Agility and the Inconceivably Large

Ade Miller and Eric Carter

Abstract

This report outlines the experiences of the Microsoft Visual Studio Tools for Office product unit (hereafter referred to as "product unit") as it applied agile principles and practices while developing features for one of the largest commercially available pieces of software in the world, Microsoft's Visual Studio® development system. Scaling agile to very large projects has proven challenging because of the escalating coordination requirements between many small teams. The product unit used the feature crew model to decouple individual teams within the project. This allowed teams to operate in a more agile manner while still participating in a much larger undertaking.

1. Introduction

The Visual Studio Tools for Office product unit is one of 15 teams, comprising Developer Division, who collectively create features that ship in Microsoft's Visual Studio product. The product unit consists of 75 people including 30 developers and 30 testers with an additional off shore test team of six. The total number of people working on Visual Studio is around 1200. The complete Visual Studio codebase comprises over 43 million lines of code.

Shortly after the release of Visual Studio 2005 the product unit made the decision to reorganize their software development life cycle as part of a wider Developer Division initiative [1]. While the prior release had been very successful, improvements in the process could be made by ensuring that work was not deferred into the stabilization phase. While the product might reach "code complete" on schedule with the full breadth of features, "feature complete" with its requisite quality and feature depth was not achieved until very late in the cycle. This impacted the ability to maintain a stable main branch during development (thereby requiring additional work to manage this), to produce frequent Customer Technology Preview releases (CTPs), and to move towards the shorter product cycles desirable in this market. Deferring work also reduced the ability to accurately predict ship dates.

Previously the product unit had been organized with different functional management structures; these included development, test, and program management. The different structures tracked and reported work differently—leading to reduced visibility across the organization. We also took a milestone based approach, with milestones lasting several months. This led to large gaps in time between specification, development, integration, and testing of new functionality.

2. Our Approach

Developer Division adopted the feature crew model that was originally used for the development of Microsoft Office 2007. Feature crews are multi-disciplinary teams of five to ten people that own a single feature and work on it in an isolated branch called a feature branch within a tiered source code tree [2]. Any completed work must pass a series of gates or acceptance criteria in order to be considered "done" and integrated from the isolated branch to the parent branch. The isolation afforded by this branch structure and the use of a common definition of "done," as followed across Developer Division, allowed individual crews a great deal of flexibility in their day to day development practices.

The feature crew model mandates little regarding the actual process used by the crew to complete their work provided it passes a set of acceptance tests—the quality gates, see Section 3.2. This afforded the product unit the opportunity to try new agile approaches to our development process within the framework of the feature crew model.

3. Our Experiences

This paper does not seek to cover all aspects of our experiences. Instead it highlights select key areas that were particularly significant on a project of this size including:

- Feature crews and how the product unit used them to encapsulate agile teams.

- The use of a tiered source tree to isolate the development activities of individual teams and provide them a greater degree of autonomy.
- Our use of agile practices within teams on a daily basis.

3.1 The Feature Crew Model

For the next release of Visual Studio a new model of development was adopted called the feature crew model, as described in Section 2. Features are typically "epic" sized [3] pieces of user functionality delivered by a crew in five to ten weeks. Once a feature is completed and integrated into the parent branch, the feature crew is then disbanded and crew members are reallocated to start on another feature.

The product unit further refined the feature crew model by applying some additional agile concepts (or practices). We used end to end user experiences—comprised of several related features—to define the product. These experiences represented a contract between the product unit and the customer who was represented by the product unit's General Manager. Instead of breaking up feature crews once they were feature complete, the product unit chose to keep feature crews together to deliver select end to end user experiences over the entire ship cycle.

Thus far we have found this long-lived feature crew model to be quite powerful as it allows teams to become more effective over time rather than repeatedly reforming teams around small pieces of unrelated work. On the other hand, it has made load balancing of development work across the product unit difficult as it precludes moving people from one end to end user experience to another.

Each feature crew delivers their end to end user experience in a prioritized fashion, integrating features into the parent branch when acceptance criteria are met and that end to end user experience is ready for customer feedback. Scheduling and planning is based on a five week iteration cadence used by the Developer Division. Our crews adopted much of the thinking behind Scrum [4] by using daily stand-ups, backlogs, burndowns, and iteration planning meetings to plan and track progress. Crews were empowered to manage the day to day development decisions. Senior management acted on behalf of the customer and was only involved in approving changes to the end to end user experiences. At the end of each iteration, crews demonstrated complete features to stakeholders. The progress was evaluated against the larger end to end user experience owned by that crew.

As part of adopting the feature crew process we also wanted to improve our project management system to increase transparency across the organization. We built a unified set of tools for backlog management and reporting to surface the burndowns, blocking issues, and dependencies for the individual feature crews. Crews entered data into a Visual Studio Tools for Office enabled Excel spreadsheet which generated reports locally and also uploaded them to a SharePoint site.

One challenge we faced was projecting the corporate hierarchy of development, test, and program management onto cross functional teams. In addition to a reporting structure and meeting cadence, ground rules for management interference were also defined. The goal was to try to keep as much control in the hands of the crew itself while meeting the needs of management.

Organizationally, we tried to align the staff on a feature crew with discipline leads who were responsible for that feature crew. Each lead was responsible for two to three feature crews. In practice we found that two features crews was about the maximum number of crews a lead could be involved with because each crew was working in a different technology space.

Even with leads in place, the crew was still given latitude also to manage themselves. For example, leads were involved in daily stand-ups, as described by Scrum, as "chickens" not "pigs" [4]. The main function of the leads was finding the gaps between experiences, making sure that everyone on the feature crew had a fair distribution of challenging assignments, reporting feature crew status up to management, and ensuring the general happiness and health of each crew.

At the end of an iteration, each crew would come up with a plan for the next iteration. Their plan would include details of the next features they would build, whether the end to end user experience as proposed would change, a product backlog, and a demo goal. Individual crew iteration plans were first drafted without lead involvement. This allowed the crew to build, commit to and manage the plan.

Iteration plans were then reviewed by the leads associated with the feature crew. The plans were summarized by the leads and submitted to product unit management for final approval. This was an opportunity for senior management to set direction on an iteration basis and do load balancing across the different crews.

Sometimes as crews would roll up their plan for the next iteration to management, changes would be requested which resulted in "re-planning." This re-planning occurred when load balancing had to occur across feature crews or when changes to the user experience being delivered by the crew were deemed not up to standard. To mitigate the re-planning problem, leads and management started to "chicken-in" to crew planning meetings to influence the plan earlier.

3.2 Quality Gates

Another key component of the development process was the use of quality gates listed in Table 1 to ensure stability in the main product branch. For more on the branching model see Section 3.3. The gates ensure that features are complete and ready for customer feedback when they are delivered as part of a CTP after the quality gates are met. The gates reduce the possibility of work being deferred, avoiding the problem of hitting code complete with a much later feature complete.

The quality gates act as a compact between teams in different branches and product units ensuring a common definition of "done" across Visual Studio. Quality gates include a set of automated build verification test (BVT) suites contributed to by all the product units that are designed to make sure that teams working in different branches maintain core product stability.

The amount of work required to pass all the quality gates proved to be significant. Typically crews would spend several weeks focused on quality gate work prior to integrating a feature into the parent branch.

Table 1: Quality gates

Quality Gate	Description
Testing	All planned automated tests and manual tests are completed and passing
Feature Bugs Closed	Any bugs found in the feature are fixed or closed
Performance	Performance goals for the product are met by the new feature
Test Plan	A test plan is written that documents all planned automated and manual tests
Code Review	Any new code is reviewed to ensure it meets code design guidelines
Functional Spec	A functional spec has been completed and approved by the crew
Documentation Plan	A plan is in place for the documentation of the feature
Development Spec	A document describing the architecture and implementation is in place
Security	Threat model for the feature has been written and possible security issues mitigated
Samples	A sample has been written showing how a customer would use the feature
Static Analysis	Tools are run to analyze the code for security and other defects
BVTs Passing	The Build Verification Tests—a set of automated tests contributed to by all the product units in Visual Studio, are passing with the new feature in place
Setup Verification	Tests are run to verify the new feature can be installed, uninstalled, and serviced
Test Matrix	The new feature is verified to work on multiple operating systems and multiple versions of Office
Code Coverage	Unit tests are in place for the new code which ensure 80% code coverage of the new feature
Localization	The feature is verified to work in multiple languages

3.3 Tiered Source Tree

Visual Studio uses a distributed source development model where different teams submit source code changes into branches within a large three level tree. Features are developed in a branch called a feature branch which is then integrated to the parent product unit branch and finally to the root main branch from which the final product is built.

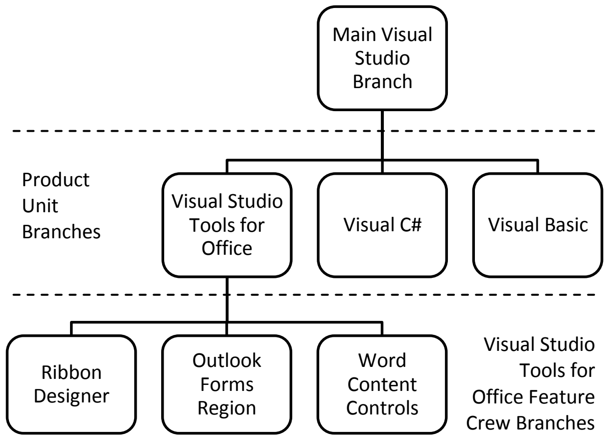
Main Visual Studio Branch
Product Unit Branches
Visual Studio Tools for Office
Visual C#
Visual Basic
Ribbon Designer
Outlook Forms Region
Word Content Controls
Visual Studio Tools for Office Feature Crew Branches

Figure 1. Feature branch layout

Feature crews develop their code in leaf branches of the tree. Crews can periodically take updates from the parent branch and merge them into their branch. This process is called a forward integration (FI) and was typically done every other week. To keep the parent branch up to date, forward integrations between the main Visual Studio branch and the product unit branches occur every other week.

Once a crew finishes all or some significant part of their feature, they integrate changes back into the parent branch. This is known as a reverse integration (RI). During RI, a feature crew must complete an FI and pass the quality gates before completing the reverse integration back into the parent branch.

The RI activity also takes place between the product unit branch and the main Visual Studio branch. The product unit branch is used as a staging area to verify that different end to end user experiences created by different feature crews interact well with each other. This verification was achieved by additional automated and manual testing. Finally, the product unit branch is integrated back into the main product branch from which monthly CTP builds are created and made available to customers.

3.4 Agile Development Practices

In addition to adopting a more agile process for this Visual Studio release cycle the product unit expanded its use of agile development practices within the feature crews. We had previously done some experimenting with unit testing, continuous integration (CI) and daily stand-ups, and saw positive results from all three practices. As a result, these practices have expanded and become core to our day to day development activities across the whole product unit.

For example, unit testing has been adopted on a far broader scale as we push towards the 2007 release of Visual Studio. We promoted its adoption with our developers both by providing education—talks and courses—and having key developers champion best practices. Our goal continues to be achieving maximum code coverage with unit tests where possible.

We also created tools to make authoring of unit tests easier. We automated the creation of skeleton Visual Studio Team System test projects within our source tree and integrated them with the command line test execution environment and code coverage tools. We planned to adopt CI fully during this release. Prior to starting the release the authors were involved in the development of a CI server application which was rolled out across Developer Division. Each feature crew was allocated dedicated hardware for running at least one CI server.

In addition to setting up a conventional CI build—build and run unit tests in ten or so minutes—we also used the CI servers to create a "defense in depth" strategy. The same CI machine could be used to do full debug builds and run automated acceptance tests frequently throughout the day. This allowed us to detect breaking issues in code an hour or so after it was checked in rather than waiting for the next test pass.

Our daily stand-ups started off being modeled after Scrum. While the basic format of a short daily meeting did not change, it was interesting to see them evolve over time as each team formed. The most noticeable influence on these meetings was the style of the team's program manager. Our feature crews were all co-located but usually occupied individual offices. Stand-ups provided a regular forum for tracking progress and face to face communication.

Practicing agile involves continuous improvement. Consequently we continue expanding our unit testing to encompass test driven development activities, and have sent more people to ScrumMaster training. There is also ongoing work to automate more of the quality gates and reduce the cost of integrating between branches. This will allow more frequent forward integrations and reduce the cost of reverse integrations.

4. Conclusions

Perhaps unsurprisingly the product unit found a lot of value in creating cross functional teams and having these teams own an end to end user experience that they delivered together. Development and test worked together more effectively: development helped with test tasks and test helped with development tasks. Any change to the product was quickly validated by test and evaluated against the specification by the program manager, thereby reducing the gaps between a feature being checked in and it being tested or evaluated against the specification.

Sizing the crews appropriately was another challenge to closely monitor. We sought to avoid making our crews too lean, which would result in their being heavily impacted by the overhead of quality gates and by team member absences. We also sought to avoid crews being too large, so as to promote the effectiveness of their stand-ups, enhance the focus on their end to end experience, and avoid the tendency of larger teams being called upon for emergency work items such as sustaining engineering work for previous products. Our choice to use long-lived crews formed around end to end user experiences, while promoting continuity and depth of knowledge within the crew, also presented management with the challenge to prioritize investment in one end to end user experience over another, as moving people between crews was disruptive.

Isolating features in branches prevented the inevitable disruption caused by having a large number of developers working in the same codebase. However it also decreased one crew's visibility into the work being done by another crew in a different branch and made getting interim updates costly. Tracking work that might have effects outside a single crew was a responsibility that was embraced by the leads. They tracked dependencies and looked for inconsistencies in end to end experiences beyond the scope of each individual crew.

The final product is the sum of each feature crew's end to end experience. Additional integration testing was performed to ensure that one experience integrated well with the rest of the experiences being developed in Visual Studio. We used our product unit branch as a staging ground for this integration and used two models for verifying a successful integration. One model assigned a feature crew to own integration testing and "application building" activities. The second model distributed the ownership across all the feature crews. Each model had its respective pros and cons. In the first the majority of crews were able to focus completely on feature work while a single crew had to put their feature on hold, in the second all crews were equally impacted by the this integration "tax" but could also continue with feature work. Servicing our existing products also presented a significant challenge for the team. Several models for servicing our existing products were used during the cycle. In one model a dedicated engineering team owned all sustaining work with input from the feature team. A second model used an offshore team to take over some major sustaining tasks and defray the impact on the product unit. A third model involved assigning sustaining tasks into the backlog of each feature crew. All of these models had their pros and cons and were used with varying levels of success.

We also discovered the importance of "slack" [5]. We found that feature crews that were afforded more slack spent their time constructively: continuously improving our tools and processes, developing and contributing to incubation projects, and better managing technology spiking. The test organization was particularly successful in this regard.

The transparency afforded by iteration reviews, backlogs, burndowns, and stand-ups proved invaluable and allowed for better course corrections on a more timely basis. The information collected in the burndowns allowed the crews to learn to better estimate and set more realistic load factors for subsequent iterations.

Forward and reverse integration schedules and processes were very complex and time consuming; thus, having a full time build engineer and full time release manager was tremendously helpful. Integration schedules were such that feature crews had to commit to an exact date, usually a few weeks out, for their reverse integration into the parent branch. This reduced the degree of flexibility as to scope, quality, and time, leading to strain on the only remaining variable—people—as they sought to pass the quality gates and make an integration window. In the future we plan to automate more of the quality gate and integration work and move some verification to the product unit branch level so as to enhance the working environment for, and relationships among and within, the feature crews.

Overall, we discovered that it is possible to mix agility and the inconceivably large. Despite the complexity of the Visual Studio product, we were able to greatly improve our ability to respond to customer needs and maintain a stable and high quality product throughout this development cycle. We were able to dramatically reduce the gap between features being implemented and features being verified, and were able to build more cohesive and empowered teams.

5. Notes

Ade Miller (ade@ademiller.com) was a Development Lead for the Visual Studio Tools for Office product unit and now works for Microsoft's patterns & practices group. Eric Carter is the Development Manager for the Visual Studio Tools for Office product unit.

The information and recommendations in this article represent our personal views and do not necessarily represent the view of our employer, Microsoft Corporation.

6. References

[1] B. Boehm, and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison Wesley, Boston, 2005.

[2] Jacob, John, Mario Rodriquez, and Graham Barry. "Microsoft Team Foundation Server Branching Guidance." Online Posting. 20 Mar. 2007. CodePlex. 18 May 2007 <http://www.codeplex.com/BranchingGuidance>.

[3] M. Cohn, *Agile Estimating and Planning*, Prentice Hall, 2005.

[4] K. Schwaber, *Agile Project Management with Scrum,* Microsoft Press, 2004.

[5] T. DeMarco, *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*, Broadway, 2002.