Cyreath.co.uk

Empirical Pragmatic Testing

Specification by Example

An Experience Report

A paper by Mark Crowther, Empirical Pragmatic Tester

Table of Contents

Overview

Inspired by Gojko Adzic's 'Let's Change the Tune' presentation, Mark set about applying the idea combined with his own approaches and techniques, to a project delivering a currency trading platform.

As the project had already commenced the approach needed to get testers working with developers, Business Analysts, Architects and others quickly and effectively. With the project intended to be over several phases test artefacts needed to remain relevant and useful.

In this Experience Report Mark will describe how he applied Specification by Example and helped fundamentally shift the way project activities were performed across all teams. He'll outline the use of Concordion and Java, the nature of test cases, the use of Illustrative & Key Examples and how many traditional testing practices become even more effective in the context of Specification by Example.

The presentation took place at the Skills Matter Exchange in central London on December 7th 2010. This paper reflects the content of the presentation which was recorded and can be viewed by visiting the Skills Matter website at: http://skillsmatter.com/podcast/agile-testing/specification-by-example-an-experience-report

Introductions

Mark Crowther

Mark has been involved in testing and QA for around 15 years. The first role where Mark was exposed to QA practices was in manufacturing QA for an electronics company making security camera systems. The next role took him to a manufacturer of power supplies and mobile phones where he implemented their ISO 9000 system.

Leaving there Mark then joined AOL (UK) as the head of testing in the UK and so commenced the current career in software testing. After just around 6 years with AOL he then moved onto other companies including Electronic Arts and lastminute.com to head their testing departments.

He describes his approach to testing as neither traditional nor agile, but a 'middle way' that draws on elements of QA and testing from manufacturing and both traditional and agile software development. Mark can be contacted on Twitter or Skype as *MarkCTest* or via his website at www.cyreath.co.uk where he maintains a blog, templates and papers.

Background to the Project

Project, Technology, Challenges

Mark was assigned as Test Manager for a project developing the service layer for an FX currency trading platform. The Conceptualisation phase had already ended which meant the project had moved into the Elaboration phase where the direction of the project delivery, test approach, team size, tools and technology, etc. were being agreed.

Components and technology being used in the project included IIS, Active Directory, Oracle DB, Diffusion from Push Technology, Fatwire's Content Server and IBM's Websphere Application Server along with Java as the main development language.

A host of challenges arose straight away regarding the test approach and management:

- The test team would be based in Leeds, but the customer was in London where the Test Manager would be based half the time.
- The test team would be developer-testers, not career testers. They'd be more likely thinking like developers not testers so needed an approach that supported their developer mind-set.
- Tests would need writing in a way that allowed easy automation which would be directed at testing functionality of the service layer components at their level, not through the UI.
- There was no specific test management tool, such as Quality Centre, in which to manage test cases, meaning a way to easily access and work with the test cases needed to be found.
- The Client had made it clear they did not want to be reviewing masses of test cases. They did want to understand the tests in context of their requirements along with test coverage and status.
- Test status against development needed to be known on a per build basis and tests needed to help development understand requirements and functional specification more easily.
- The final challenge was one of communication, all tests needed to be understood by non-testers such as the Technical Architect, Developers, Project Managers and the Client – i.e. the whole project team.

It was clear that an agile like approach to testing was needed, something that allowed just enough documentation to be created, that supported speedy delivery of usable automated tests and helped show test status and progress, while at all times remaining accessible to the entire project team.

Overview of the Test Approach

Based on previous experience Mark immediately proposed the use of Concordion in place of a heavyweight tool for test case management such as Quality Centre. The added benefit would be that, if done correctly, the test documentation would be *Live Documentation* and could become an *Active Specification*.

It was agreed that automated tests would be written in Java as this was the application development language being used. Having both tests and code in Java also meant closer collaboration between the development and testing functions would be possible.

It was decided the Gherkin, Given-When-Then form of test case structure would also be used, anticipating how this would support the writing of automated tests later on. In addition this would allow authoring of test cases in a more Client accessible format, lowering communication barriers.

Testing would use a combined approach of executing automated Examples and following up with Exploratory Testing sessions applying Test Heuristics and Memes. This was especially important as component integration took place and more system level functionality began to emerge. It was also the mindset that Examples became regression 'checks' once executed and not finding bugs, thereby making Exploratory Testing of essential value.

Reporting would be a combination of daily updates emailed to project management and highlighted in the morning stand-up with weekly reports summarising the test effort and current Concordion test status.

Experience Report – Specification by Example

Each aspect of the Specification by Example model, discussed in full in Gojko Adzic's book of the same name, was used within the project. The temptation is to say they were used 'by the test team' but as we'll see this would be incorrect. The experience report is given from the perspective of the test team but the entire project team worked with each aspect of Specification by Example to varying degrees.

Collaborative Specification

The concept of Specifying Collaboratively is that all functions of the project team should have an opportunity to collaborate in understanding and defining the criteria by which they'll recognise the software solves whatever the business problem the Client has.

This activity of acceptance implies that it's possible to demonstrate the application does what it's required to do and so we set Acceptance Criteria that we can demonstrate achieving. The issues that arise from this apparently simple approach include who sets the criteria and whether they are up for discussion and whether they have been defined clearly and completely enough.

Initially there was a desire expressed to take a typical Test First approach which would see the test team write test cases against acceptance criteria and then have development work from those. The complication here was that requirements didn't have acceptance criteria stated and a number of requirements were complex and ambiguous.

To help address this, the project's Technical Architect had written a Functional Spec based on the requirements and in doing so had taken steps to clarify their meaning and make them less ambiguous.

From the test perspective there is always risk in this (re)interpretation of requirements within a Functional Spec or any other document for that matter. A better way would be for the requirements to express their acceptance criteria directly.

This situation then was a great opportunity for Collaborative Specification. As James Bach mentions in Lessoned Learned in Software Testing a powerful technique to use alongside Reference and Inference when writing test cases is Conference.

The key here is that Collaborative Specification was done in conference with other project team members including the Client. The test team set about writing tests that captured the explicit or implicit acceptance criteria stated in the requirements. The intention was not just to allow a Test First approach but to help the Client and technical Architect better articulate and understand the requirements.

How was it done?

Collaborative Specification in conference with others was firstly done between the test team members; Gojko refers to this in his book as the *Three Amigos* approach. Three testers would take a section of the requirements and the current Functional Spec and disappear into a brainstorming meeting. One would play scribe, another be at the white-board and the final member would have a copy of the spec and follow along.

As the tester at the whiteboard read through the requirements and Spec they drew them up on the whiteboard. Flow diagrams, mind-maps, questions and question marks, dead ends and possibilities were all captured. Where test conditions were clear the scribe wrote them up straight away, where anything wasn't clear they were captured as queries and questions to take back to the project team and Client.

In conjunction with the test lead other project team members and Client then undertook essentially the same process but at all times the conversations were in terms of understanding functionality and related acceptance criteria - not about testing. The test cases being written were already being referred to in terms of examples of functionality and behaviour, not test cases.

The outcome of this Collaborative Specification activity was a set of Test First looking test cases that were actually a set of Examples of functionality and behaviour. The process was repeated and as each section of original requirements was reviewed in this way it allowed the Functional Spec to be re-written in a simpler and more accurate manner.

Outcome

In using just one aspect of Specification by Example we had clarified and disambiguated requirements, help define clear acceptance criteria and enabled a more succinct and effective re-writing of the Functional Spec. We'd also got a robust set of Examples to develop and test against without talking in terms of test cases at all.

Illustrating with Examples

When following a more traditional test approach test cases are commonly step-by-step scripts containing sections such as preconditions, steps, data and expected outcome of the test. Testers will apply design techniques such as Boundary Value Analysis and Equivalence Partitioning. They'll apply analysis techniques such as Failure Modes and Effects Analysis - though might not know it's called that.

The outcome of this test case design and analysis session is a raft of test cases intended to cover each test condition and provide thorough coverage of requirements. The problem is this traditional test case authoring process is flawed and inefficient.

Testers taking this approach will write as many test cases as possible, often ending up with hundreds of very similar tests or tests that are covering many low risk conditions. What's more they are a re-interpretation of the Requirements and will contain a lot of assumptions and conditions of little interest to the Client. Conditions of interest to the Client being those that show the software is solving their business problems. Scores of edge conditions may be interesting to testers but they're often not to a Client.

It's worth restating that test cases which are a re-interpretation of Requirements, often through the filter of a Functional Spec that is itself an interpretation of those Requirements introduces;

- Test case being once or twice removed from the original Requirements
- Risk of misunderstandings and interpretation errors
- Language of the artefacts becomes more technical and less Client accessible
- Administrative burden to maintain, update, re-publish and correlate artefacts

Consider also that the project which is the subject of this paper was being delivered in an agile, fixed price manner. It was understood that the BJSS mantra of 'necessary and sufficient' was to be thoroughly applied. Writing masses of test cases was not an option. An approach needed to be found that allowed for the authoring of only the necessary test cases in a way the rest of the project team could work with.

How was it done?

As part of following the Specification by Example model it was decided that not only would test cases be written using the Given-When-Then format, but not every possible test case would be written. This approach surprised the Client and some work had to be done to explain how sufficient test coverage would be achieved.

During each Collaborative Specification session the participants would look to write a set of Concrete Examples in the Given-When-Then format. The Concrete or Key examples were to be considered examples of the behaviour that the system should enact. The Then statement in each Example was the acceptance criteria replaced the typical expected outcome of a test case.

Each Requirement typically had several Key Examples to illustrate various pieces of functionality that it covered. Successful execution of a Key Example would then be used to demonstrate the achievement of the acceptance criteria for that functionality.

An example of the functionality from the project included the ability of a user to perform analysis on a given currency pair and include in their query dates and projections. A Key Example that was derived from this is given in figure 1 below. It should be noted that the test was of the correct type of data in the range expected being returned, not of testing the correctness of the returned results.

4.3.6 FX Market Implied Distribution Analysis

Requirements 6866, 6867 and 6868

4.3.6.1
GIVEN: The user StevenMason-White@domain.com has successfully logged in.
AND: User is authorised to use the Market Data Analytics service
WHEN: User performs *FX Market Implied Distribution Analysis* command
AND: enters the Currency pair EURNOK
AND: the start date is 2010-06-01T00:00:00Z
AND: the end date is 2010-07-01T00:00:00Z
AND: the projection end is 1Y
AND: the projection interval is 1M
THEN: Function returns Historic rates and Projection correctly

Show XML result

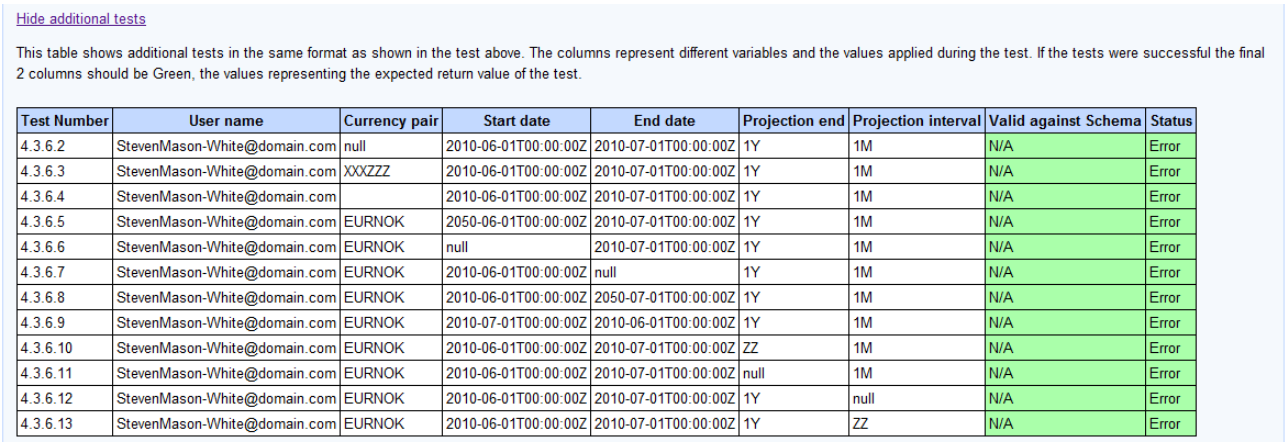
Show additional tests

Figure 1: Key Example of functionality within the Distribution Analysis component

In practice around 4 or 5 Key Examples were found for each piece of functionality. In common test terms this meant the happy path and roughly 2 in bound and 2 outbound edge conditions. The edge conditions typically being replaced with syntax type checks (valid or invalid data) where the functionality wasn't related to numerical values. Clearly there were also occasions where the functionality was much simpler and fewer Key Examples were produced.

A keen tester would now be thinking about all the other test conditions that might exist with the above Key Example and be considering that so far we've only looked at positive test conditions.

Here is where we changed the model slightly by introducing Illustrative Examples. In context of the above these were additional test conditions that while valid were less important than the Key Examples. They could be seen by clicking the 'Show additional tests' link and expanding the section.



Hide additional tests

This table shows additional tests in the same format as shown in the test above. The columns represent different variables and the values applied during the test. If the tests were successful the final 2 columns should be Green, the values representing the expected return value of the test.

Test Number	User name	Currency pair	Start date	End date	Projection end	Projection interval	Valid against Schema	Status
4.3.6.2	StevenMason-White@domain.com	null	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.3	StevenMason-White@domain.com	XXXZZZ	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.4	StevenMason-White@domain.com		2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.5	StevenMason-White@domain.com	EURNOK	2050-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.6	StevenMason-White@domain.com	EURNOK	null	2010-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.7	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	null	1Y	1M	N/A	Error
4.3.6.8	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	2050-07-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.9	StevenMason-White@domain.com	EURNOK	2010-07-01T00:00:00Z	2010-06-01T00:00:00Z	1Y	1M	N/A	Error
4.3.6.10	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	ZZ	1M	N/A	Error
4.3.6.11	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	null	1M	N/A	Error
4.3.6.12	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	null	N/A	Error
4.3.6.13	StevenMason-White@domain.com	EURNOK	2010-06-01T00:00:00Z	2010-07-01T00:00:00Z	1Y	ZZ	N/A	Error

Figure 2: Illustrative Examples of functionality within the scope of the Key Example in figure 1

This mix of Key and Illustrative Examples is similar to the idea of the Most Important Tests (Key Examples) and the other contextually relevant tests that are identified (Illustrative Examples), and that we'd like to automate and run but that we might not get time to do so.

Outcome

After working through the Requirements document and conferring in small teams about the Examples of behaviour the system should enact or characteristics it should exhibit a set of Key Examples were agreed upon. Then typical test case design and analysis techniques were applied and a set of Illustrative Examples that demonstrated the scope of functionality were written.

The Example sets were in a format that was accessible to every member of the project team and reflected the Client's language and business context. They could be discussed and understood by the Client and taken directly by the developers and used as task cards to size and plan then develop against.

The test team now knew the Client's acceptance criteria more clearly and had simply to validate each Example was moving to a passing state as development progressed. They also had a set of Illustrative Examples that extended the Key Examples and removed the risk of failing edge and boundary conditions.

Refining the Specification

When the Functional Specification document is written it's written in a way that tries to improve our understanding of the Requirements and tell us what behaviours and characteristics the system should enact to deliver those Requirements.

Typically the Client writes the Requirements, a Technical Architect writes the Functional Spec and the testers take both and write test cases around them. In this way the act of writing each as separate artefacts doesn't directly help refine overall specification of the system.

In practice, overall specification of the system is contained not only in the Functional (or Technical Spec if you prefer) but in the Requirements too. Additionally it's in the test cases that are written, but the problem is one identified earlier in that these are separate artefacts re-interpreting the shared understanding of what the Client wants.

What needs to happen is that the creation of each document helps distil the project team's understanding of the business functionality so that a refined specification can be achieved.

How was it done?

The Collaborative Specification activity and the creation of Key and Illustrative Examples were important first steps in helping to refine the specification. Illustrative examples proved useful in helping to refine the specification by stretching the project team's and Client's understanding of the true scope of requirements and the business problem that was being addressed.

Precondition:
1. User has a valid login account
2. User has performed UI login procedure

Inputs:
1. Username
2. Password
3. <login> command

Outputs:
a. Successfully Authenticated
   1. login confirmation
   2. Session identifier

OR

b. Failed to authenticate
   1. login failure notification

Figure 3: Refined specification as seen in the re-written Functional Spec document

The Technical Architect was able to take a core set of the Key Examples and substantially re-write the Functional Spec in line with them to more clearly and simply state desired functionality.

Each Example was reviewed by the project team and Client to ensure they clarified the specification and didn't leave out any key details or introduce any errors in interpretation. Examples were sanity checked to make sure they were self-explanatory, concise and didn't introduce detail of any implementation specifics or dictate software design.

Outcome

Traditional test cases and scripts were avoided and replaced with artefacts that were precise and testable, geared towards being automated, self-explanatory to those reading them, accessible by the entire project team and in the domain language of the Client.

A refined and simplified specification document was published that mirrored the structure of the Given-When-Then form of Examples. The document drew on the Collaborative Specification activity and used the Requirements and Examples as its basis thereby tying the three artefacts together.

Literal Automation

A risk with traditional automation approaches is that the automation will be written in a way that doesn't directly and tightly adhere to just testing the acceptance criteria that have been agreed. Often an automated test is an (re)interpretation of the test case and includes many steps that are not entirely relevant with regards to what's literally stated in the test case.

This translation issue is more common when using tools that write scripts through record and playback, tools such as QTP, Selenium (IDE) and Visual Studio's Coded UI tests. The tester can all too easily add steps, validations and traversal of functionality that's outside the stated scope of the test case. In addition these tools may automatically use or mandate the use of libraries of test functions that further push the automation away from what's needed for the test case.

In an agile setting this introduces two important issues. Firstly, the test team are delivering work that is out of scope and may be burning time that has not been allocated for the task as they are delivering too much work. Secondly, testers in particular risk raising bugs that are not strictly relevant to the functionality that's been specified.

Though these bugs may be interesting and in some regards useful we continue to enable the dual risks of test scope creep and re-interpretation of the specification – a specification that when following the Specification by Example approach has been distilled, refined and agreed upon. The risk now is non-literal automation could trash the work that's been done.

How was it done?

The expectation was that automation would be used as extensively as possible and so was planned for upfront. This was done by ensuring the test team members who joined the team were skilled in Java development and the tools to be used were trialled in combination early on. This involved ensuring the team were comfortable with the use of Concordion, Eclipse IDE, Perforce and development in Java.

Each Key and Illustrative Example had been written in a clear and specific way so as to allow development and automation against them to be a simple and intuitive as possible. Therefore the guidance to the test team was to avoid re-interpretation of tests cases where possible. The main point here was that with a clear and refined specification in place we don't want to go changing it by virtue of the way we wrote automation.

Automated scripts were written to use a minimum set of steps to achieve the assertion of acceptance criteria for the service layer components under test. This meant automation didn't try to replicate the business logic that was part of the UI layer. For example, service layer tests didn't involve anything related to input validation, checking valid XML, etc. as the service layer would receive inputs and data that was ready to use.

Using Concordion meant the team would firstly *instrument* tests in the Concordion HTML page and then *implement* automation in Java via the Eclipse IDE. Eclipse proved useful as tests could be executed under JUnit before integration into the full automated test suite. Concordion uses a range of tags to connect to methods in the Java class of the test; an example can be seen in figure 4 below.

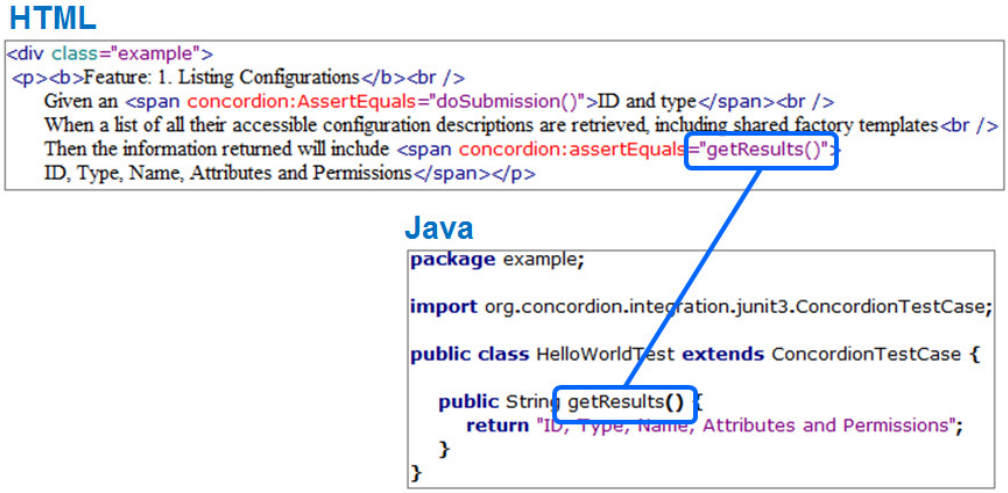


Figure 4: Example instrumented in Concordion and implemented in Java

Test data was reference data where this was available and mirrored current business processes. Where entirely new functionality was being introduced data was designed against the spec. For larger data sets used in the Illustrative Examples the data was drawn from external data files. Test code, along with fixture code, was then submitted to the code repository using Perforce and stored as part of the overall code base.

Outcome

Use of tools and technology that aligned with development team approaches ensured greater collaboration between the teams. With test fixture and Example test code stored alongside the application code and Developer tests it was made available for others for use in Frequent Validation.

Literal automation of each Key and Illustrative Example ensured test code was lean and tightly tied to demonstrating the acceptance criteria had been met and not trying to test everything the functionality might relate to. This avoided over elaboration of the automation, prevented previous work on distilling the Functional specification from losing value and avoided the risk of re-defining the Functional specification and Examples due to test scope creep. The outcome of this was an Executable Specification that could later be run and it's test status clearly and quickly known.

As the automated tests didn't traverse or replicate other functionality they proved to be more maintainable and less fragile when change did occur. Key Example automation scripts provided a template for the automation of Illustrative Examples. Test data within an external data file supplied the data to the Illustrative Examples in most cases.

Frequent Validation

A common mistake in applying testing effort is to leave testing to late in the development cycle. The result is that knowing the level of quality, completeness of development and the test status are all unknowns for a lengthy period.

With a focus on manual testing it's generally possible to run tests sooner than when using automation. However, the use of automation allows greater volumes of tests to be run more frequently and for tests to be run that cannot be performed manually.

On a more agile project, perhaps using a Test First approach, the expectation is that tests will be in place against which developers can then code, in line with the Functional Specification and requirements. The issue here is that the execution of those tests, and the subsequent validation of acceptance criteria being met, is often still done at a later point.

Ideally the team are applying some form of continuous integration that provides the opportunity to validate the completeness of development and testing status. There can be an issue in definition regarding this however as continuous integration practices may be seen as relating more to developers integrating code and running unit tests, than being about validating the Client is progressively getting software that solves their problems.

What's needed is an approach that includes the idea of continuous integration of code but extends to validating the code is delivering what the Client asked for and is progressing towards a complete delivery of expected functionality.

How was it done?

The developers adopted a continuous integration approach right from the start of the project. Daily submissions were the norm with more frequent submissions happening as critical bugs or functionality servicing lots of dependencies was completed. Unit tests were executed by developers on their changes and code submitted.

New builds were run several times a day with developer unit tests running first as build verification and validation tests, managed by Cruise Control.

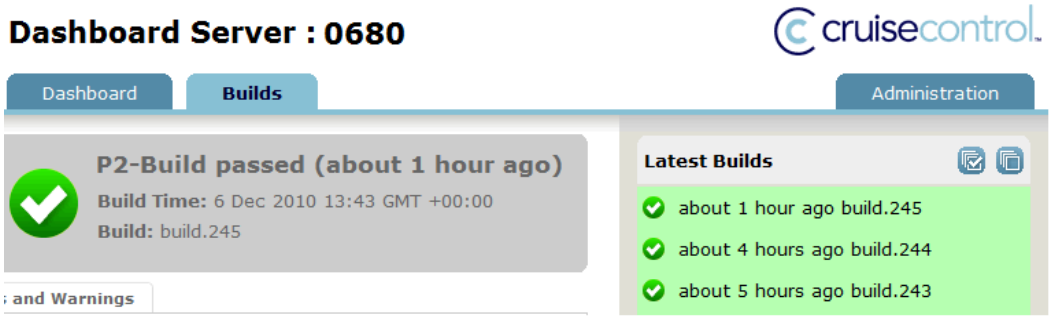


Figure 5: Continuous integration several times a day under Cruise Control

When the unit tests were complete the Key and Illustrative Example automation would then execute. The HTML output produces by Concordion could then be reviewed to check test status and completeness of development. The output would also be stored as a record for future review.

To help show the status of development the Concordion configuration was changed to have a status of Pass, Fail and To Do.

Feature: 1. Listing Configurations
Given an ID and type
When a list of all their accessible configuration descriptions are retrieved, including shared factory templates
Then the information returned will include ID, Type, Name, Attributes and Permissions

Feature: 2. Read a Configuration
Given a configuration ID
When a configuration read is requested
Then a matching configuration is retrieved

Feature: 3. Create Configuration
Given a populated and correct configuration
When a new record is inserted into the store
Then it's possible to return its ID

Figure 6: Concordion output showing the Pass, Fail and To Do test status of the Examples

This additional To Do status could then be used to identify where a test had been set-up but the application development was not complete, thereby avoiding Examples being flagged as a fail when in fact it's an outstanding task.

Outcome

With the approach taken the team weren't just doing continuous integration with related tests running. The practice adopted was much broader and termed Frequent Validation. This included a combination of having developer unit tests run against submitted code, tests executed against Examples and Concordion configured to show test status that included items that were still To Do.

This meant that status regarding the wider development effort was provided at a more regular cadence and it was clearer what development and testing work was left to do.

At the highest level therefore we had used Concordion to help create an Executable Specification that was tied to the Examples it contained, the code developers were delivering and the underlying automated tests created by the test team.

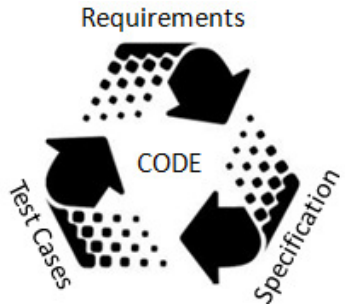


Figure 7: Each element linked so that significant change in one affects the other

The further affect of this was that any significant changes to the Examples, code or automated tests, that invalidated some aspect of another item, would be seen the moment the next Frequent Validation was performed. Refactoring of each item is still allowed so long as that change remained in scope of what was then agreed, anything more was managed as a change request.

Evolving a Documentation System

Many test approaches see the test documentation as artefacts that are only directly relevant to the test team. Though other members of the project team and even the Client to some degree may review or refer to them at some point it can often be that test artefacts are never seen outside of the test team.

This problem is in part due to the way that test cases are written and stored. Their format isn't always that understandable or even relevant in terms of content to other team members. A further issue can be the tool they are stored in may be not be well known or even accessible to the project team and very often not to the Client. Training in their use, access and costly licensing are all common issues with document and test case management tools.

The project team needs test artefacts written in a way that they understand and that they can readily access. However, the documentation produced in the Specification by Example approach can be much more than just Examples of functionality that are relevant primarily to the test team.

The documentation produced in the way proposed in Specification by Example can start to replace that which exists or is usually produced. All documents including Requirements, Use Cases, Functional Specification and Test Cases can to a greater or lesser degree can be replaced with documentation that talks in terms of Examples. The challenge is introducing the change in approach to the project team and ensuring the change is well understood and applied.

How was it done?

Requirements for the current project were already in place when the project team got involved. In addition the first version of the Functional Specification was published in the typical, narrative style.

It wasn't practical to get the Client to re-write the Requirements nor was it the best use of time. The start point for creating our Example based documentation was to use the Requirements and current Functional Spec as the basis for test analysis. As mentioned previously the test team performed test analysis workshops to identify sets of Examples. Each analysis session focused on a single area of functionality and Concordion documentation was arranged in this way.

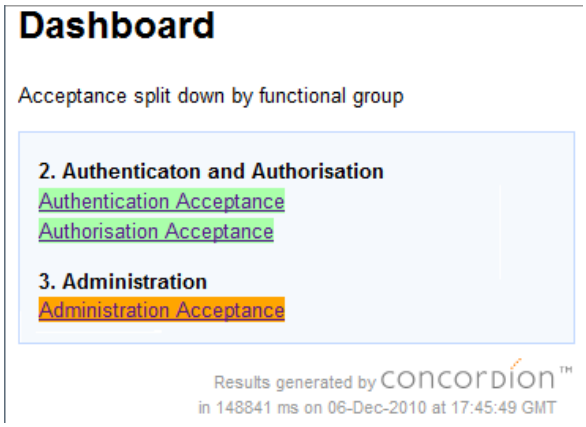


Figure 8: Documentation organised by functional group, using Concordion's breadcrumb feature

This ensured the documentation and test status for each functional group was easily accessible by simply clicking a set of links. The Concordion pages were published onto the BJSS wiki and the URL provided to the Client. The terms used for each functional group matched that which the client used and the language within the Examples mirrored the language of the Client where ever possible. A new version of the Functional Spec was issued that mirrored the structure of the Examples.

Outcome

There were a number of effects on the project team's documentation with regards to adopting the Specification by Example approach. Firstly, the traditional, narrative type test cases were done away with and replaced with Given-When-Then format Examples. These evolved the test cases from something the test team cared about to something the Client, developers and Technical Architect cared about.

All three other teams were able to relate to and use the Examples, whereas with test cases this would not have been possible. The Examples were added to Concordion and so became *Live Documentation* that when updated or added to would show how the overall test and development status had changed, meaning they had become an *Active Specification*.

By working with Examples the test team had an artefact that could be readily automated and used as documentation at later stages. Examples have now become a set of documentation that describes the current system and can be used by Support Teams or as a building block for future iterations of functionality.

Other Observations and Comments

Exploratory Testing

The overall test approach involved demonstrating achievement of acceptance criteria with the Key Examples and testing the scope of stated functionality by using Illustrative Examples. However, these tests were very much focused on the service layer and not on functionality of the UI layer. For example, Illustrative Examples that checked the service layer handling invalid data in no way checked how the UI layer might validate data entry. This type of test condition was handled through testing of the UI by the team developing the UI, which was later integrated with the service layer.

The risk here is there might have been gaps in test coverage and that system levels tests after integration may not have been as robust as preferred. To help address this Exploratory Testing was also used alongside the automated Examples.

Exploratory testing benefitted from 'test questions' that arose in the test analysis session that were primarily intended to produce the Examples. Areas requiring clarification, system component interaction that was not fully understood, questions about expected behaviour of the system on failure were all examples of where many Exploratory Test conditions were identified. When new builds were deployed onto the test environment and the automated tests had run the test team then ran the exploratory test sessions.

Requirements Traceability

To assist with having traceability between Requirements and Examples two steps were taken. Firstly, each Example in Concordion had a note of the Requirement ID included in its description. Secondly, the team produced an Excel based traceability matrix for the Client to review.

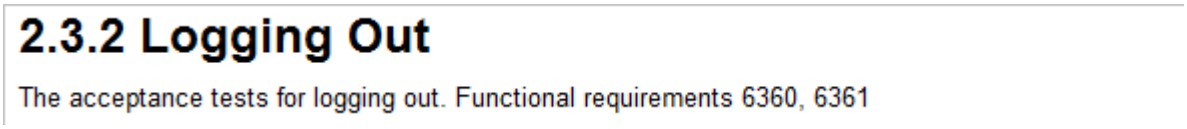


Figure 9: Requirements ID added to the Concordion Examples

References:

Tools and Technologies

Concordion
http://www.concordion.org/ Accessed 30-Nov-2010

Diffusion Message Broker – Push Technology
http://www.pushtechnology.com/diffusion-product/message-broker/ Accessed 01-Dec-2010

Content Server – Fatwire Software
http://www.fatwire.com/products/product/contentserver Accessed 08-Dec-2010

Websphere Application Server (WAS) - IBM
http://www-01.ibm.com/software/webservers/appserv/was/ Accessed 02-Dec-2010

Cruise Control – Continuous Integration tool
http://cruisecontrol.sourceforge.net/ Accessed 08-Dec-2010

Books

Specification by Example. Adzic; 2010
http://specificationbyexample.com/ and http://manning.com/adzic/ Accessed 03-Dec-2010

Lessons Learned in Software Testing. Caner, Bach, Pettichord; 2002
John Wiley & Sons, ISBN-10; 0471081124

Related Articles

http://gojko.net/2010/08/04/lets-change-the-tune/ Accessed 25-Nov-2010
http://www.developsense.com/blog/2010/08/acceptance-tests-lets-change-the-title-too/ Accessed 04-Dec-2010