



PACIFIC NW
28TH ANNUAL
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010



ACHIEVING
QUALITY
IN A COMPLEX
ENVIRONMENT

*Conference Paper Excerpt
from the*
CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Effective Testing Techniques for Untold Stories in Story-Driven Development

Erbil Yilmaz, Gokhan Ozer

erbily@microsoft.com, gokhano@microsoft.com

Abstract

Feature crews (teams) take a few stories and implement them as part of story-driven development in each sprint. The goal, at the end of each sprint, is to demonstrate and complete the customer experience along the stories designed. This is a great software development methodology proving the software being built can deliver the customer experience.

However, story-driven development brings unique challenges for software testing. As stories are implemented, relying on each other, a complex software system emerges capable of doing more than just the stories told. With the addition of each story, testing surface and capabilities (and defects) of the software grow exponentially.

This paper describes various techniques developed and used over several releases through experiments in testing within agile development by the Visualization & Modeling Tools team for Visual Studio Ultimate. It also presents a case study to illustrate and emphasize key points, ranging from using architecture diagrams as part of test planning to particular test coverage along with test spectrum range. The guidance presented here focuses around a set of best practices that teams can adopt to accomplish optimal test coverage over the test spectrum.

Biographies

Erbil Yilmaz has over 7 years of software testing experience with Microsoft as the owner of security and performance exit criteria for the Visualization & Modeling Tools team for Visual Studio Ultimate. He is part of the Test Architecture Group and drives integration testing with agile development, effective test planning, and performance testing strategies. He holds an M.S. in Computer Science specializing in Computer Security from Florida State University. He also has B.S. degrees in Computer Engineering and Mathematics from Bogazici University.

Gokhan Ozer has also been working on Visualization & Modeling Tools team for Visual Studio Ultimate for 2.5 years, as the owner of stress/memory exit criteria. He holds an M.S. in Computer Science from University of Houston and B.S Computer Engineering from Istanbul Technical University.

1. Introduction

Agile methodology focuses on producing working software that can be demonstrated (close to shipping) to customers at the end of each sprint. The software product is in increments of enabled stories or features over multiple sprints; inherently all software development activities required to ship the current product are completed within the sprint boundaries.

However, this brings several unique challenges to software testing:

- (1) Usually, different parts of the software are developed by different sub-teams, making it very difficult to manage and understand a complete set of capabilities of the software. As a result, at the end of each sprint, the software product consists of:
 - a. Features that are developed for the told stories
 - b. Features/defects that emerge from untold stories, which result from the interactions of the implemented stories. Sometimes these features are as designed, but often hide significant integration bugs.
- (2) Focus on testing the story within the iteration accumulates test debt for integration points that is hard to estimate and requires long stabilization/verification cycles after all stories are implemented (code-complete).
- (3) As iterations progress, more test defects will be discovered around the integration points (untold stories) rather than coming from existing test coverage. This causes questions around the existing test coverage and test automation, if it exists.

These challenges are not solved yet, partially because software testing has not matured enough within agile development methodologies. This paper introduces a set of techniques and best practices that were developed by the Visualization & Modeling Tools test team over the last two releases to alleviate these challenges.

2. Testing for Stories in Agile Methodology

In Agile methodology, stories define what the customer will be able to do after the implementation of the feature. Naturally, test planning and testing for a given story focuses on variations and contexts around the story, enumerating different use cases within the scope of the story in test plans.

There is also significant focus on unit testing and acceptance testing as part of iterations. While unit tests ensure that implemented methods achieve what they are designed for, acceptance tests target the end-user experience at the system level. In both cases, there is a test design tendency towards isolating the software implementing the story from its dependencies or interactions with other features with various techniques such as mocking interactions.

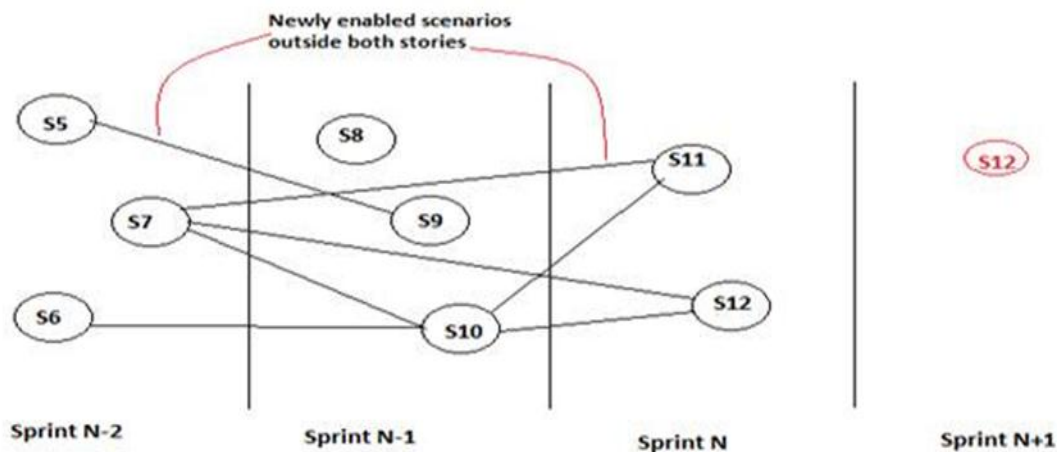
As a result, for each sprint, stories become high level encapsulations of software increments fulfilling stories (features), and test cases mostly verifying stories in isolation.

3. Testing Challenges arising from Story-Driven Development with Agile Methodology

Story-driven development over iterations brings several challenges to testing. As iterations progress and new stories are completed, new test surface emerges from interactions of stories that inherently are not well covered with testing focused mostly on stories within iterations.

The following figure explains how interactions between stories create a complex map of stories over iterations. Each link between stories represents a newly enabled scenario that is not accounted for in either story. This section will describe a series of testing challenges with references to this figure. As a result, for each sprint, stories become high level encapsulations of software increments that fulfill stories (features), and test cases that mostly verify stories in isolation.

For example, in the following diagram, S11 is an extended story on S10, but, through implementation, it also enabled new scenarios interacting with S7 that were not covered with S7 or current S11 test planning.



3.1 Complex Story Map over Iterations: Untold Stories

Discovering a complete map of interactions between stories is a very hard task, which is proven by the significant number of defects that are found after iterations and that are not covered by designed test cases. However, carefully examining these interactions to uncover defects could be overwhelming. In some cases, interaction between stories might be easy to discover if the current story is a natural extension of previous story in consecutive sprints. However, if the stories are not in consecutive sprints, or if multiple stories are involved in interactions such as through a cross-cutting feature, it is extremely hard to plan tests around these interactions. This results in incomplete test plans with potential high risk defects.

3.2 Integration of Features Developed by Different Teams

If there are multiple teams involved in developing a set of stories that contribute to a feature, the chances of discovering all interactions decrease significantly, and assumptions and code interactions are not surfaced.

3.3 Accumulated Test Debt

Since most of the interactions among features are not covered within sprints delivering stories, teams usually accumulate significant test debt over the release cycle. As untold stories are discovered, new test work items are added to the product backlog. Even though defining newly found test work helps to ship the product with fewer integration defects, it doesn't solve all the potential problems. Additionally, the product owner is likely to continue to prioritize new features over older test needs (or debt).

Depending on the time of discovery in the product cycle, teams react to these test holes in two ways. If the integration test hole is uncovered during coding sprints, then agile teams usually expand test plans to cover these areas and implement regression test cases for all significant bugs. This is accounted for as newly found work and treated like the rest of the backlog items. Even though this does not disturb the rhythm of sprints, it still extends the product ship cycle and questions the readiness of the implemented feature. In some cases, it requires architectural changes with effects on the rest of the features or cross-cutting features.

In general, fixing defects that are discovered later in the product cycle costs more. If these inherit integration defects are discovered late in the product cycle, it becomes even harder to react. In some cases, untold stories are discovered after the release by customers causing expensive patches or service packs and more importantly the loss of reputation and lost future sales.

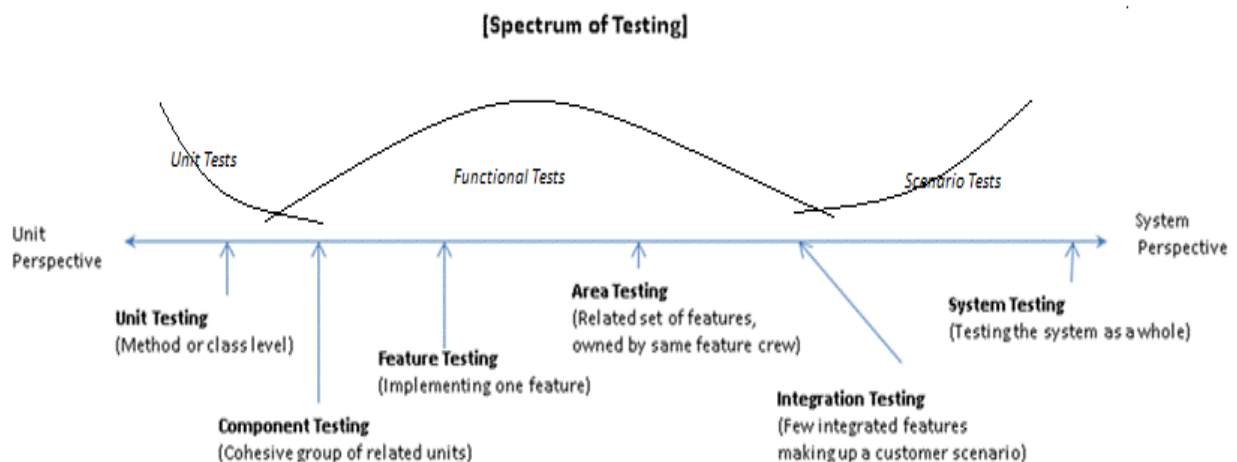
3.4 Degrading Trust for Existing Test Coverage

One of the most common problems with the lack of coverage for untold stories is that after code-complete (i.e. all stories with the implementation of features and tests for each story are complete), there is still a significant number of defects in the defect database mostly due to integration points between completed stories. This significantly degrades trust for existing test coverage as none of these defects fail existing test cases.

4. Techniques for Capturing Links among Stories

4.1 Using Test Spectrum

Test spectrum is one of the tools that the Visualization & Modeling Tools team devised and has been using to help improve test coverage. This spectrum provides a broad view on what type of testing needs to be done to verify the product. It also reminds the team that in order to have optimal coverage, they need to have a distribution of test cases across the spectrum, including touch-points among features implemented with area, integration and system level testing.



The following list describes the tests used within this spectrum,

Unit Tests: Tests that confirm a single unit of execution (e.g. a public method) behaves as expected.

Component Tests: Targets a few related units together to see if they work as expected. The majority of tests that are implemented as part of development (TDD) will fall into this category.

Feature Tests: Targets whether a product feature works as expected. I would expect most of these tests will be implemented as part of development (at least positive paths) and some edge cases, while negative paths might be delayed a few days. Feature tests can be implemented as either functional tests or as scenario tests (definitions below).

Area Tests: Targets how a series of features intersect with each other within the scope of feature crew ownership. I expect major paths are implemented as part of development and that all these tests should be done within sprint.

Integration Tests: Targets features across feature crew boundaries, making up a customer scenario. I expect new paths that are added by the feature crew should be tested within the sprint as well as all paths changed due to features implemented by feature crew.

System Tests: Targets the system as a whole, potentially new or changed paths need to be included in test planning within the sprint, and tests should be available as integrations happen. End-to-end scenarios are good examples of system level testing. Stress testing, performance testing, etc. can be considered as a cross-cutting set of test categories presented with the spectrum. From a coverage perspective, such individual test cases would fall into one of the following test categories.

4.1.1 Test Types within Spectrum

Across-the-spectrum tests can be categorized into three broader groups:

Unit Tests: Tests that confirm that a single unit of execution (e.g. a public method) behaves as expected.

Functional Tests: Functional tests should confirm that programmatic classes, objects, and methods work together to do what they're supposed to do, where possible. Functional tests are written against the public API of the product.

Scenario Tests: Tests that confirm a specific user experience. Scenario tests simulate the experience of a real end-user. Scenario tests are written against an abstraction that mimics user action. It should ideally read like a spec. It can also be written to not use UI (using the API) for better performance and robustness.

4.1.2 Test Planning Guidelines with Test Spectrum

Test spectrum is designed to remind the team that for any given feature, multiple levels of coverage are required focusing on different aspects of the implemented code. As test planning and low-level test implementation starts, keep in mind that optimal coverage could be achieved only by a combination of the various elements in the test spectrum. For the story being implemented, choose a good set that is a combination of these types – do not rely only on unit and feature tests.

As part of a test plan review checklist, it also helps to question how test coverage will be achieved at different levels. This suggests that the test plan should call out integration points with other components/features/areas that are implemented or planned. For optimal coverage, either add tests for these to the current test plan or add a task to the backlog to identify or implement these later. As other team members review tests plan/implementations, they should provide feedback about missing integration pieces in the coverage.

4.2 Test Meta-Data

After test cases are designed to consider the full test spectrum, multiple levels of test cases will be created. Another dimension of meta-data is required to relate test cases to stories and provide traceability. At a minimum, each test case should have meta-data that describes: (i) components to be tested with their scenarios, (ii) test types according to the test spectrum (iii) stories that the tests are covering.

This meta-data will provide a set of attributes that can be queried to find relevant sets of test cases for each story implementation. Visual Studio 2010 provides extensible test categories and test properties to attribute test cases with proper meta-data. The Visualization & Modeling Tools test team created a set of extensions to test categories and used them to implement the Visual Studio 2010 Visualization and Modeling Feature Pack.

4.3 Using Architecture Diagrams to Relate Stories to Components

Another technique that the Visualization & Modeling Tools test team used in test planning was to include architecture diagrams as a requirement. Each test plan had one or more UML architecture diagrams that defined the context for the test plan. Using these diagrams in test plans helped relate test areas to product architecture and also question the test plan in terms of links between components/layers in the product architecture.

If there is a link between the components being tested and other components, it means there might be a code execution path that enables a story (possibly an untold one) which goes through these interacting components. This will translate as requirement to design a set of test cases in the test plan to cover the interaction.

The Visualization & Modeling Tools test team commonly used layer diagrams, UML component diagrams, and UML class diagrams to set context for the test plan and to discover potential untold stories. Visual Studio 2010 Ultimate includes these modeling diagrams.

4.4 Story Links over Architecture Diagrams

As the next step in the evolution of test techniques to cover untold stories, the Visualization & Modeling Tools test team created a set of component diagrams to improve the links between stories and the product architecture. In these diagrams, each story that has been implemented thus far overlays architectural components that implement or enable that story.

As more stories are implemented, these component diagrams encapsulate all stories overlaid on the component diagrams that represent the current architecture. If there is a link between two architectural components, it implies a link between the stories that overlay those architectural components. Each link at either level deserves an investigation on how to design test cases to cover the interaction. If the link is at component level, the corresponding test could range from component test to integration test. If the link is at the story level, then the corresponding test could range from feature testing to system testing.

5. Planning for Untold Stories

5.1 Case Study: Closing Integration Test Gap

In some cases, a team might discover that existing test coverage is sub-optimal and needs to be improved. The following work was completed in Sprint 5 by one of the Visualization & Modeling Tools teams to improve test coverage for integration points between already implemented stories.

As part of an integration test work item in Sprint 5, QA members worked on a two-step process. First, they investigated and identified the low-level testing gaps for all areas. They used area tests only for code-coverage runs and, without integration/system level tests, figured out how much coverage gap they had with existing tests. All areas should have good low-level test coverage by only area test.

Apart from low-level testing coverage, they went over all the previously implemented stories and looked for possible integration links or scenarios across the areas that they didn't account for as part of implementing the story. Given that there were twenty-five stories that were implemented in five sprints with a significant number of interactions among them, it was a big challenge from both a story perspective and a quality perspective to account for all the enabled (intentional or unintentional) scenarios as stories come online. For example, when we implemented the Cancel story, the Update story did not exist. Cancel story enables the user to cancel a layout operation, whereas Update story enables changing various properties of nodes in the diagram. After we implemented the Update story, the functional surface that the Cancel story exposed had increased significantly. Therefore, as these two stories came online at different times, there were possibly interactions between these two stories that did not get highlighted in either of the test plans.

QA team members had brain-storming meetings to enumerate integration scenarios across components or areas as well as stories. They updated the test plans as they found these integration scenarios. It was relatively easy at this stage as the team had already started using architecture diagrams in their test plans. Going forward, all test plans will have sections for integration scenarios with enabled or previous stories as well as a section for possible feature story integration scenarios.

The team is now more successful at questioning the integration points between stories that come online and is more immune to impactful bugs that fall into the cracks between stories or components. This activity is not limited to QA; the entire team should be looking for these cracks in design discussions and story/spec reviews as they build the product daily.

6. Conclusion

Designing tests to cover integration points is a very difficult task. It gets more difficult in agile development as there are logical separations between stories or features. Designed tests usually focus on acceptance criteria and low-level coverage for the components that enable the story within sprint boundaries. It is also difficult because there is a "done" expectation at the end of each sprint, specifically, that the product should be ready to ship. It frequently adds pressure to complete the feature with very limited time left for testing.

The following steps are advised to ease issues with integration points during test design:

1. Ensure that you have a good set of test cases across the test spectrum for a given story. Do not focus only on unit tests or acceptance tests, but also carefully consider integration points for already implemented stories.
2. Mark your test cases with meta-data associated with test coverage intention, test type, and the story or feature that is related to the test. This will provide traceability and a test bed that you can query.
3. Use internal product knowledge, especially architecture diagrams, to map stories to actual code. Further, design test cases over this mapping with both links to stories and product code. This will help you to discover links among stories that would be very hard to find otherwise.
4. If you fail to identify integration points within the iteration, add high priority test coverage work items to the backlog as you discover them. Do not accumulate debt as it carries significant risk closer to the release date.

If integration points are considered carefully during agile development from testing perspective with the techniques presented in this paper, then end game stress can be more predictable and manageable.

References

Beck, K. et. al., 2001. Manifesto for Agile Software Development, <http://agilemanifesto.org/>