating automated GUI regression testing into an ever-evolving product, and throughout the entire product lifecycle, requires not only the buy-in and continuous efforts of both Development and Quality Assurance, but also an automated GUI regression test framework capable of adapting and growing with your product.

Problem

While adapting to new development methodologies, software companies around the world are also trying to determine how to best ensure any regressions are quickly identified, while also ensuring the quality of each new feature before the package hits the hands of consumers.

Because testing now occurs throughout the application life cycle, QA teams face new challenges including: shorter testing cycles, often with overlapping timetables, as well as accommodating a variety of feature or code changes across each supported platform. Maintaining sufficient testing coverage for each package and package variation creates significant challenges.

Some QA and Development teams have used automated GUI testing as an aid to regression testing for many years. However, depending on the automated GUI regression test framework implemented, the automated tests can either help or exacerbate the problem by creating a test framework maintenance nightmare.

QA and Development teams need the ability to supplement their manual and unit testing with reliable, rapid and repeatable tests. The challenge is to create an automated GUI regression test framework that works with your product as it exists today, and is flexible and intelligent enough to continue working as your product evolves. The magic solution, requiring zero maintenance, no matter the changes in your application, does not exist. And high-coverage, rapidly created, automated GUI tests simply don't have the flexibility and intelligence to evolve with your product.

So, what solutions do exist? The remainder of this article will present methods and tools for implementing an adaptive automated GUI regression test framework. That is, if you're dedicated and willing to put forth the effort now!

Solution Options

Correctly implementing an adaptive automated GUI regression test framework requires careful planning and alignment with your organizational and software product needs. For example, if the software within your organization is build on a cross-platform application and UI framework such as Qt or Java, then the automated GUI testing tool must also support the corresponding framework(s) and operating systems. Consider also, the volume and variety of data the software exchanges, or how frequently objects shift or dynamically display within the UI. Each consideration impacts the automated GUI regression testing tool features required to implement your test framework without restricting your testing coverage and result-producing capabilities from the gate. Settling for partial-product support weakens the effectiveness of the entire undertaking before the project even begins.

When examining your software, ask the team which components of your application can benefit most from automated GUI testing. froglogic's blog discusses "Where to start?", directing implementers to repetitive tasks and script modularization.** Capture additional ideas by reviewing existing manual test cases and procedures, or defect trends. HP examines how to identify candidates for automated testing in their article "Best practices for implementing automated functional testing solutions". The author recommends, "focus[ing] automation efforts on critical business processes" as well as "repetitive tasks".***

Other implementation-critical considerations:

- ✓ Record-and-playback with refactoring capabilities
- ✓ Data-driven capabilities
- ✓ Object-level data validation
- ✓ Test execution order
- ✓ Third-party integration capabilities
- ✓ Complete command-line interface
- ✓ Ability for Development team to extend or adapt
- ✓ Native support for all platforms
- ✓ Ability to interact with controls outside your software toolkit
- ✓ Non-invasive interaction with your software
- ✓ Exception and Error Handling
- ✓ Test debugging capabilities
- ✓ Distributed testing support
- ✓ Test file format for versioning and portability
- ✓ Cross-platform testing
- ✓ Script maintenance and handling application change

Options Applied

One of the highest impact factors when implementing an automated GUI regression test framework: Script maintenance and handling application change. The key with automated GUI regression testing is not only how well it can regression test your application today, it’s if those same tests will work tomorrow, and the effort required to in maintain and expand the test framework going forward.

As a product evolves, so does the software design, features, layout or even workflow. Creating and maintaining modularized tests, capable of quickly adapting to such changes, increases the longevity of each test, set of tests, and test framework.

Take for example a Customer Relationship Management (CRM) solution: Often highly configurable, with dynamic layouts and using business-rules based security and workflow. Depending on the modules available to the signed-in user, the order and presence of visible components may vary. Writing a test, which navigates to a specific component, regardless of other available components, and confirms the component displayed as expected, represents a single basic test.

Apply the aforementioned example to the following industry-standard approaches:

Approach 1: Image-based Recognition

Image-based Recognition captures images and actions corresponding to each step performed while. In some tools, variations of the images can be captured in an effort to overcome minor graphical changes, which would otherwise cause the tool to fail to locate the object(s) of interest.

Approach 2: Object-based Recognition

Object-based Recognition captures key properties for each control on which an action was performed, as well as the action performed. Captured objects and a select set of their properties are stored in a repository (often called an object repository or object map) using a unique name derived from the object. Any test can use the repository, and in some tools, the repository is available across more than one test suite or collection of tests. Future recorded tests first check for the existence of the object in the repository and only add a new entry should it not exist.

During initial record and playback, both Approach 1 and Approach 2 execute without modification or issue.

Due to a recent re-branding of the CRM tool, the list of components, while still present, have taken on an 'edged' versus 'rounded' shape among other minor graphical design adjustments

Approach 1:

During scheduled test playback, the Image-based Recognition tool fails to locate the component clicked in the original recording. The results are viewed the morning after the nightly execution of tests completed, and the test is assigned to a team member to update. The team member re-runs the test, witnessing the issue, and captures new images for the test, accommodating the change in UI design.

Approach 2:

During scheduled test playback, the Object-based Recognition tool completes the test without issue.

Note the purpose of the test was not to validate the look and feel of the UI, but to confirm the functionality; performing a specific set of actions navigated the to the expected component within the application. Had the purpose of the test been to validate the graphical design of the application, both tests would require updating the expected result to the new UI. Two very different tests.

Multiple components, approximately 10, in the CRM tool received alternate names to accommodate shifting trends in CRM terminology. Teams are prepared and aware of the change, and update the tests accordingly.

Approach 1:

Using the Image-based Recognition tool, the team captures new images for the test, and once again, the test is running without issue.

Approach 2:

Using the Object-based Recognition tool, the team opens the repository, updates the label property associated with the changed component to reflect the new naming convention. The test remains untouched, and runs again without issue.

Now imagine this single component was used throughout an entire test suite. Referenced hundreds of times.

Approach 1:

Using the Image-based Recognition tool, locate all instances of the component used in the test suite and update each instance to point at the newly captured image.

Approach 2:

Using the Object-based Recognition tool, the team opens the repository, updates the label property associated with the changed component to reflect the new naming convention. The all tests remain untouched, and run again without issue.

The team has now been tasked to create a test that generates a new support ticket, retrieves the support ticket ID and date/time stamp, modifies the new ticket by adding a comment, and verifies the ticket ID remains unchanged, the creation date displays in the expected format and matching the data/time stamp retrieved upon ticket creation, and the last modified status contains the expected result.

Approach 1:

Using the Image-based Recognition tool, images are captured of the actions and Optical Character Recognition (ORC) is used to interpret and retrieve text from a user-specified location in each image containing text to be used as data. The interpreted text is later compared to a new OCR interpretation. In some tools, an area for capture must be defined. The defined area will pull all text visible. Should the desired text fall outside of the defined range that text will not be included, and should other text appear within the range, that text would also be included in the capture. This creates a potential for error when retrieving data from the AUT, as the target cannot be limited to the precise object of interest. In the event the date initially captured, while the same date, appears in an alternate format, a combination of OCR and string parsing must be implemented both for the initially captured date and the date for comparison.

Approach 2:

Using the Object-based Recognition tool, steps, objects and properties are captured. Verification points, or checkpoints, are recorded, identifying the objects containing the ticket ID and date/time stamps. The test is modified after the recording is complete to save the verification point object values, and compare those values with basic scripting logic to the corresponding data later in the script. In the event the date initially captured, while the same date, appears in an alternate format, a datetime class available for example in Python, can be used to parse dates, and compare the appropriate date components. Furthermore a regular expression, or again a datetime class, can be used to confirm each date displays in the format expected. Simple lines of script, that even those not familiar with the scripting language in use, can search the web (assuming the scripting language isn't proprietary to the tool) and find an answer within seconds. Changes to the date format are a simple change to the scripting logic.

Forgot to mention, this tool is cross-platform, and tests must run on Windows, Mac and Linux.

Approach 1:

Using the Image-based Recognition tool, assuming the tool is capable of running natively on Windows, Mac and Linux, OS-specific images are captured for each of the alternate operating systems and operating system variations. A separate set of tests must be either maintained for each OS and OS variation, or logic is incorporated into all scripts to use the OS-specific images for each step.

Approach 2:

Using the Object-based Recognition tool, assuming the tool is capable of running natively on Windows, Mac and Linux or Unix, no changes are required, and tests run against each OS and OS variation without issue.

Options Compared

These scenarios illustrate how dramatically efforts differ in the image-based and object-based recognition approaches following initial test creation.

Although the initial test creation began with roughly equal effort, maintenance efforts between approaches quickly deviated, even with minor application changes.

The chart below compares the anticipated maintenance time needed using the two approaches for each scenario:

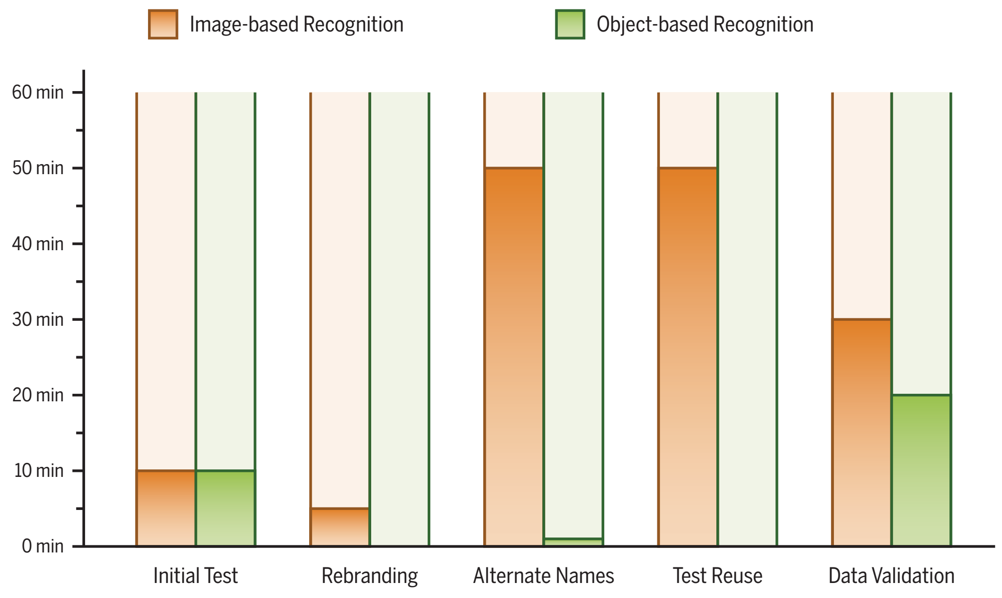

Figure 1: Maintenance Effort Duration

Note also that the Data Validation scenario would potentially require even more effort when using Image-Based Recognition due to the stability and accuracy challenges inherent in extracting application data using image-based recognition.

Looking at the data in "Figure 1: Maintenance Effort Duration" , a two-test scenario, undergoing three relatively simple software application changes would require an estimated 31 man-hours using the Object-based Recognition approach vs. an estimated 145 man-hours using the Image-based Recognition approach.

Expand that collection of tests from two to one hundred, and you're now facing an automated GUI regression test framework whose maintenance can cost you either 3,100 man-hours or 14,500 man-hours given three updates to existing tests per release cycle. Now multiply that by ten similar iterations per year, and you see how dramatic the difference is.

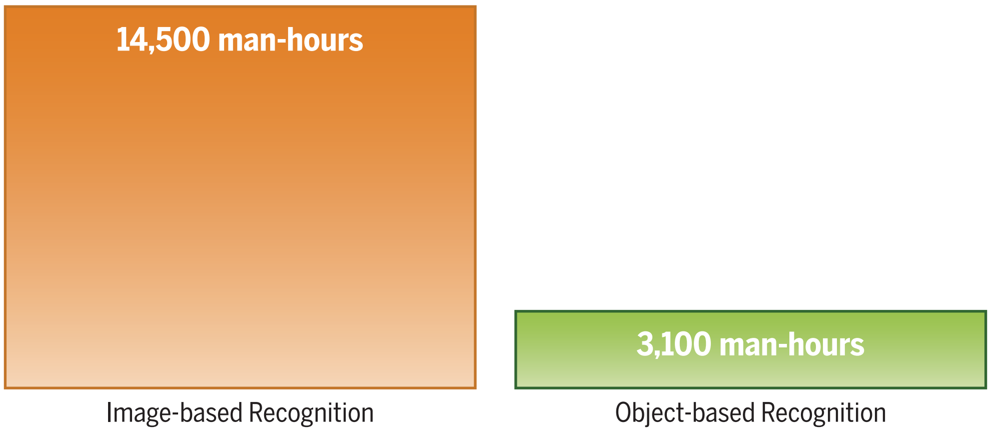

Figure 2: Maintenance Effort Variation

In the end, the object-based recognition approach requires just 21% of the maintenance time that would be required using the image-based recognition approach. This difference greatly outweighs any benefits that might come from creating tests slightly more quickly using the image-based recognition approach.

Conclusion

To produce an automated GUI test framework that interacts with your software and is better able to adapt to changes, QA and Development managers need to focus on object-based recognition approaches. Using partially-supported or image-based approaches will only result in dramatically more pain, maintenance, and re-work.

A test framework that's aware of your application and all corresponding toolkits, with the technology necessary to accurately and effectively interact with your application is one of the most critical factors that determine a QA team's ability to create and maintain a robust and stable test framework. The ability to quickly and easily adapt tests for change from a central location, as well as modularize and share test segments and test logic all impact maintenance time.

*See how **Squish** produces an automated GUI regression test framework that withstands change.*

Evaluate today!

Now is the time to identify the critical workflows and business processes of your software, and implement tests for repetitive tasks. You can use varying data to drive repetitive tasks, validating a variety of scenarios. You can build your framework with the future in mind; extracting and organizing tests to optimize re-use. You can document and collaborate, sharing the test framework throughout all phases of the application lifecycle, accessible from a centrally located and versioning capable portal. You can educate across teams, demonstrating how and why the test framework will help at each phase, reducing commonly overlooked regressions, and validating critical application paths are operational following the latest code or environment change.

* "Agile Testing Best Practices." Whitepaper: Agile Software Testing. 2013. Belatrix Software Factory. 19 Nov. 2013 <http://www.belatrixsf.com/index.php/whitepaper-agile-software-testing>.

** "Squish Tip of the Week: Where to start?" Froglogic's blog. 12 Nov. 2013. Froglogic. 20 Nov. 2013 <http://blog.froglogic.com/2013/11/squish-tip-of-the-week-where-to-start/>.

*** "Best practices for implementing automated functional testing solutions." Five keys to automating QA testing. Nov. 2012. Rev4. HP. 19 Nov. 2013 <http://on.hp.com/LP=2826>.