Turning a Marathon Runner into a Sprinter:

Adopting Agile Testing Strategies and Practices at Microsoft

Jean Hartmann

Test Architect
Office Client Services
Application and Services Group
Microsoft Corp.
Redmond, WA 98052
Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

For many years now, Office has successfully released its Productivity Suite including Word, Excel, PowerPoint and Outlook with a regular cadence of about three years. These major releases have relied on a substantial amount of testing to ensure high-quality and compliant products are shipped worldwide. However, with future releases, the client division is moving to a more agile model for software development, which has major ramifications for software quality and testing. The move is compounded by the need to also deliver these products on a wider variety of devices and platforms. This paper will highlight improvements such as greater emphasis on developer unit testing for early bug detection, smarter regression test selection to enable faster, higher quality deployments and streamlining of mandatory ship criteria and improved product telemetry for evaluating and assessing test automation quality.

Biography

Jean Hartmann is a Principal Test Architect in Microsoft's Office Division with previous experience as Test Architect for Internet Explorer and Developer Division. His main responsibility includes driving the concept of software quality throughout the product development lifecycle. He spent twelve years at Siemens Corporate Research as Manager for Software Quality, having earned a Ph.D. in Computer Science in 1993, while researching the topic of selective regression test strategies.

1. Introduction

A waterfall-based, endurance-like approach to software development has worked well over the years for teams developing Office desktop products like Word, Excel, PowerPoint and Outlook. Going forward, however, teams did not want their customers to have to wait for several years before receiving the latest set of features and enhancements. Instead, our teams wanted to deliver a steady stream of new features at shorter time intervals or sprints as well as across a broad range of platforms and devices. It was time for these marathon runners to transform themselves into sprinters!

To achieve this goal and successfully make the transition to an agile world, teams realized that they needed to strike a careful balance [1]. They needed to embrace the best aspects of agile (and cross-platform development) and then blend those with best practices that already existed within Office.

Planning teams were assembled and tasked to explore key themes. These would need to be promoted, developed and supported by the organization as it geared up for a fresh round of development. In some cases, the topics and investments were obvious with ROI (Return On Investment) data available from within and outside the company to demonstrate their benefits. In other cases, the decisions to invest in a theme were based more on engineering intuition and experience, especially when empirical data was not readily available. A good example of the latter was the broad adoption of unit testing practices. Much anecdotal evidence existed within and outside the company regarding unit testing, but little ROI data was available regarding its cost vs. benefit.

Key test- or quality-related themes included:

- *Moving quality upstream.* We anticipated that by supporting a better developer (unit) testing experience during refactoring and componentizing of existing products, bugs would be caught earlier, reducing defect leakage and maintaining quality. Such work would also be a cornerstone of efforts to improve code velocity and deployment in the context of continuous integration.

- *Better leveraging product telemetry.* Collecting and analyzing product telemetry data for key attributes, such as bugs, performance, reliability, security and network latencies, etc. represents a powerful mechanism to continuously improve on product quality and can also be leveraged to better assess, guide and influence process improvements, such as the quality and scope of teams' test automation.

- *Adapting cross-platform testing for agile development.* With Office products sharing a number of common components across different platforms and devices, it was important to improve agility in the context of cross-platform development. Rationalizing resources around validating those components via a single platform-agnostic test suite per product rather than multiple, platform-specific test suite became important.

- *Streamlining mandatory shipping criteria - Quality Essentials (QE) - including security, accessibility and localization testing in the face of faster release cadences*. In previous ship cycles, such non-functional testing required substantial resources and effort to satisfy. Through clear definitions of

deployment quality gates (so-called “rings of validation”) and better integration of associated work items into the agile process, the additional effort incurred was intended to be kept to a minimum.

- *Ensuring rock-solid tools support and integration*. It was clear that shifts in product development philosophy and cadence could not be accomplished without good tools support and proper integration into the new agile development process. As a result, the central engineering and tools team needed to quickly evaluate their existing tools and delivery mechanisms, to gather requirements and close outstanding gaps.

The remainder of this paper summarizes our ongoing efforts with respect to each of these themes.

2. Emphasizing Developer Testing

Many years of development effort have resulted in large, monolithic code bases for classic Office products. So the need to refactor and componentize these products was steadily growing. The broad adoption of developer (unit or component) testing was seen as one way of ensuring product quality was maintained during refactoring and componentization. It also provided the benefit of early bug detection, reducing defect leakage. Developers were able to implement and validate their classes or components quickly and frequently and as early as possible.

Apart from driving quality improvements upstream, developers needed confidence with integrating and deploying their code into production faster. Continuous integration became a key issue. This required more extensive testing, against a wider set of integration and system criteria than just unit or component tests. At the same time, check-in times needed to stay reasonable. Smarter regression test selection criteria were developed to address this issue and are currently being evaluated.

2.1 Introducing Unit/Component Testing

While most Office developers have not (yet) embraced true, agile testing practices such as TDD [2] – they still write the code first, then create the tests to validate it – they are making strides in developing automated unit test suites. Developers have rapidly accumulated hundreds of thousands of unit tests to validate core functionality and are continuously improving their levels of coverage.

Rather than rigidly adhering to the classic definition of unit testing that prescribes validating every class and its methods, developers have the flexibility to validate larger chunks of code, namely components or sub-systems. The decision is largely based on the developers’ assessment of the code, its complexity and dependencies.

When writing unit/component tests, developers typically produced C++ code, ensuring that it was portable across multiple platforms (compilable and executable using platform-specific compilers). Unlike system or scenario-based tests, these test suites were executed as part of the same execution process as the product (or “in-proc”), making their execution very fast and efficient; and more importantly, enabling quick and easy debugging of the product.

In some teams, developers and testers shared the development of component tests. The anticipated benefits for testers included getting a deeper (white-box) understanding of the product code, its algorithms, data structures and event models. It also gave testers the opportunity to improve product testability by adding appropriate tests hook that made it easier to verify the products. Apart from developers' workloads being reduced, this collaboration also represented a source of independent verification and validation for their designs and implementations. From an agility point of view, having testers more informed about product code intent and purpose resulted in deeper, more meaningful testing.

Our initial belief that unit testing would be costly in terms of mocking and stubbing due to the large number of potentially complex dependencies of legacy code was groundless, although in many cases this may have been due to developers sidestepping the issue by validating at the component level instead.

As this unit testing initiative is still in progress, its benefits and ROI (additional bugs found vs. effort invested) are still being evaluated. It is too early to draw any definitive conclusions about the broader, long-termer benefits. Having said that, more immediate benefits, such as leveraging unit tests to help developers correctly resolve merge conflicts, have been reported. Also, the stability of some products *appears* to be improving with system-level regression test runs passing at higher rates than in the past. However, at the time, we have not been able to correlate unit testing efforts with the latter observation.

2.2 Smarter Regression Test Selection

Smarter regression test selection techniques and criteria enable developers to validate code changes quicker and with more confidence, enabling faster deployment to production. The aim is to give them ways to more effectively and efficiently validate code by selecting an appropriate subset of tests for rerun, instead of having to execute all tests [3]. The techniques enable developers to perform *pre-checkin validation* themselves rather than relying upon *post-checkin validation* efforts conducted by testers. In the former case, regression testing acts as a powerful gatekeeper mechanism to raise developers' confidence in selecting and running the most appropriate test automation from amongst a team's unit, component and system tests as well as informing them about code changes that coincide with sections of code, not previously tested. Selecting from a range of different types of test cases ensures that the code is being exercised in the broadest possible context. Continuous integration and deployment now becomes a more reliable, predictable and measurable process.

At this time, smarter regression test selection for developer check-ins has just been introduced. Its benefits and ROI (additional bugs found vs. tests selected for rerun) are still being evaluated. Based on results achieved by these techniques during post-checkin validation, they will provide significant, additional value.

2.3 Improving Product Testability

As part of the drive towards refactoring and componentization of the various products, developers and testers also had an opportunity to collaborate to improve the testability of their product code. Better testability focuses on improving controllability and observability of a product and is typically achieved by exposing more of the internal behavior and properties of the product via a set of application interfaces. As a result, developers and testers can, for example, place their product into specific states or add a specific

data value to a document property (set). Conversely, they can easily observe state or document property changes (get).

Improving product testability helped developers, making their unit and component testing efficient and effective. In addition, testers benefitted as they could now significantly reduce their reliance on user interface (UI) testing and instead, access the equivalent application logic[1]. The resulting component and system tests executed faster, proved to be more reliable and exposed deeper, more fundamental flaws in the application logic. After seeing the early benefits of testability, developers have now become accustomed to exposing internal product details.

3. Adapting Cross-Platform Testing

In a previous PNSQC paper, the topic of cross-platform testing at Microsoft was described in detail [4]. The use of this approach was thought to be particularly compelling, when a significant portion of core product code could be shared across multiple platforms as in the case of many Office applications. It was also anticipated that it could contribute to our agile development practices as a single platform-agnostic test suite would replace multiple, platform-specific ones. A good example were the cross-platform File I/O or file fuzzing activities where Office applications need to be validated against multiple platforms and devices using a common, shared scenario test. Previously, multiple tests were written; now the same goal can be achieved by one, shared test!

Several Office teams recognized the potential for leveraging such cross-platform test suites and are starting to adapt the approach described in the paper. They are currently authoring new shared, desktop-based C# tests, which in turn validate their applications on different platforms and devices. They are, in effect, future-proofing their test investments and improving their agility in terms of being able to quickly validate shared, common code on new platforms and devices, as these arrive in the marketplace.

Having a single test entry point (PC desktop) also enables teams to leverage existing, mature and very useful tools for the same application on different platforms, making the ROI even more compelling. A good example of this are in-house fuzzing tools, used on the PC for years, but not available natively for other platforms. Now, the generated tests can also drive the same application on those non-PC devices, improving agility and quality!

4. Leveraging Telemetry

Product telemetry enables developers and testers to add additional code to their products in order to track various quality attributes and ensure minimal deployment downtime for our applications. Telemetry typically focuses on collecting data to ensure a better customer experience. In the past, there was little telemetry

[1] The limitations of UI testing are well-documented, including lack of test stability and performance due to frequent changes in UI layout during development and potential masking of bugs by the UI, which were generated within the application logic, but were never ‘visible’ in the UI.

gathered to give better visibility into customer issues. It also took a considerable amount of time for the organization to react, namely provide bug fixes, in response to issues. One of the few forms of product telemetry available to Office was delivered courtesy of the Windows operating system. The Windows Error Reporting (or Dr. Watson) service reported various operating system and application errors and crashes [5].

With the advent of online services such as Exchange, SharePoint and now Office365, this has changed. By extensively instrumenting these services with telemetry points and monitoring their functional and non-functional behavior, Office online services teams are able to resolve problems and deploy fixes much faster than their Office counterparts on client platforms and devices. This is, however, changing as those client teams are also intending to leverage telemetry as part of their improvement efforts. They are essentially using the same backend infrastructure to collect telemetry data as their services counterparts, but developing a unified logging mechanism that can be used across various platforms and devices, browser-based or native applications. The result is the development of a unified telemetry infrastructure where products spanning client platforms and devices as well as servers can be analyzed.

Product telemetry is an important consideration when moving to an agile development philosophy as products and services are being continuously integrated, built and deployed. While developers and testers do a good job validating main product features and workflows in any given sprint, they also need more visibility into the product to minimize downtime when something does go wrong. Beyond debugging, the question arises – can this product telemetry data also be used to assess, guide and influence the need for test automation?

5. Integrating Release Criteria

Ship or release criteria represent an extensive list of non-functional quality criteria that need to be satisfied by every team within Microsoft before deploying a product or service to external customers. Examples of such criterion include accessibility, security, privacy, world-readiness and localization [6].

In the past, teams spent considerable development and testing resources on satisfying these criteria. They would often spend dedicated time after completing their functional features to validate against these non-functional criteria, reworking designs to satisfy them. In each product release, a significant portion of the total number of work items and bugs would be attributed to such work.

With the shift to agile development methodologies, test teams no longer have the luxury of validating and satisfying these criteria after completing their features. The challenge here is to spread the required testing effort across the new agile development cycle without lengthening it. The hope is that better techniques, tooling and procedures can help. For example, localization requirements can be pushed upstream, so that developers consider them as part of their early design discussions and planning. Tools could be deployed to perform certain checks at code check-in time rather than running them later on entire products. In the past, testers often verified these criteria during the later phases of development, for example, by performing accessibility testing. The intent of this initiative is to revisit the existing set of criteria and more effectively integrate specific aspects into the agile development workflow. The benefit is that teams satisfy the criteria

as an integral part of their daily workload rather than as a distinct and separate workload after the product functionality is already completed and is ready to be deployed.

6. Streamlining of Engineering Tools and Services

An important and yet often overlooked consideration when moving to an agile development methodology is how the engineering tools and services can support this transition. Key technological improvements, such as continuous build, virtualized test and debugging improvements, need to be adopted and integrated into the new agile development processes. Improvements to test automation were particularly important to developers and testers who needed to frequently run their respective automated test suites and debug products. They needed fast and reliable tests and test results that detected product bugs rather than highlighting test failures. In the past, they were often hampered by slow test execution and unreliable test results. In order to remedy this situation, the engineering team initiated efforts to better measure and improve the stability of the test collateral to achieve higher pass rates. To complement that, test execution leveraged virtual rather than physical machines, which significantly reduced set up times and helped both debugging and test throughput. Whenever tests failed and product needed to be analyzed, the virtual machines (VMs) containing the buggy product could be saved elsewhere and debugged at leisure with the hardware being reclaimed for further test runs. Average test job turnaround times have been reduced by up to 50%. As part of these improvements, the engineering team also promised more visibility into their services including timeliness and reliability by developing better “dashboards”.

7. Conclusion

In this paper, I have highlighted key areas of improvement to our development and testing processes as Office teams move away from a waterfall-based, endurance-like model to embrace a more agile, sprint-like model. While I discuss important aspects of each area, work is ongoing and the verdict is still not out as to their ROI. Having said that, teams are starting to reap some of the early benefits.

8. Acknowledgments

I would like to thank Marty Riley, Principal Test Manager, Curtis Anderson, Director of Test and Tara Roth, Corporate VP of Office Client Services for their ongoing support. I also want to express my gratitude to all Office team members for their contributions, support and discussions concerning this work. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

9. References

1. K. Sureshchandra, J. Shrinivasavadhani, "*Moving from Waterfall to Agile*", IEEE Agile '08 Conference, pp. 97-101, Aug. 2008.
2. K. Beck, "Test Driven Development: By Example", Addison-Wesley Longman, 2002.
3. J. Hartmann, "Thirty Years of Regression Testing", PNSQC 2012.
4. J. Hartmann, "Exploring Cross-Platform Strategies at Microsoft", PNSQC 2011.
5. K. Foster, "Windows Error Reporting: Elementary, My Dear Watson", www.tabpi.org/2006/rch3.pdf
6. Introduction to the Microsoft Security Development Lifecycle (SDLC),link