

## **Quality Assurance Strategy**

Jason Gorman

8 September 2008

## **Summary**

The goals of our quality assurance strategy are:

1. Deliver the highest quality software possible with the available time and resources
2. Achieve quality economically
3. Ensure a lower total cost of ownership

The strategy rests on three widely accepted premises in software development:

1. It is usually exponentially cheaper to deal with defects when they're caught early than it is to fix them later
2. The higher the level of test assurance (coverage, frequency, effectiveness), the easier it is to catch defects early
3. It is usually exponentially cheaper to automate a test than it is to manually execute it many hundreds of times during the life of a software product

Our strategy aims, therefore, to employ continuous automated testing throughout the development process, and to apply quality assurance wherever defects can be introduced in the process to increase our chances of catching them sooner.

We will apply techniques to highlight defects in the requirements specification ("building the wrong thing") as well as in the implementation ("building it wrong"), and will address not only functional requirements but also other areas of quality like software maintainability, scalability and performance, usability and accessibility, security and more.

## **What Will We Be Testing?**

- Functional Requirements
- System Design
- Implementation
- Configuration & Integration

## **How Will We Be Testing It?**

Functionality & UI	UI & System Storyboards Customer Walkthroughs Automated Acceptance Tests Usability Testing & Heuristics
--------------------	--

## Example Agile Quality Assurance Strategy

	Accessibility Testing Exploratory Testing
System Design	Guided Inspections* Simulated Execution* Filmstrips*
Implementation	Unit Tests Non-functional (e.g.): <ul style="list-style-type: none"><li>• Code Quality Analysis</li><li>• Performance Tests</li></ul>
Configuration & Integration	Continuous Integration Smoke Tests End-to-end Tests

*\*Applied on a case by case basis, according to risk and cost of quality*

## ***The Testing Process***

### **Functionality & UI Testing**

In our test-driven approach to development, where tests serve as an unambiguous executable specification of what is required of the software, test analysis and design is essentially requirements analysis and system specification and starts before any new code is written.

When the team is ready to start working on a new user story, the tester, the developer, the UI designer and – most importantly – the product owner, must get together and define the acceptance tests for the story.

These tests will be captured primarily as executable test scripts – probably in some documentation/testing tool like FitNesse or Story Runner. These executable tests are the primary specification from which the team agrees to work. To complete the user story, the software must pass all of the agreed acceptance tests.

To help visualise the specification, and to help validate the user interface design, the UI designer at this point may draw UI storyboards that mirror the flow of the acceptance tests – including any identical test data choices to make the scenario as clear as possible.

At this point it's feasible that test scripts and UI storyboards can be used as the basis for early usability and accessibility testing, by simulating real system usage scenarios before any new code has been written.

It is also possible to validate these designs within the context of a wider end-to-end business process through simulated execution, using user stories, acceptance test scripts and UI mock-ups as placeholders for the finished software. This will help give assurance that we are building the right features.

### **System Design Testing**

After we have validated the functional specification for a user story, we can use each test script as input to high-level internal design. Developers may sketch UML diagrams – class diagrams, sequence diagrams, etc – or use simple design tools like Class-Responsibility-Collaboration cards to capture information about the objects in the system, what their responsibilities are and how they interact to complete the scenarios.

Such models can be tested through a variety of techniques. For example, we can simulate the execution of interactions in a sequence diagram and use snapshots of the objects in the system to visualise and make assertions about object states (a technique known as “filmstrips”). Or we can role-play the sequence of interactions denoted by CRC cards.

Simulated program execution is a more formal and rigorous technique for verifying designs, where execution steps are formally evaluated and their effect on objects calculated according to precisely defined model rules. Such lengths may make sense in areas that are especially quality-critical.

Such techniques can help give assurance that our implementation design is valid before we begin coding.

We can also use such models as a basis to evaluate the impact on non-functional aspects of system quality (e.g., a sequence diagram might reveal that a high-level of “chattiness” between two components through a web service, suggesting a potential performance problem.)

### **Implementation Testing**

Once the internal design is well-enough understood by developers, they can plan their implementation as a sequence of unit tests they will need to pass. Following a strict process of Test-driven Development, the developers will work through their unit test list until they have implemented just enough to pass the acceptance test.

TDD involves continuous refactoring of the code after each test is passed (as necessary) to keep the code as easy to change as possible. In particular, developers will refactor to:

- Remove duplication
- Simplify complex code
- Manage dependencies
- Make the code more readable
- Ensure the software complies with required architectural guidelines and rules

To aid them in this, we will continuously monitor code quality, testing for duplicate code, complex code, high coupling and/or low module cohesion and readability. This will be achieved through a combination of:

- Automated code analysis
- Pair programming
- Regular code reviews, including independent reviews from relevant technical authorities

We can also test the level of assurance our unit tests give us through a practice called mutation testing, where deliberate defects are injected into the code (“mutations”) and the tests are run to see if they detect the defects.

## **Configuration & Integration Testing**

A time-worn excuse of developers when they deploy software that doesn't work is: "well, it worked on my PC". A working software system is more than just working code. Supporting files, databases, user accounts, network connections and other supporting items also make up the finished product.

It is vitally necessary to continuously test the configuration of our software in all of the environments it will need to operate in – including the developers' desktops, testing environments, staging environments and live production environments.

And for a configuration to be testable, it needs to be repeatable. We will achieve continuous testing of our configuration through the practice of continuous integration. Every hour or two, when developers have satisfied themselves through local testing on their desktops that the software works, they will integrate their changes into the shared code repository. This will trigger the code to be built and tested in a dedicated integration environment to provide high levels of assurance that a deployed version will work. Automating building, testing and deployment of software will make configurations easily reproducible and therefore readily testable.

We must also satisfy ourselves that the software system we're delivering will integrate successfully with other systems in the execution of wide-reaching business processes that involve multiple systems.

We will create an end-to-end testing environment for a regular "build of builds", and run end-to-end business-focused tests to assure ourselves that our software will integrate successfully. This "build of builds" will take the outputs from successful point solution builds and deploy them into the end-to-end testing environment.

## ***Quality Assurance Governance***

### **Managing Acceptance Tests**

The acceptance tests for each user story will be identified – but not specified in detail – during iteration planning.

A separate card will be written for each acceptance test. A simple traffic light system (red light – green light) will be used to visually track the progress of each acceptance test.

A red sticker on the card denotes that we have begun work on that test. A green sticker means that the product owner has agreed that the acceptance test has been passed.

Detailed acceptance test scripts will be captured as Wiki pages using FitNesse, in collaboration with the product owner. These will represent the definitive executable specification, and will be managed through source/version control. These tests will be written at an implementation-independent level, and should be readily understood by non-technical stakeholders who have an understanding of the business problem domain.

### **Changing or Removing Acceptance Tests**

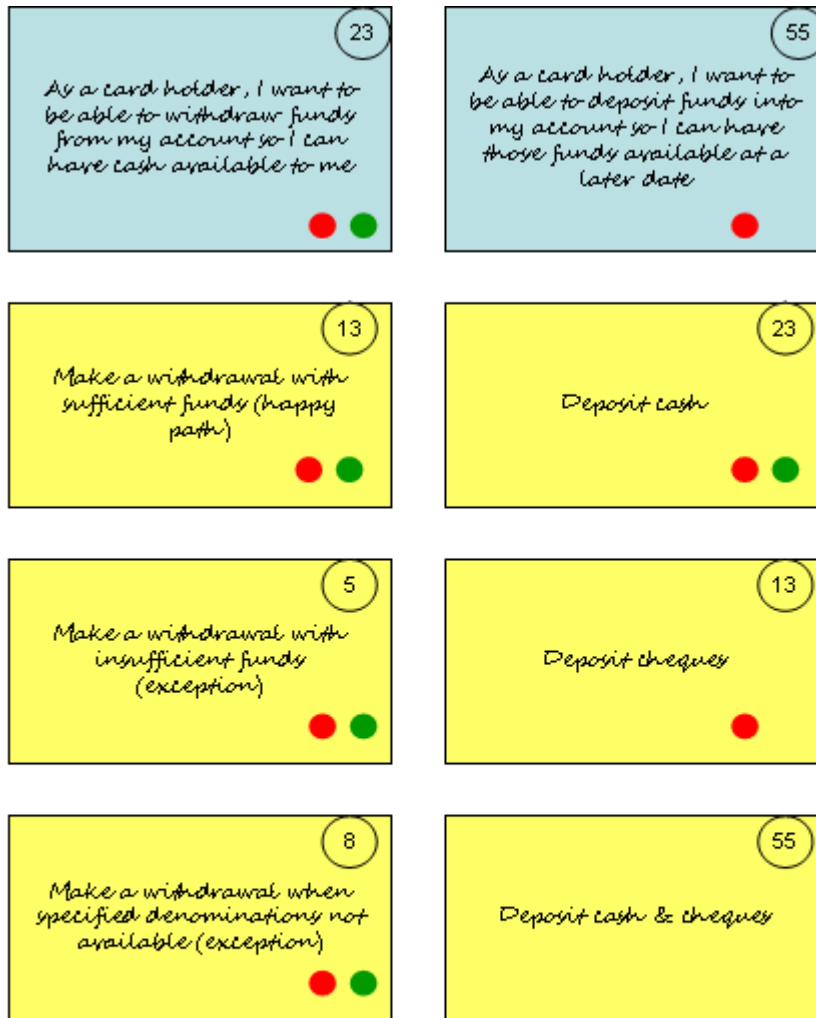
When a change is requested to a user story, this will be reflected in any associated acceptance tests that will need to change, too. If a change means that a test that was showing a green sticker is now no longer passing, then we effectively remove the old passing test and treat it as a new failing test and give it a red sticker accordingly. For the purposes of Agile planning, changes to stories are treated as new stories, also.

### **Monitoring Progress Against Tests**

In a completely test-driven approach to development, features are only delivered if they pass the agreed acceptance tests. Tests passed are therefore the most objective measure of actual progress.

The project burndown chart is therefore a reflection of stories that are testably complete. The iteration burndown can be calculated from tests passed, also (see example below). This is a departure from the traditional approach of planning and measuring progress against tasks, which history teaches us tends to lead to less objectivity and realism in assessing actual progress.

## Example Agile Quality Assurance Strategy



$$\begin{aligned}\text{Story points collected} &= 23 * 100\% + \\ & 55 * (23/(23 + 13 + 55)) \\ & = 23 + 14 = 37\end{aligned}$$

### Managing Defects

Defect reports are captured and managed in much the same way as user stories (but on different coloured cards to make the difference obvious). This information will also be captured in appropriate detail electronically.

Importantly, a test needs to be agreed that makes the defect reproducible and clearly specifies what the product owner believes *should* happen in that scenario.

Defect fixes are scheduled like user stories and progress is tracked against tests passed.

## Example Agile Quality Assurance Strategy

A record of all reported defects and their statuses will be kept so we can report on defect levels, defect injection rates and use the reports/tests to do ongoing root cause analysis so we can take steps to eliminate classes of defects that might have reoccurred.

The golden rule of Agile Defect Management is:

Bugs that get fixed *stay* fixed

Incorporating bug fix acceptance tests into our suite of automated tests will effectively guard against regressions.

### **Measuring Defects**

To help assess how effectively we're ensuring the quality of the software, and communicate this to project stakeholders, we will provide a dashboard with the following defect-related metrics:

- Defect Injection Rate – how many defects were reported during an iteration?
- Defect Closure Rate – how many defects were closed during an iteration?
- Defect Level – how many open defects exist at different levels of criticality?
- Defect Density – how many defects have been raised per thousand lines of code?
- Regressions – How many defects are reported again after being closed?
- Defect Closure Effort – what is the total developer effort expended on closing a defect? (compared to other tasks)

### **Measuring Code Quality**

To help assess the likely maintainability of the code, we will measure and communicate the following metrics:

- Class Size (Lines of Code)
- Method Size (Lines of Code)
- Method Cyclomatic Complexity
- Class Cohesion – Lack of Cohesion of Methods
- Class Coupling – Class Afferent Couplings
- Assembly Cohesion & Coupling – Internal Dependencies / Afferent Couplings
- Assembly Distance from Main Sequence (Abstractness vs. Instability)

## Measuring Test Coverage

To help assess the level of assurance our tests might be giving us, we will monitor test coverage at the system, component and class level:

- Unit test coverage
  - % executable LOC exercised by unit tests
- Integration test coverage
  - % of component interactions exercised by tests
- Automated Acceptance test coverage
  - % of acceptance criteria covered by automated tests

## Unspecified Behaviour

When executable tests serve as specifications of what is required of the software, we accept that if the software passes all of its acceptance tests, it satisfies its specification.

Therefore, when undesirable behaviour (a “bug”) is discovered, and all the tests are passing, it is **unspecified behaviour**.

In these situations, the product owner needs to clearly specify what should happen in that scenario, effectively defining a new user story with its own acceptance test(s).

## ***Retrospectives & Process Improvement***

As well as continuously testing our designs, our code and our configurations, we will continuously evolve the practices and processes we apply to deliver working software.

One mechanism for process improvement will be regular retrospectives. A retrospective is a meeting that usually takes place at the end of each iteration. Team members and other key process stakeholders – including the product owner – are encouraged to openly and honestly articulate their views about what they think went well (and should be replicated in subsequent iterations), and what they think could be improved upon.

The “prime directive” of retrospectives reads:

*“Regardless of what we discover, we understand and truly believe that everybody did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand.”*

The purpose of retrospectives is to plan ways in which our approach to delivery can be improved without apportioning blame or “just having a general moan”. It’s intended to be a constructive activity, and the actions that fall out of retrospectives should be positive.

A good thing to monitor from one iteration to then next is how many areas that are identified as needing improvement continue to be a problem or blocker to the project.

With relatively long iterations, we also propose to have weekly reviews – mini retrospectives – to address pressing issues and reinforce good habits more immediately.

Daily stand-ups are another regular opportunity to raise process issues and seek to address them sooner.

## **Evidence-based Process Improvement**

One criticism often levelled at techniques like retrospectives is that they are highly subjective, and it is true that teams may sometimes choose to ignore major problems (the so-called “elephant in the room”) and focus on the things they feel comfortable talking about.

We can add some objectivity to this by complementing anecdotal accounts of team performance and software quality with metrics and other kinds of evidence. Sometimes these metrics will reinforce the anecdotal picture; sometimes they will contradict it – in which case, further investigation might be needed to establish the reality. Oftentimes they will highlight things that the team had failed to notice “in the heat of battle”.

Again, there are critics of metrics who argue strongly against relying on them completely. Which is why we favour a balanced approach of personal accounts and discussion, illuminated with a useful metrics to provide a fuller picture.

## **Further Reading**

Agile Iteration Planning

<http://martinfowler.com/articles/planningXpliteration.html>

Test-driven Development

[http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

Refactoring

<http://www.refactoring.com/>

Unit Testing Frameworks

<http://www.junit.org/>      <http://www.nunit.org/>

Agile Testing

<http://skillsmatter.com/podcast/home/understanding-qatesting-on-agile-projects>

FitNesse

<http://fitnesse.org/>

Selenium

<http://selenium.openqa.org/>

Catalysis

<http://www.trireme.u-net.com/catalysis/>

OO Design Principles & Software Metrics

<http://www.parlezuml.com/metrics/index.htm>

NDepend

<http://www.ndepend.com/>

Retrospectives

<http://www.retrospectives.com/>