

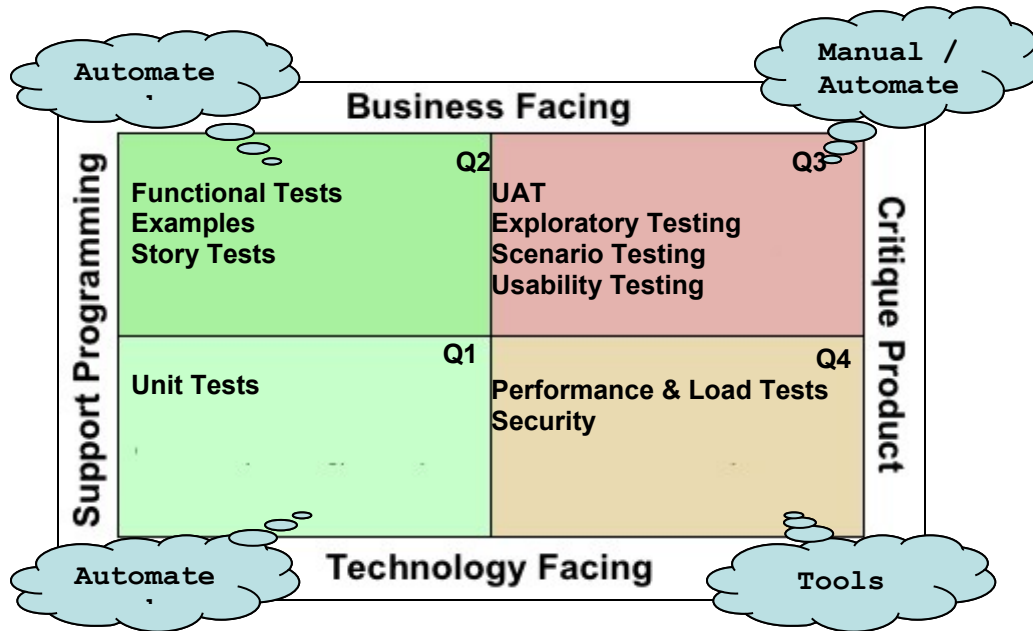
Chapter 9+

Summary of Testing Quadrants

Review of the Testing Quadrants

In Chapter 3, we introduced Brian Marick's testing quadrants, and in the next few chapters we talked about how to use these concepts in your agile project. In this chapter, we'll bring it all together with an example of an organization that used many of the techniques we have discussed.

Figure 9.1
Agile Testing Matrix



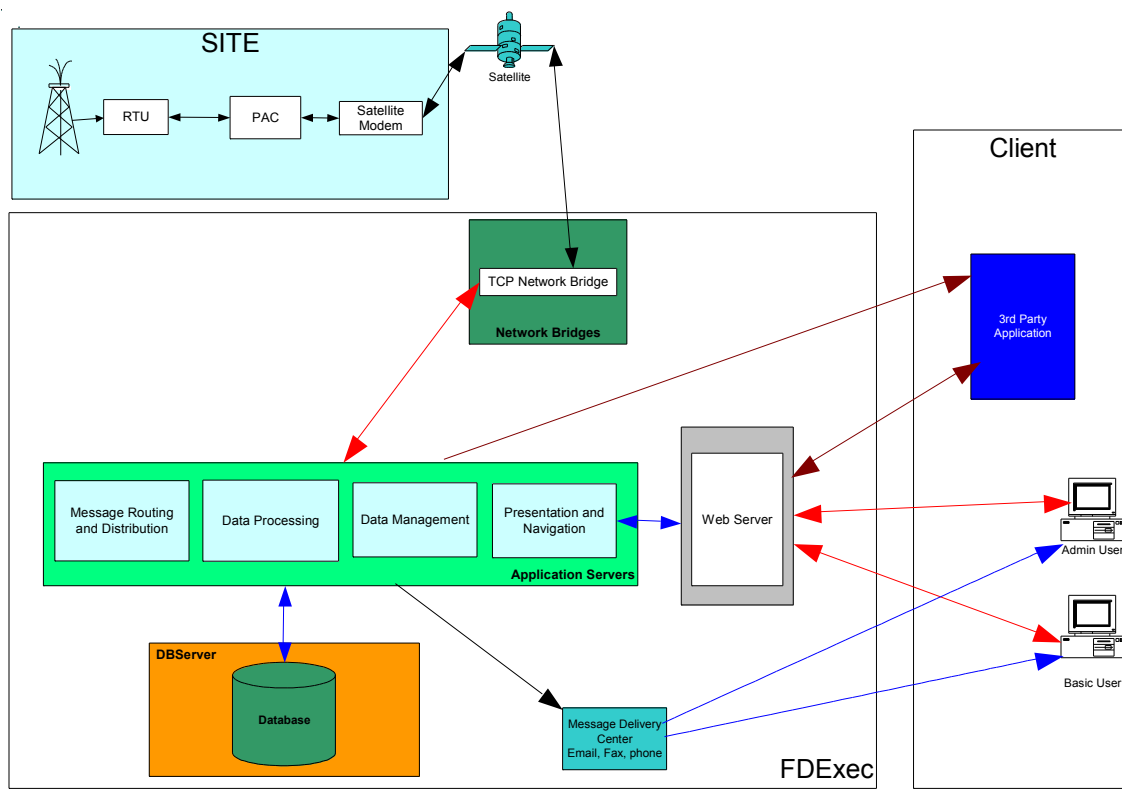
When to Use Which What

We've just spent six chapters talking about each of the quadrants and some of the possible tools you can use. The next trick is to know when to use them and how to put them altogether. This chapter is devoted to an example showing many different types of tests. We'll give you the example and talk about how it fits into the quadrants.

A System Test Example

The following story is about one organization's success in testing their whole system, using a variety of home grown and open source tools. Janet worked with this team, whose main test architect was Paul Rogers. This is his story.

The system solved the problem of monitoring remote oil and gas production wells. The solution combines a remote monitoring device that can transmit data and receive adjustments from a central monitoring station using web services over satellite communications. The measurement devices on the oil wells, Remote Terminal Units (RTU) use a variety of protocols to communicate with the device. This data from the RTUs was transmitted via satellite to servers located at the client's main office. It was then made available to users via a web interface. A notification system, via email, fax or phone is available when a particular reading is outside of normal operational limits. A JMS feed and web services are also available to help integration with clients' other applications.



Unit Tests

Unit tests are technology facing tests that support programming. Those that are developed as part of test-driven development not only help the developer get the story right, but also help to design the system.

The programmers bought into Test Driven Development (TDD) and pair programming whole-heartedly. All new functionality was developed and tested using pair programming. All stories delivered were supported by unit tests, and very few bugs were found after coding was complete. The bugs that were found were generally integration-related.

However, when the team first started, the legacy system had few unit tests to support refactoring. As process changes were implemented, the developers decided to start fixing the problem. Every time they touched a piece of code in the legacy system, they add unit tests, and fixed the code as necessary. Gradually, the legacy system became more stable and was able to withstand major refactoring when it was needed. The power of unit tests!

Acceptance Tests

The Product Engineer (the customer proxy) took ownership of developing the acceptance tests. These varied in format depending on the actual story. Although he struggled at first, the Product Engineer got pretty good at giving the tests to the programmers before they started coding. The team created a template that evolved over time that met both the programmers' and the testers' needs. The tests were sometimes informal, but they included data, and required setup if that wasn't immediately obvious, different variations that were critical to the story, and some examples. The team found that examples helped clarify many of the stories as to what was expected.

These tests were not automated when first delivered with a story, but were automated by the test team as soon as possible. Sometimes tests were automated post-development, but usually the test team finished the automation before development on the story was complete. Of course, the Product Owner was available to answer any questions that came up during development.

These acceptance tests served three purposes. First, since they were given to the programmers before they started coding, they were business facing tests that supported development. Secondly, they were used by the test team as the basis of automation that would feed into the regression suite and future ideas for exploratory testing. The third purpose was acceptance testing. The Product Engineer used these tests as the basis for his acceptance of the story.

The Automated Functional Test Structure

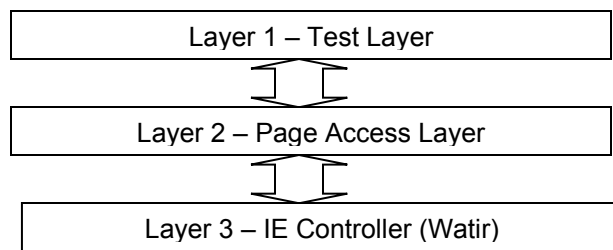
The automated test code included three distinct layers. The lowest layer was Watir and other classes, such as loggers which wrote to the log files.

The second layer was the page access layer, where classes that contained code to access individual web pages lived. For example, in the application under test (AUT) there was a login page, a create user page, and an edit user page. Classes written in Ruby contained code that could perform certain functions in the AUT, such as a class that logs into the application, a class to edit a user and a class to assign access rights to a user. These classes contained no data. For example, the login class didn't know what username to login with. Layer 3 supplied that data.

The top layer, Layer 3 was the test layer, and contained the data needed to perform a test. It called 2nd layer classes which in turned used the first layer.

Layer Two also knew how to handle the error messages the application generated. For example, when an invalid username was entered on the login page, the login class detected the error message, and then passed the problem to the Test Layer.

This means the same Layer Two classes could be used for both happy path testing and for negative testing. In the negative case, Layer Three would expect Layer Two to return a failure, and would then check to see if the test failed for the correct reason, by accessing the error messages that Layer Two scraped from the browser.



The functional tests used Ruby with Watir to control the DOM on the browser, and could access almost all the objects in the page. The automated test suite was run nightly on every new build.

Exploratory Testing

The automated tests were simple and easily used by the whole team. Single tests could be run to set up specific conditions, allowing effective exploratory testing to be done without having to spend a lot of time manually entering data.

The team performed exploratory testing to supplement the automated test suite, and get the best coverage possible. The human interaction found issues that automation cannot find. Usability testing was not a critical requirement for the system, but the testers watched so that the interface made sense and flowed. The testers used exploratory testing extensively to critique the product by the testers. The Product Engineer also used exploratory testing for his solution verification tests.

Embedded Testing

In addition to the web interface, the RDM system consisted of a small embedded device that communicated with measuring equipment using various protocols. Using Ruby, various tests were developed to test part of its administrative interface, which was a command line system similar to FTP.

These data-driven tests were contained in an Excel spreadsheet. A Ruby script would read commands from Excel (again using the OLE interface), and send them to the embedded device. The script would then compare the response from the device with the expected result, also held in the spreadsheet. Errors were highlighted in red.

While this provided a lot of test coverage (the automated test took approximately one hour to run, and doing the same tests manually would take eight hours) it didn't actually test the reason the device was used, namely, reading data from RTUs.

A simulator was written in Ruby with a FOX (FXRuby) GUI. This allowed mock data to be fed into the device. Since the simulator could be controlled remotely, it was incorporated into automated tests that exercised the embedded device's ability to read data, respond to error conditions, and generate alarms when the input data exceeded a pre-determined threshold.

Embedded testing is very technical, but with the power provided by the simulator, the whole team was able to participate in testing the device. The simulator was written to support testing for the test team, but the programmer for the firmware found it valuable, and used it to help with his development efforts as well. That was a positive unexpected side effect.

WebServices

Web Services were used by clients to interface with some of their other applications. The development group used Ruby to write a client to test each service they developed. For these tests, Ruby's unit testing framework, Test::Unit, was used.

The web services tests were expanded by the test team to cover over 1000 different test cases, and took just minutes to run. They gave the team an amazing amount of coverage in a short period of time.

The team demonstrated the test client to the customers, who decided to use it as well. However, they subsequently decided it didn't work for them, so they started writing their own tests but in a much more adhoc fashion using Ruby. They used IRB, the interactive interface provided by Ruby, and feed values in an exploratory method. Much of their User Acceptance Testing was done this way.

Three different methods of testing serving three different purposes, but all stemmed from the need of a developer to test his code.

Testing Data Feeds

As shown in **Figure xx**, the data from the system is available on a JMS queue, as well as the web browser. To test the JMS queue, the development group wrote a Java proxy. It connected to a queue, and printed any arriving data to the console. They also wrote a Ruby client that received this data via a pipe, which was then available in the Ruby automated test system.

Emails were automatically sent when alarm conditions are encountered. The alarm emails contained both plain text email, and email with MIME attachments. The MIME data contained data useful for testing, so a mime decoder was written to provide access to this data.

The End-To-End Tests

From the beginning, it was apparent that correct operation of the whole system could only be determined when all components were used. Once the simulator, embedded device tests, web services tests and application tests were written, it was a relatively simple matter to combine them to produce an automated test of the entire system. Once again, Excel spreadsheets were used to hold the test data, and Ruby classes were written to access the data and expected results.

The end-to-end tests were complicated by the unpredictable response of the satellite transmission path.. A pre-defined timeout value was set and if the test's actual value did not match the expected, the test would cycle until it matched or the timeout was reached. When the timeout expired, the test was deemed to have failed. Most transmission issues were eliminated this way.

Reliability

Reliability was a critical factor of the system, since it was monitoring remote sites that people couldn't get to easily. The simulator that was developed for testing the embedded system was set up on a separate environment, and was run for weeks at a time measuring stability of the whole system.

User Acceptance Testing

User Acceptance Testing (UAT) is the final critique of the product by the customer. In this example, the customer was in France, thousands of miles from the development team. The team had to be inventive to have a successful UAT. The customer came to work with the team a couple of times during the year, so was able to interact with the team a little easier than if they had never met him.

After the team introduced agile development, Janet went to France to facilitate the first UAT at the customer site. It worked fairly well, and the release was accepted after a few critical issues were fixed. The team learned a lot from that experience.

The second UAT sign-off was in-house. To prepare, the team worked with the customer to develop a set of tests the customer could perform to verify new functionality. This was not done in isolation. The customer was able to test the application along the way, so UAT didn't produce any issues. The customer came, ran through the tests and signed off.

We cannot stress the importance of working with the customer enough. Even though the Product Engineer was the proxy for the customer, it was crucial to get face time with the actual customer. The relationship built over time was critical to the success of the project. Janet strongly believes that the UAT succeeded because the customer knew what the team was doing along the way.

Reporting the Test Results

Although comprehensive testing was being performed, there was little evidence of this outside of the test team. The logs generated during auto tests provided good information to track down problems, but were not suitable for a wider audience.

To raise the visibility of the tests being performed, the test team developed a logging and reporting system using Apache, PHP and MySQL. When a test ran, it logged the result into the database. A web front end allowed project stakeholders to determine what tests were run, the pass/failure rate and other information.

Documenting the Test Code

During development, it became clear that a formal documentation system was needed for the test code. The simplest solution was to use RDoc, similar to Javadoc, but for Ruby. RDoc extracted tagged comments from the source code, and generated web pages with details of files, classes and methods. The documents were generated every night, using a batch file and available to the complete team. It was easy to find what test fixtures were created.

The documentation of the test code helped to document the tests, and make it easier to find what we were testing, and what the tests did. It was very powerful and easy to use.

Summary

This example demonstrates how testing practices from all four agile testing quadrants are combined during the life of a complex development project to achieve successful delivery. The experience of Janet's team illustrates many of the principles we have been trying to emphasize. The whole team, including developers, a test architect, testers, a customer proxy and the actual customer, contributed to efforts to solve automation problems. They experimented with different approaches. They combined their homegrown and open source tools in different ways to perform testing from the unit level all the way through end-to-end testing and UAT. The success of the project demonstrates the success of the testing approach.

- Choose or create tools that solve the problem
- Don't forget the customers when planning your testing needs
- Don't forget to document ... but only what is useful
- Think about all four quadrants of testing throughout your development cycles
- Use lessons learned during testing to critique the product to drive development in subsequent iterations