Agile Testing at Scale

Mark Fink

Author contact: mark@mark-fink.de

Abstract

"Agile Testing at Scale" is an important topic in Agile software development of enterprise business applications. More and more companies adopt Agile development methodologies which introduce much shorter release cycles. Frequent releases push requirements regarding continuous integration and test automation.

This talk introduces methods on how large automated test suites are structured and maintained efficiently.

At scale, test automation will eat up a significant part of your development teams project budget and if it is not running effectively you will not even gain the desired benefits. Consequently you need to set goals and metrics so you can easily keep your quality initiative on track and / or being able to make adjustments to the life cycle of your project at any point.

For software development projects which have not yet introduced it the continuous integration tool chain really opens up the next level of software quality. Continuous integration ensures that the application can be build and integrated at any given point of time. Feedback to developers via email regarding integration problems will be given immediately after a code change is applied. So by introducing continuous integration big massive integration problems are reduced to a few tiny ones.

Successful test automation does not only depend on processes, tools, and infrastructure. It is also necessary to enable the person who writes the testcases in the first place. I will introduce the ACC methodology on how to approach writing tests. I will also discuss a styleguide for writing new tests.

I will round up the paper with internet resources that you can access to download a continuous integration server and configuration that demonstrates all relevant test automation aspects on a working Javascript application.

Biography

Mark Fink runs an independent consultancy providing software testing services. In 2013 Mark released his book "The Hitchhiker's Guide to Test Automation". The book summarizes his practical experience with software testing methodologies and tools in the context of agile web application development. His combination of academic qualifications with 20 years' practical experience have made him a popular speaker at international conferences on software test automation and performance testing.

1 Introduction

Many aspects need to be considered for an agile software development project to be successful in test automation. Continuous integration is a precondition and needs much consideration in addition to installing another tool. Another example is the integration of the test-related tasks into the agile software development process. Of course all these important topics can not be covered within a single conference paper. They could fill books. In fact I wrote one myself: "The Hitchhiker's Guide to Test Automation". So in this paper I will focus on the aspects that I think need the most attention when it comes to "Agile Testing at Scale".

With scale I mean:

- the scope of the project is in the area of 100 man-years or more
- the team consists of 5 developers or more
- high velocity - new features and changes are added to the code base at high pace (the above mentioned developers are frequently adding changes to the code base)

In a situation like this it is crucial for your test initiative to have methods in place and tool support for writing and maintaining tests. This topic is covered in section 2 "Writing Tests". Section 3 "Evolving Automated Testing" covers an approach on how you structure your tests in a way so that they stay efficient and effective. I give examples where ever possible. Section 4 "Installing the Continuous Integration Server and Demos" gets you started with the continuous integration server and a sample application.

2 Writing Tests

This section is aimed at writing automated tests, tooling and styleguides.

Successful test automation not only depends on processes, tools, and infrastructure. It is also necessary to enable the PEOPLE who write test cases in the first place. Consequently this chapter introduces the activity-testing pattern for approaching test-writing and elaborates on the environments necessary for writing. We also discuss a styleguide for writing and / or maintaining tests.

2.1 Activity-Testing Pattern

When implementing GUI regression tests, it is a good idea to start implementing them in a top-down fashion all the way from the intention of the test to the technical activities [ADZ2010].

The activity-testing pattern recommends describing the test and the automation at the three levels depicted in Figure 1 (an example is provided for each level).

Table 1 provides definitions and examples for each of the three abstraction levels of GUI test automation.

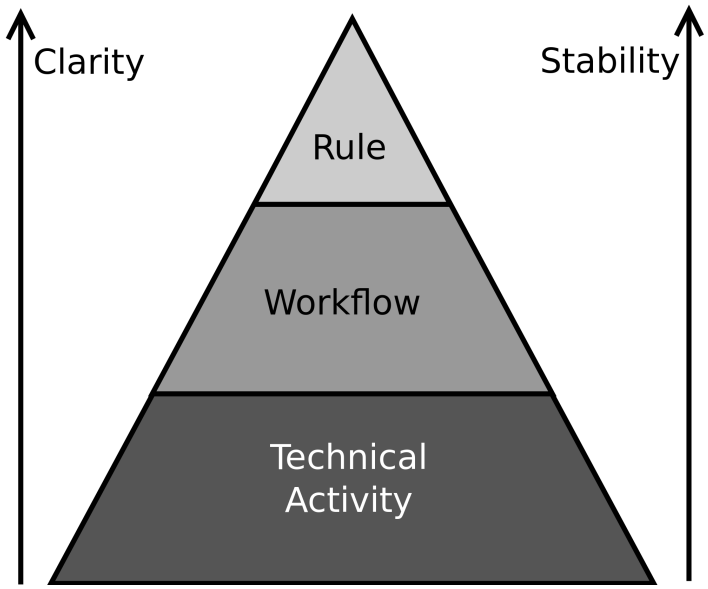

Figure 1: Three levels of abstraction in GUI test automation.

Table 1: GUI test-automation abstraction levels.

Level	Description	Example
Business Rule / Functionality	What is the intention of the test?; what is it demonstrating or exercising? A good start when implementing tests is to use examples gathered during specification of the story regarding the feature in question	Free delivery is offered to customers who order two or more books
GUI Workflow	What does a user have to do to exercise the functionality through the GUI on a higher activity level?	Put two books in a shopping cart, enter address details, verify that delivery options include free delivery
Technical Activity	What are the technical steps required to exercise the functionality? On this level the API of the automation tool (in this case Selenium) is used. There is no need to wrap the features of the automation tool!	Open the shop homepage, log in with 'testuser' and 'password', go to the '/book' page, click on the first image with the 'book' ID, wait for page to load, click on the 'Buy-now' link and so on

FitNesse's main purpose is to provide a means to build the abstraction levels described above. Robert C. Martin has compiled an excellent series of video tutorials on how to achieve this using FitNesse [MAR2008]. The video *Writing Maintainable Automated Acceptance Tests* is particularly relevant in this context.

2.2 Page Objects

Page objects are a layer that guards you against changes in application frontend technology. This can be very useful if you need it but it comes at a cost, too. Using page objects it is also possible to fence yourself against the technology used for browser automation. But the same mechanics apply here. Unless you need the flexibility to exchange the automation technology, I advise against taking on the cost. I observe a trend toward maximising the use of pattern and abstraction layers. These bring additional complexity and costs and you should only consider using them if you foresee significant benefits in the near future.

The page-objects pattern has the disadvantage of structuring the writing of test cases in a bottom-up way. This leads to an overly technical focus instead of a focus on the test intention

and workflow steps relevant to a user story.

I strongly recommend using the activity-testing pattern. This is a way superior concept, structuring testing activities in a top-down fashion. The activity testing pattern works well with the S/M/L scopes (see *Evolving Automated Testing*) that provide a model for structuring your test automation. If you have doubts, I recommend reading Eric Evans' book [EVA2004].

2.3 Standardised Working Environment

To lower the entry barrier on test automation for your colleagues it is very important that you provide a standardised environment for running tests, analysing test results, improving tests or writing new test cases. Your environment might consist of test tools, IDEs, source code and test code repositories and the like. You have to make sure that installing and using this environment is as easy as possible.

I found it good practice to hold everything related to the standardised working environment in a repository. Everybody in the team (including analysts) knows how to work with a repository, so upgrading the test environment is only a few clicks away. In one project, the team developed multiple applications in parallel. I have split the working environment from the test suites, so one can work with multiple test suites within the same working environment.

Of course some people have different requirements for their working environment, just as developers have different IDEs or different clients for the source repository. But they benefit from the low entry barrier provided by a standardised working environment. This way they have a smooth start and can integrate the test tools into their customised workflow later once they have seen how everything goes together. They just pick what they need from the standardised working environment.

By maintaining a single working environment, you can reduce need for maintenance and make sure it works smoothly. Broken test tools are not exactly what you need when building a reputation with your team.

2.4 Staging Environment

Test the test! Your automated test suite, wall display, and working environment become essential components to support your development process after a little while. Now if you change anything, such as maintaining or adding a test or refactoring some test function, you might break that development process. If the test automation reports a failure, it must be in the application or infrastructure. If the test automation reports false positives, this becomes an issue. Once you have to deal with lots of false positives, developers will start to ignore the results.

This means you have to strictly separate the test-automation runtime that supports the development process and test executions that you need to test the test. In other words, you need a dedicated environment to test the test.

Newbie Test Suite

> All new and changed tests get tagged as "Newbie". They do not run with the regular test suite. So if they fail, a tester needs to look into them. Their failure does not show on the wall display. The Newbie Suite runs multiple times a day and can be triggered manually. So within two days we have more than ten executions of a new test and we get a first indication of whether there are timing issues we must fix immediately. If after 10 or more executions a new test case turns out to be stable, we promote it to the test suite.

Staging Environment

> Once you have a test suite of significant size, you cannot just run the whole test suite on your tester's machine. Neither can you check in the refactored test code. If you have changed anything in the test infrastructure, such as upgrading to a

new Selenium version, you need to make sure everything works well before you promote the changes to your development process. This is why you should plan for a staging environment from the beginning. Without it, you cannot improve your test infrastructure without compromising the acceptance of the test automation.

When sizing the hardware for your test infrastructure, you have to plan for newbie and staging environments as well. If you share the environments with the regular test execution, you run the risk of slowing down test execution considerably, which in turn could lead to acceptance problems. Section *hardware* goes into calculating the sizing for test infrastructure.

2.5 Styleguide for Writing Tests

Styleguides are very specific to your company, team and application, so I can only provide you with a common styleguide as a starting point. Update your styleguide regularly to address issues you face in your day-to-day work.

Naming Convention

The naming convention is an important aspect of your test-automation initiative. It heavily influences the maintainability of your test suite. If you need to find test cases for regression testing or for adding verification steps for new features, it is important that you find them quickly.

I consider the following aspects to naming tests:

- Business rule
- Workflow
- Sub-Application
- Feature
- Dialog / Panel
- When naming the test case think "How would I 'google' for this?"
- If you have similar tests, highlight the differences
- Test intention
- A naming example for a test for the Supercars application would be "SupercarEdit"
- Do not use project or change request numbers as part of the name

If I create new test cases I usually complete the documentation section first before I define the name for a test case.

Documentation

I start each test case with a documentation section. It is difficult at first to find the right amount of documentation (the right abstraction level) necessary. Reviews by colleagues can help here.

Aspects relevant for test-case documentation are:

- Document the intention (the business rule) of the test case with a few sentences
- Do not repeat the test-case name
- Document which features or functionality you intend to test
- Document the expected results (validations)
- Do not repeat the complete set of test steps

Journal of Changes

Document recent changes to the test case. There's no need to journal that you fixed a typo, but significant changes should be recorded.

Test-Case Blocks

Often it makes sense to structure tests into multiple blocks (sections):

- If the test consists of multiple sections, then add some documentation / headlines on each section
- Make explicit which sections are preparation, which are closing / cleanup, and which is the relevant test block

Timeout

Timeouts are important to manage the execution time of your test suite. You do not want to wait forever for something that is missing.

When setting timeouts in tests, consider the following aspects:

- Use reasonable general timeouts (60 seconds)
- Use specific timeouts for steps that are known to take longer. Set back to the general timeout at the end of the step

Abort Conditions

If you are testing on test environments with real backends and databases such as (integration - or system - test environments), you will experience unexpected downtimes and failures in these systems. It is essential that you make yourself immune to these issues. If you do not prepare for these issues, your suites will run forever and you will have difficulty finding out why. Abort conditions are the right tool to manage execution time for larger test suites.

Following is some guidance on using abort conditions:

- Abort when condition is not met
- Abort if some of your test infrastructure is missing (e.g. Selenium server)
- Abort for situations such as "500 system is not available", or user login failed
- Abort when not on the right page
- Abort as soon as possible to save valuable testing time

Locating HTML Elements

There are different ways to locate HTML elements. If you do HTML element location wrong, your tests will eat up lots of resources. Use element IDs or CSS selectors whenever possible. Try to avoid using XPATH altogether.

You should agree with your developers that HTML elements need useful IDs or name attributes assigned. In my projects this is an acceptance criterion that falls into the category of testable application.

Waiting for Events

Always wait for specific events. For example "wait for element visible" is much more reliable than "wait 500".

Tagging of Test Cases

Tags are applied to individual test cases. The tags are used orthogonally to the test suite structure (see *structure-test-suite*). You can use the tree structure and tags as additional filter criteria to include or exclude specific tests.

Some applications for tagging test cases are:

- Tagging a test for specific environment (DEV, INT, SYS)

- Tagging new tests for newbie environment (NEWBIE, ...)
- Tagging for specific purposes (POWERON)

If you think this styleguide lacks something important, please let me know so I can add the missing information.

3 Evolving Automated Testing

In an Agile development process, automated testing is an ongoing activity that is interwoven with the development process.

In this section I introduce the concept of small / medium / large scopes that as a mental model will allow you to keep your test suites effective and maintainable while your application evolves over time.

3.1 Complexity

Complexity is an important topic in software development and consequently, in software testing. Scientists from various disciplines such as Mathematics, Biology and even Engineering have made huge advances in the last few years in understanding complexity, but we have not yet come to a solid understanding of how complexity influences engineering and software development processes in particular. Nevertheless, we understand that complexity drives efforts in software development and also creates the risk of failure. For this reason, in test automation, we set ourselves the goal of reducing complexity by replacing large test cases by multiple test cases of medium or small scope that are easier to maintain and faster to execute.

Google proposed classifying test automation into three scopes of large, medium and small tests. The former Google tech-leader Prof. James Whittaker explains this approach as determining testing scope by "emphasising scope over form" [WHI2012].

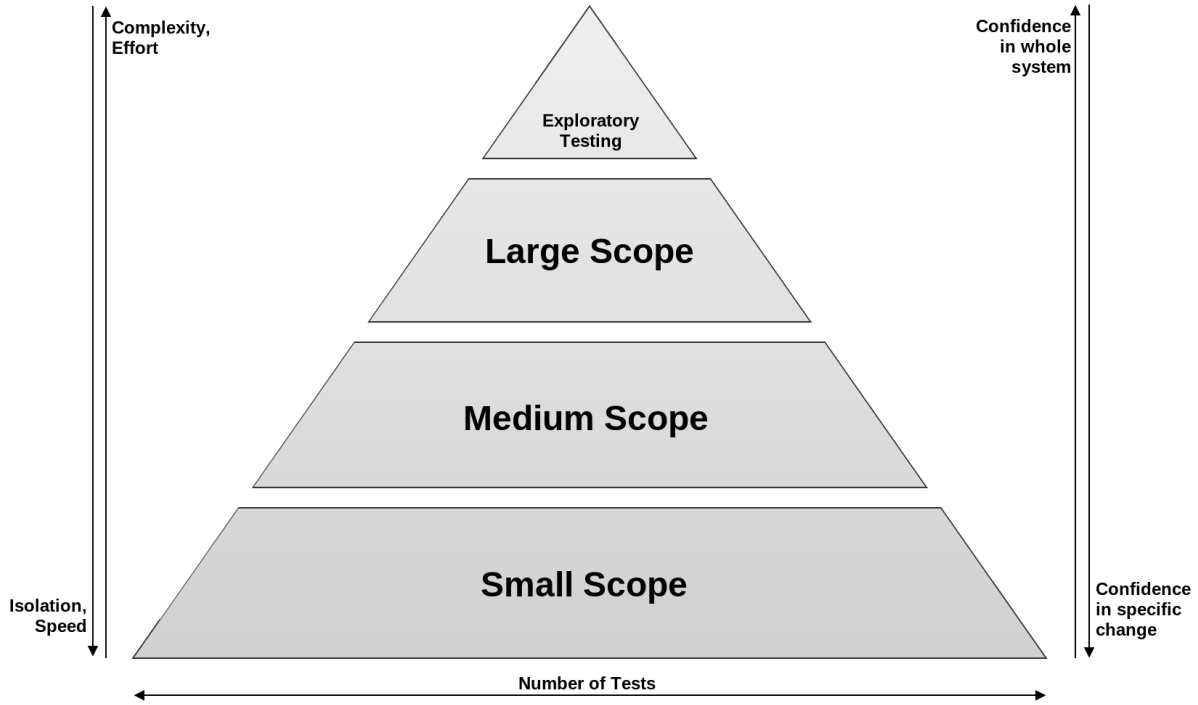

Figure 2: The testing pyramid consists of exploratory testing and large, medium, and small test-automation scopes.

Figure 2 shows how the different scopes in test automation relate to each other in the testing pyramid. The figure also contains a scope for exploratory testing, a subject covered in Part IV of the book.

3.2 Large Scope / Acceptance Tests

Large Scope tests exercise the entire application in an end-to-end fashion, typically through the GUI. This scope should be limited to checking for catastrophic errors. The question large scope tests attempt to answer is, "Does the application operate the way the user would expect?" Execution time for large tests is on the order of minutes or hours.

Table 2: Classification of Large Scope.

Benefit	**Weakness**
Test the most important aspects of the application	Nondeterministic (flaky) results because of dependency on external systems and test data
Test external subsystems	Broad test scope makes result analysis difficult and drives analysis efforts
	Time-consuming test data setup
	A high level of operation often makes it impractical to exercise specific corner cases. That's what small tests are for
	Long running
	Resource-intensive

Use large-scope tests sparingly! This is important for keeping your test-automation effort on track and ensuring the test suite stays maintainable. A word of caution: if your testers think it is best to test everything in the large layer because it is the "best" way to identify defects, you are soon destined to suffer test-automation congestion. Your test suite will become ineffective and inefficient. This means it simply takes too long to run and due to its flaky nature, it requires a lot of analysis to decide whether results point to a problem.

3.3 Medium Scope Tests

Medium Scope tests exercise interaction between components. They may have system dependencies or run multiple processes. The question medium-sized tests attempt to answer is, "Do near neighbouring functions interoperate the way they are supposed to?" They run on the order of seconds or minutes.

Table 3: Classification of Medium Scope.

Benefit	**Weakness**
Loosened mocking requirements and runtime limitations provide developers with a stepping stone that can be used to move from large tests to small tests	They can be non-deterministic because of dependencies on external systems
Tests run in a standard developer environment, so developers can run them easily	They are not as fast as small tests
They run relatively fast, so developers can run them frequently	
Tests account for interfacing with external subsystems	
Testing of configuration variants. Because the tests run fast, you are able to run thousands or even hundreds of thousands of variants, something that is prohibitive in the large scope	

Mocking technologies are often used to make components testable by taking out technical infrastructure such as message queues and email-based workflows.

3.4 Small Scope / Developer Tests

Small tests are very finely grained and exercise code within a single function or class. No external resources are involved (maybe some data files). Implementation of small tests requires mocks and fake databases. Developers often use small tests when diagnosing a particular failure. The question a small test attempts to answer is, "Does this code do what it is supposed to do?" Small tests execute on the order of milliseconds or seconds.

Table 4: Classification of Small Scope.

Benefit	**Weakness**
Lead to cleaner code because to be testable, methods must be relatively small and tightly focused; mocking requirements lead to well-defined interfaces between subsystems	Do not exercise integration between modules. That is what the other test categories do
Because small tests run quickly, they can catch bugs early and provide immediate feedback after code changes have been made	Mocking subsystems can sometimes be challenging
They run reliably in test environments	Mock or fake environments can get out of sync with reality
They have a tighter scope, which allows for easier testing of edge cases and error conditions, such as null pointers	
They have focused scope, which makes analysis of errors very easy	

Generally speaking, there is no need for exact classification criteria with the intent to achieve a sharp distinction between the three test scopes. On the contrary, we do not want to have a sharp separation, since we want to use this as a mental model to establish a consistent migration tendency for test automation from large to medium and small scopes.

3.5 How to Use the Model of S/M/L Scopes

If you are going to add new tests to your test suites, add them as low in the testing pyramid as possible.

Keep an eye on the number of large scope tests you use. If you count too many, identify areas where large scope tests can be replaced with one or more medium / small scope tests. In this way, you keep your automated tests in shape while your application evolves over time.

Within Google, project teams strive towards well-balanced test suites with an approximate goal of about 70% small, 20% medium and 10% large tests.

4 Installing the Continuous Integration Server and Demos

For "The Hitchhiker's Guide to Test Automation" book I created tools and demos with installation scripts and all necessary configuration. The Supercars application and tests are intended for to demonstrate all the different test automation aspects for a web application. The whole installation procedure should take no more than half an hour, depending on your Internet bandwidth and the machine you are working with.

Overview of the book's Github Repositories

The tutorial files and demo software can all be found at *http://github.com/markfink*. The files are contained in git repositories.

Relevant Github repositories are:

- supercars - the Supercars application

- fitnesse_jukebox - tutorial on using FitNesse
- SelRunner - using FitNesse for browser automation
- tutorial_ci - Jenkins plus test automation runtime
- tutorial_jasmine - tutorial on using Jasmine for testing the jukebox sample

Further installation details and installation procedures you can find at https://github.com/markfink/tutorial_ci.

5 Conclusion

The paper provides a brief overview on the topic of "Agile Testing at Scale" and reference to more detailed information. The paper does not provide an exhaustive documentation on the topic or technologies used for the implementation of test automation. Its intention is to reach people interested in software testing, in order to enter into fruitful discussions on software testing and test automation methodology. The acompanied installation scripts and demo applications aim to foster adaption into a variety of project environments.

References

[ADZ2010] Gojko Adzic, "How to implement UI testing without shooting yourself in the foot", http://bit.ly/9OoUf6

[MAR2008] Robert C. Martin, "FitNesse video tutorials", http://bit.ly/tBfUQ

[WHI2012] James Whittaker, 2012, *How Google Tests Software*, Addison Wesley