9

Lessons Learned at Microsoft Developer Division

We must, indeed, all hang together or, most assuredly, we shall all hang separately.

—Benjamin Franklin, upon signing the treasonous Declaration of Independence

FIGURE 9.1: At any one time, Developer Division has to balance multiple competing business goals.

I JOINED MICROSOFT DEVELOPER DIVISION (DevDiv) in 2003 to participate in the vision of turning the world's most popular *individual* development environment, Visual Studio (VS), into the world's most popular *team* development environment. Of course, that meant embracing modern software engineering practices for our customers.

At the same time, DevDiv faced significant challenges to improve its own agility. I had no idea how long a road lay ahead of our internal teams to change their culture, practices, and tooling. It has been and continues to be a fascinating journey.

In this chapter, we compare three waves of improvement that we undertook, each of approximately two years. The first big change in our practices was the transition from the release of VS 2005 with .NET 3.0 and VS 2008 with .NET 3.5,[1] where we focused on the reduction of waste and trustworthy transparency. Second came VS 2008 to VS 2010, where we emphasized flow of value. And most recently, from VS 2010 to VS 2012, where we shortened cycle time.

Scale

For context, let me review the scale of work. DevDiv is responsible for shipping the VS product line and .NET Framework. These major releases are used by millions of customers around the world. They have ten-year support contracts. They are localized into nine languages. More than 3,500 engineers contribute to a release of the stack. Our divisional Team Foundation Server (TFS) instance manages more than 20,000,000 source files, 700,000 work items, 2,000 monthly builds, and 15 terabytes of data.[2]

We are also continually "dogfooding" our own products and processes. This means that we experiment internally on ourselves before releasing functionality to customers. For example, we implemented the hierarchical product backlog I describe on TFS 2005 although TFS didn't really support hierarchy until its 2010 release, and our internal experience drove the TFS product changes. In the next chapter, we describe a breadth of practices we pioneered internally and will be releasing in vNext.

Like many customers at our scale, we had to customize our TFS process template, both to allow the innovations and to deal with specific constraints, notably interoperation with our own legacy systems. As we have been developing TFS, we have had to interoperate with five separate internal predecessors for source control, bug tracking, build automation, test case management, and test labs. These predecessor systems were all homegrown and designed over decades in isolation of each other, not to mention of TFS.[3]

Business Background

As with any organization, it's important to start with the business context. DevDiv provides tools and frameworks that support many different Microsoft product lines. Many of DevDiv's products, such as the .NET Framework, Internet Explorer's F12 tools, and Visual Studio Express, are free. They exist not to make money, but to make it easier for the community and customers to develop software to target Windows, Windows Azure, Office, SQL Server, and other Microsoft technologies. Other products, such as the rest of the VS product line and MSDN, are commercial, typically licensed by subscription.

An obvious tension exists among the business goals. Different internal and external stakeholders have very different priorities. And very frequently the number one top item for one constituency is invisible to other groups.

As I've explained this situation to customers over the years, I've realized that these sorts of tensions among conflicting business priorities are quite common. Every business has different specifics, but the idea that you cannot simply optimize for one goal over the rest is common. As a result, divergent business goals create conflicting priorities among stakeholders.

Scrum teaches us that the right way to reconcile these priorities is through a single product owner and common product backlog, and at this scale, we have to aggregate to coarser-grained portfolio items. When I started in 2003, prior to the availability of TFS, the division had no way to look at its investments as a single backlog or portfolio. No one (literally)

had the ability to comprehend a list of more than a thousand features. Accordingly, the primary portfolio management technique when I joined was head-count allocation. Head count, in turn, had become the cherished currency of status.

Culture

Microsoft has three very healthy HR practices:

1. Hiring the best, brightest, and most passionate candidates, usually straight from university
2. Delegating as much responsibility as far down the organization as possible
3. Encouraging career development and promotion through rotation into new roles and challenges

These practices make Microsoft a great place to work. In 2003 DevDiv, however, they were creating an unexpected consequence of reinforcing Conway's law, that *organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations*.[4]

USC Professor Dave Logan and his colleagues have probably studied company culture as much as anyone. In their book *Tribal Leadership,* Logan et al. identify five stages of organizational maturity. Stage Three covers 48% of the professionals that they have studied.

> The essence of Stage Three is "I'm great." Unstated and lurking in the background is "and you're not." …The key words are "I," "me," and "my."[5]

This dysfunctional tribalism was widely visible in the DevDiv I saw in 2003. At the time, the main organizational tribe was a "product unit" (PU), averaging roughly 60 people. The dysfunction was characterized by five behaviors, which I stereotype here:

- **Don't ask, don't tell.** There was an implicit convention that no manager would push on another's assertions, in order not to be questioned on his own.

- **Schedule chicken.** Scheduling was a game of who blinked first. Each PU self-fulfillingly knew that the schedule would slip because someone else would be late. Therefore, each PU kept an invisible assumption that it would be able to catch up during the other team's slippage.
- **Metrics are for others.** No PU particularly saw the need for itself to be accountable to any metrics, although accountability was clearly a good idea for the other guys because they were slipping the schedule first.
- **Our customers are different.** Because DevDiv has such a broad product line, with many millions of users of VS and hundreds of millions of users of .NET, it was very easy for any PU to claim different sources of customer evidence from another and to argue for its own agenda.
- **Our tribe is better.** Individuals took great pride in their individual PUs, and their PUMs (product unit managers) went to great lengths to reinforce PU morale. Rarely did that allegiance align to a greater whole.

Waste

In 2003, DevDiv experienced every kind of waste listed in Table 1.1 in Chapter 1, "The Agile Consensus." One illustration of this is shown in Figure 9.2. This chart shows the bug trends to Beta 1 of what became VS 2005. Different colors show different teams, and the red downward-sloped line shows the desired active bug "step-down" for Beta 1. This is a prescriptive metric with all the negative behavioral implications listed in Chapter 4, "Running the Sprint."

More important than the "successful" tracking of the step-down is the roughly flat line on top. This represents the 30,000 bugs whose handling was *deferred* to the next milestone, Beta 2. Imagine a huge transfer station of nondegradable waste, all of which has to be manually sorted for further disposal. This multiple handling is one waste from bug deferral.

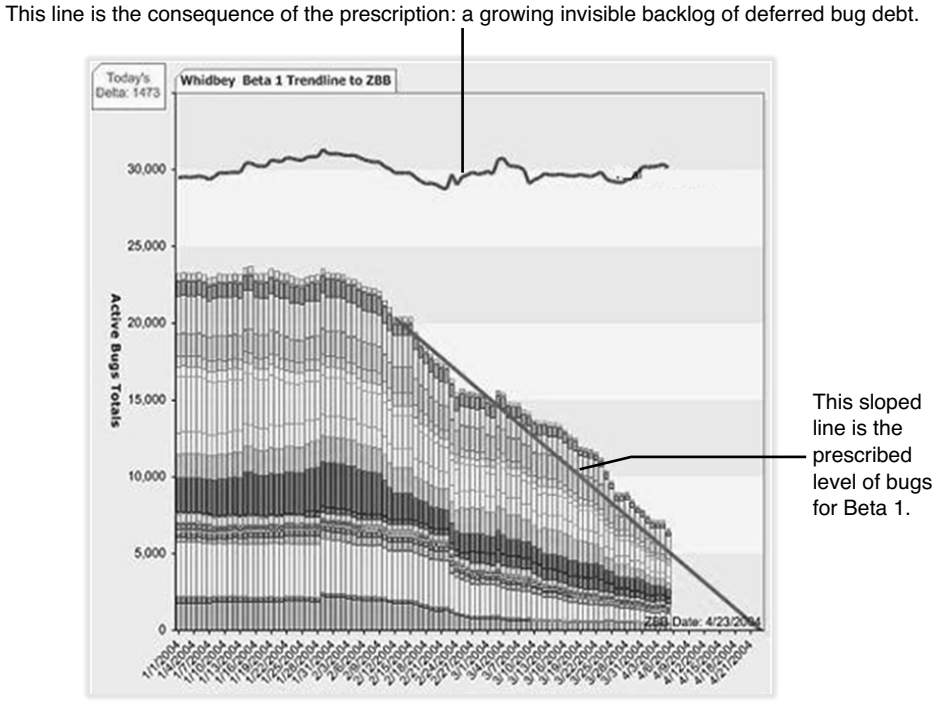

FIGURE 9.2: This chart shows the actual VS 2005 bug step-down, as of two weeks before Beta 1.

Debt Crisis

In everyday life, debt incurs interest, and the less creditworthy the borrower, the higher the interest rate. When the subprime lending bubble of 2003–8 burst, it clearly showed the "moral hazard" of not following such basic economic principles. Similarly, the uneven product quality implied by this high bug count creates its own moral hazards:

- It makes the beta ineffective as a feedback mechanism. Customers see too many broken windows to comment on the positive qualities.
- Internal teams see others' bug backlogs and play schedule chicken.
- Teams are encouraged to overproduce (that is, put in pet features) rather than fix the fundamentals.

- The endgame is very hard to predict. No one knows how much of the iceberg still lies below the water and, therefore, how much real work remains in the release.

Not surprisingly, we experienced significant schedule slippage in the 2005 release cycle and, by the time we shipped, had very uneven morale.

Improvements after 2005

So what did we do differently the next time? Broadly speaking, we put seven changes in place for the next release cycle. I cover each in turn:

- Get clean, stay clean
- Tighter timeboxes
- Feature crews
- Defining done
- Product backlog
- Iteration backlog
- Engineering principles

Get Clean, Stay Clean

Prior to the start of any product work, we instituted a milestone for quality (MQ for short). The purpose of MQ was to eliminate our technical debt and to put in place an engineering system that would prevent its future accumulation. The two main areas of technical debt we addressed were bugs and tests. Both of these were large sources of multiple handling.

The goal was to have zero known bugs at the end of MQ. This meant that any bug that had been previously deferred needed to be fixed (and validated with an automated regression test) or closed permanently. As a result, we would no longer waste time reconsidering bugs from earlier releases. This idea runs contrary to a common practice of seeding a release by looking at previously deferred work. It is enormously healthy; you start the new release plan from zero inventory.

The goal with tests was to have all tests run green reliably. Unreliable tests would be purged and not used again. In other words, we wanted to eliminate the need for manual analysis of test runs, especially build verification tests (BVTs). In the past, we had found that test results were plagued by false negatives (that is, reported test failures that were not due to product failures but to flaky test runs). This then led to a long manual analysis of test runs before the "true" results could be acted on. (Have you ever seen that?) Eliminating the test debt required both refactoring tests to be more resilient and making improvements to test tooling and infrastructure. (You can see the partial productization of these capabilities in the build and lab management capabilities of VS 2010.)

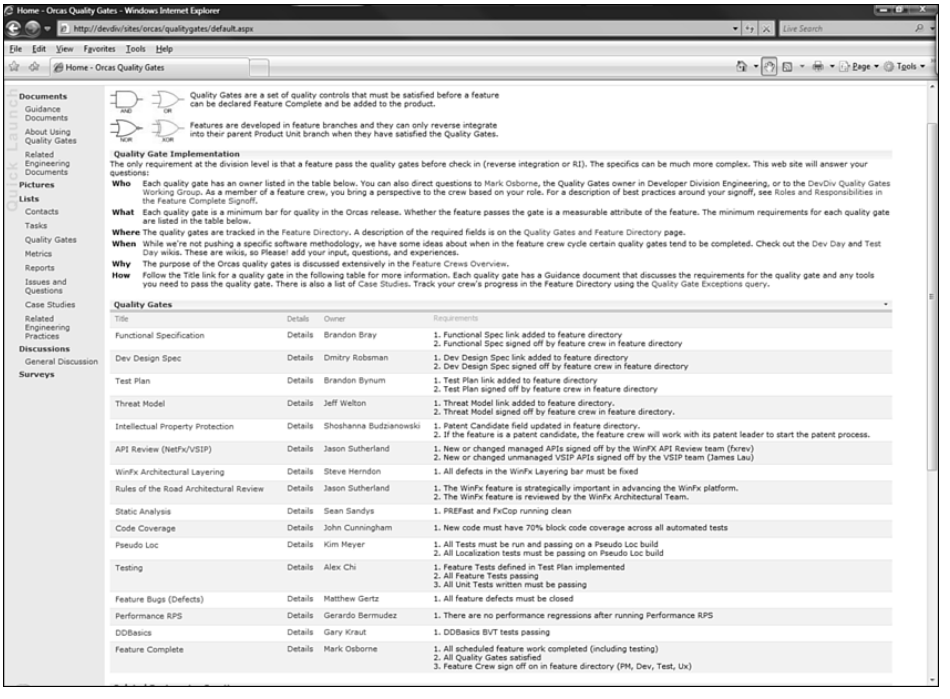

FIGURE 9.3: Divisional quality gates were the definition of done for the feature crew.

Different quality gates applied to different components of the product line. For example, redistributable platform components, such as the .NET Framework, required architectural reviews around compatibility and layering that were not necessary for the VS IDE. These rules were visible to the whole division.

Integration and Isolation

When developing a complex product like VS, a constant tension exists between the need of feature crews to be isolated from other teams' changes and the need to have the full source base integrated so that all teams can work on the latest code. To solve this, we allowed feature crews to work in isolated branches, and then to integrate with closely related crews, and then to integrate into Main. Figure 9.4 shows the branching structure to support the feature crews.

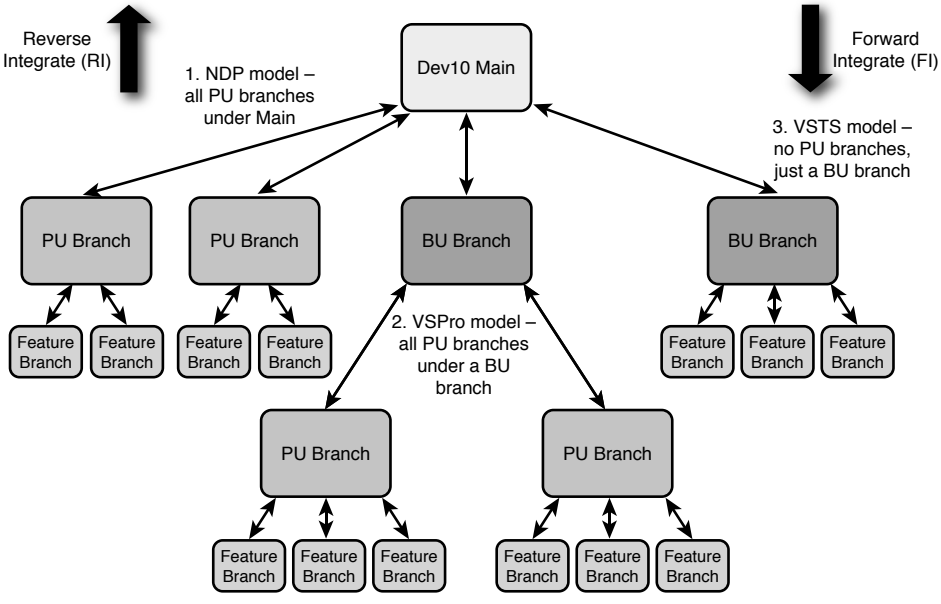

Figure 9.4: The branching structure balances isolated workspaces for the feature crews during the sprint with easy integration of related features by value propositions in a PU.

The third level of done was the integration tests to support the promotion of code across branches. When the feature passed the quality gates, the source code, and the tests, the feature crew promoted it to the "product unit branch," where the integration tests would run. Upon satisfying these, the crew then promoted the feature into Main and it became part of the divisional build. The fourth level of done criteria was applied to Main. Both nightly and weekly test cycles were run here.

It's worth noting that both the *branch visualization* and *gated check-in* of VS 2010, as described in Chapter 6, "Development," and Chapter 7, "Build and Lab," were designed based largely on our internal experience of enforcing quality gates for feature promotion. The core idea is to automate the definition of *done* and ensure that code and test quality live up to the social contract.

Product Backlog

DevDiv was an organization conditioned over a decade to think in terms of features. Define the features, break them down into tasks, work through the tasks, and so on. The problem with this granularity is that it encourages "peanut buttering" (as described in Chapter 3, "Product Ownership"), an insidious form of overproduction. Peanut buttering is the mind-set that whatever feature exists in the product today needs to be enhanced in the next release, alongside whatever new capability is introduced. From a business standpoint, this is obviously an endless path into bloat. This is a big risk on many existing products.

A key to reverse the peanut-buttering trend is the need to conceptualize the backlog at the right granularity. You have to test that proposed enhancements really do move customer value forward, when seen from the product line as a whole. At the same time, you need to make sure that you don't neglect the small dissatisfiers.

Accordingly, we took a holistic and consistent approach to product planning. We introduced a structure of functional product definition that covered value propositions, experiences, and features. For each level, we used a canonical question to frame the granularity. We rolled out training for all the teams.

Conceptually, the taxonomy is shown in Figure 9.5. To manage this data, we set up a team project in our TFS with separate work item types for each of the value proposition, experience, and feature.

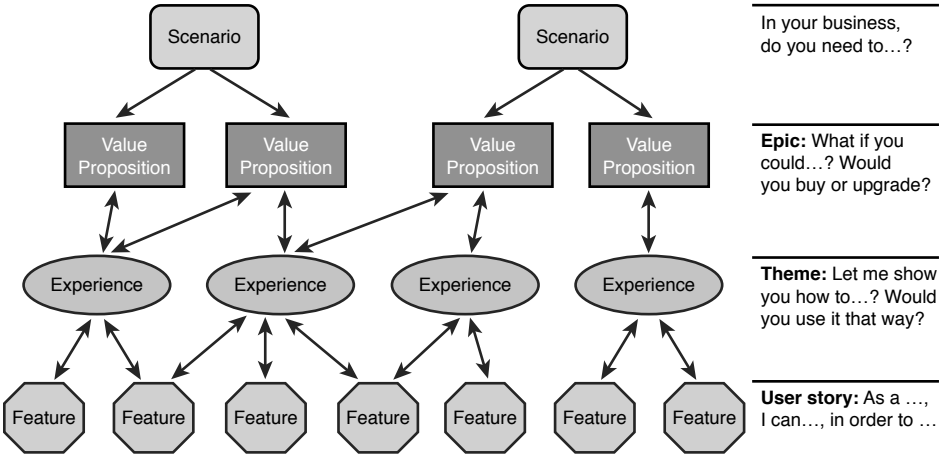

Figure 9.5: To keep the backlog at the right level of granularity for a product line of this scope, we used scenarios, experiences, and features, each at the appropriate level of concern.

Scenarios

In Agile terms, scenarios are epics. In a scenario, we start by considering the value propositions that motivate customers (teams or individuals) to purchase or upgrade to the new version of our platform and tools. We consider the complete customer experience during development, and we follow through to examine what it will take to make customers satisfied enough to want to buy more, renew, upgrade, and recommend our software to others.

A scenario is a way of defining tangible customer value with our products. It addresses a problem that customers face, stated in terms that a customer will relate to. In defining a scenario, we ask teams to capture its value proposition with the question: *What if you could…, would that influence you to buy or upgrade?* This question helps keep the scenario sufficiently large to matter and its customer value sufficiently obvious.

We also created two categories that didn't really belong to scenarios, but were managed similarly. These were called *Fundamentals* and *Remove Customer Dissatisfiers.* Fundamentals speak to ensuring that the qualities of service are suitably met. In the case of the VS product line, these include compatibility, compliance, reliability, performance, security, world-readiness, user experience, and ecosystem experience.

Remove Customer Dissatisfiers, in turn, was there to ensure that our users didn't "die from a thousand paper cuts." Plenty of small complaints can show up individually as either low-priority bugs or small convenience features, but can collectively create large distractions. If these items are triaged individually, they usually don't get fixed. This is an example of the aggregation of small items in the product backlog into meatier ones for stack ranking that I described in Chapter 3. Accordingly, we suggested a discretionary level of investment by teams in this area.

Experiences

Scenarios translate into one or more experiences. Experiences are stories that describe *how we envision users doing work with our product:* What user tasks are required to deliver on a value proposition? The test question here is to imagine the demo to a customer: *Let me show you how….*

Features

Experiences, in turn, drive features. As we flesh out what experiences look like, we define the features that we need to support the experience. A feature can support more than one experience. (In fact, this is common.) Most features are defined as user stories.

Figure 9.6 shows a top-down report of the product backlog. It is opened to drill down from a scenario (called *value propositions* here) into the experiences and features.

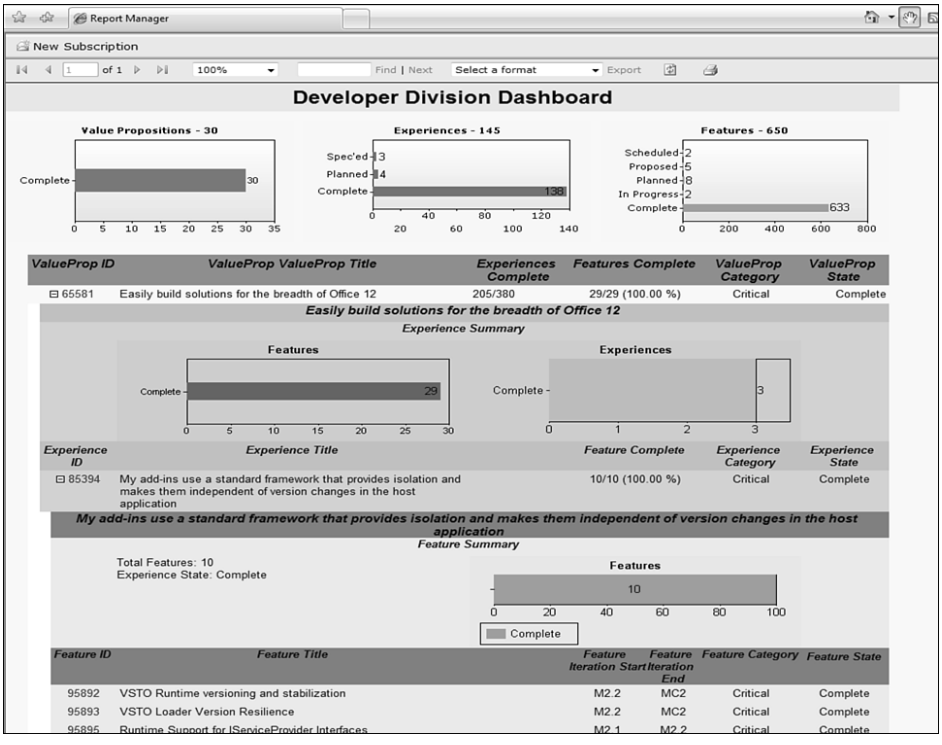

FIGURE 9.6: An internal custom TFS report showed the status of features, rolling up to experiences, rolling up to scenarios (value propositions). This has been superseded in TFS 2010 by hierarchical queries.

Iteration Backlog

Features were the connection between the product backlog and iteration backlog (see Figure 9.7). As we moved into a sprint, feature crews committed to delivering one or more features according to the quality gates. This affected how we had to define features when grooming the product backlog.

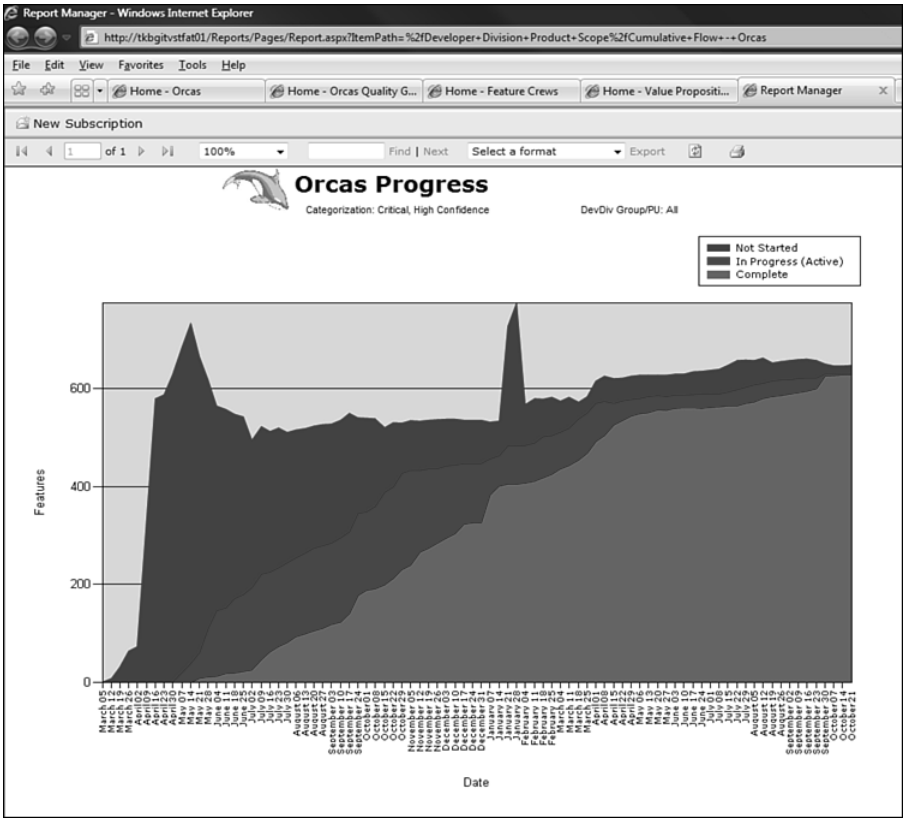

FIGURE 9.7: Because features were the deliverable units of the product backlog, overall progress could be tracked as a cumulative flow of features. Again, this was a custom report, now superseded by the TFS dashboards.

Because features turned into units of delivery, we tried to define them to optimize productivity. Well-defined features were *coarse-grained enough to be visible to a customer* or consumed by another feature, and *fine-grained enough to be delivered in a sprint.* To pass the quality gates, they needed to be independently testable. Dependencies among features needed to be clearly defined. Figure 9.8 shows a track of remaining work for a single feature, and Figure 9.9 illustrates an intermediate organizational view of features in flight.

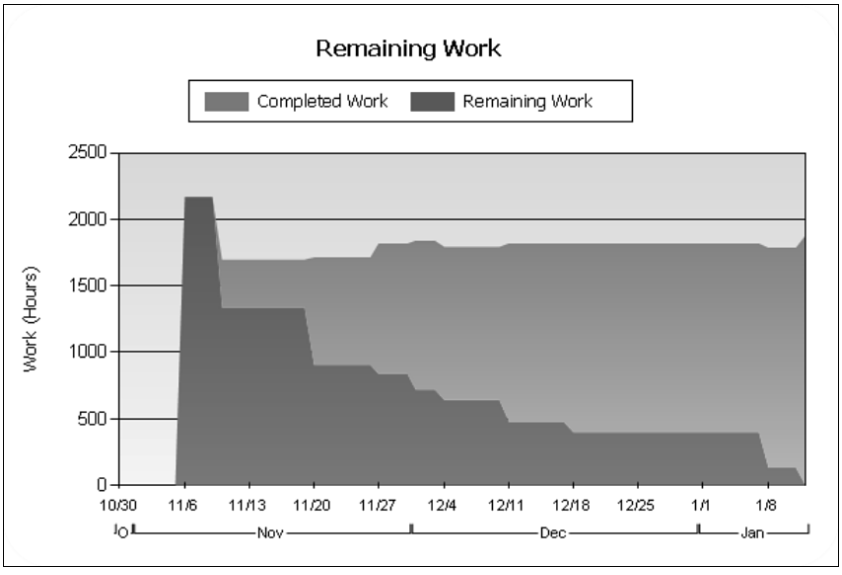

FIGURE 9.8: This simple burndown chart measures the progress of a single feature.

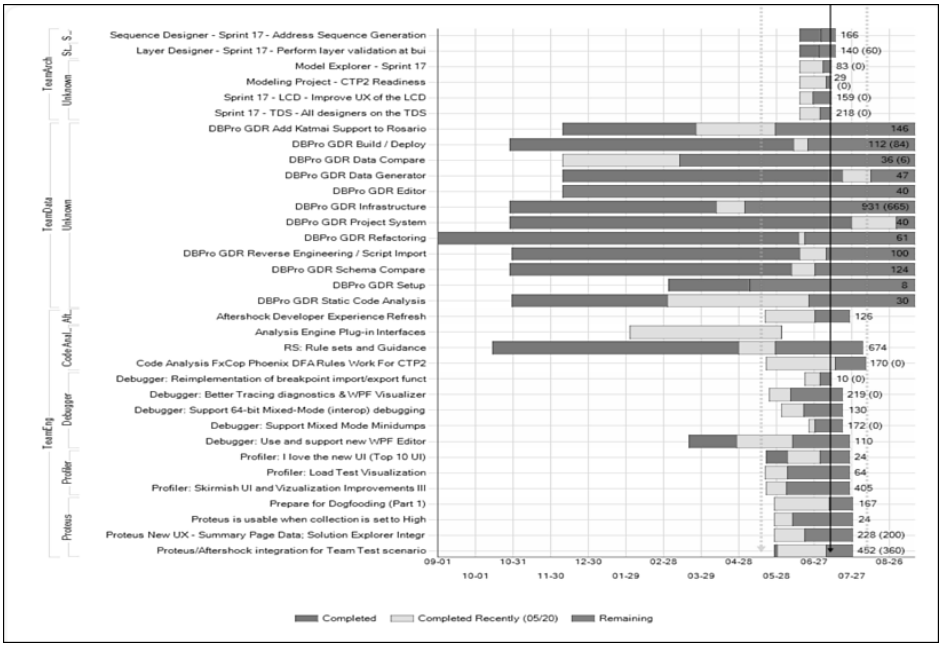

FIGURE 9.9: Features in progress can be viewed by organizational structure. The black vertical line shows today's date. There are three status colors on this chart: dark green for completed more than seven days ago, light green for completed in the last seven days, and red for remaining.

Engineering Principles

In summary, we applied most of the practices described as engineering principles in Chapter 2, "Scrum, Agile Practices, and Visual Studio." We eliminated technical debt and put in place rules and automation to prevent deferral of work. Small feature crews and short timeboxes kept work in process low. A consistent definition of *done,* coupled to the correct branching strategy and automation, kept the codebase potentially shippable. Automated testing was used widely, and exploratory testing was used selectively where new scenarios were not ready for automation.

Results

The results were impressive. Figure 9.10 shows the contrast between the bug debt at Beta 1 for VS 2005 and VS 2008. Unlike 2005, there is no overhang of deferral, and *the reduction in debt was greater than 15x.* At the same time, the schedule from beginning of the release work to general availability was *half* as long. And the transparency of process allowed reasonable engagement of stakeholders all along the way. Post release, we saw the results, as well. There has been a huge (and ongoing) rise in customer satisfaction with the VS product line.

In addition to improving our product delivery, this experience improved the VS product line. Many of the practices that we applied internally became product scenarios, especially for TFS.

Acting on the Agile Consensus

Having significantly reduced waste and improved transparency in the VS 2008 product line allowed us to act on the third principle of the Agile Consensus: We could improve the flow of value to our customers. As we entered the VS 2010 product cycle, we laid out several ambitious scenarios, based on the combination of customer requirements and what we discovered through our own usage. As you can see from the names in Figure 9.11, these were informed by the journey so far and have been described through the previous chapters.

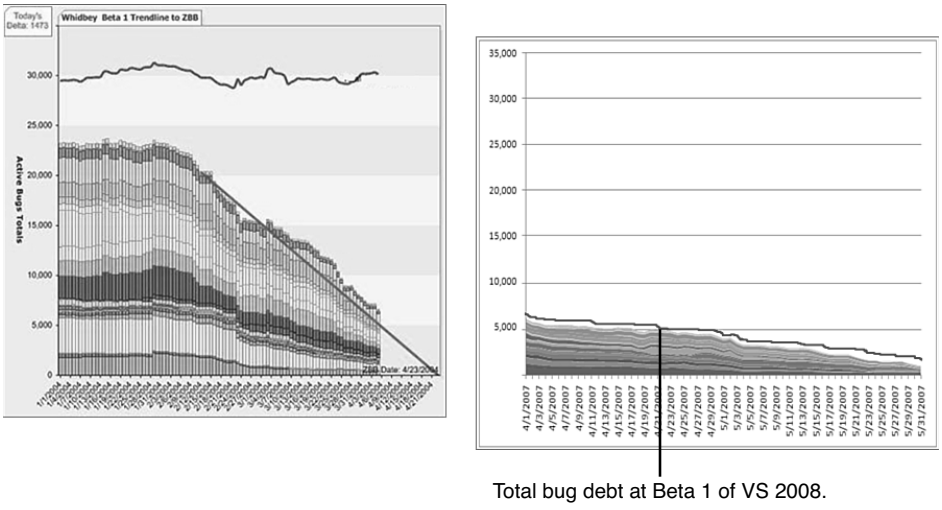

Figure 9.10: Comparison of bug debt at Beta 1 between VS 2005 and VS 2008. The left chart is identical to Figure 9.2, and the right shows the total bug debt leading to Beta 1 of VS 2008. The improvement is a reduction from 30,000 to 2,000 at comparable points in the product cycle.

Aspirational Lean Scenarios

What if you could experience No More...

No repro	Late surprises
Production mismatches	Build breaks
Waiting for build setup	Parallel development pain
UI regressions	Bewildering admin
Performance regressions	Butterfly effects or legacy fear
Planning black box	Code & fix
Missed requirements or changes	

Figure 9.11: The scenarios planned for VS 2010 show how the team embraced reducing waste and increasing transparency through the development life cycle.

We succeeded. VS 2010 achieved a level of customer recognition that was unparalleled. The product line won readers' choice awards, analyst acclaim, and market share, measured both in usage and revenue, to the point where it is largely the undisputed leader in Application Lifecycle Management (ALM) software.[6]

Lessons Learned

Although the improvements we achieved from the 2005 to 2008 and 2008 to 2010 releases of VS were very real, there were some subsequent surprises. Newton's third law states that actions beget reactions, whether good or bad. For DevDiv, some of these were due to "soft" issues around people and culture; others resulted from unforeseen effects of the engineering practices.

When we ship a release at Microsoft, people often change jobs. For employees, this rotation is an opportunity both to develop a career and improve personal satisfaction in trying new challenges. Indeed, several of Microsoft's divisions build a reorganization period into the beginning of their release planning. Although this is a healthy pattern for the company and its employees overall, in the short term it can create a sort of amnesia.

Social Contracts Need Renewal

Unfortunately, one success is not enough to create long-term habits. In 2008, DevDiv experienced excessive optimism after a successfully executed release. As many new managers took their jobs, they confidently plowed ahead without an MQ and without planning and grooming the backlog. Accordingly, the road to the 2010 release suffered from some considerable backslides.

It was reminiscent of a scene in 1981, when an assassination attempt incapacitated President Ronald Reagan, the vice president was abroad, and Secretary of State Al Haig convened the White House press corps to announce, "I am in charge here." Haig prompted wide and immediate ridicule because he demonstrated his own ignorance of the line of succession specified by a constitutional amendment two decades earlier. During the reorganization after we shipped VS 2008, some positions were vacant

longer than usual, and in the interim, several folks declared themselves in control of release planning. Of course, this self-declared authority did not work here either.

Lessons (Re)Learned

DevDiv recovered, and in the end, VS 2010 has been the best release of the VS product line ever. The progress was not linear, however. We learned several engineering lessons from the sloppy start in 2008 and skipping MQ in particular.

Product Ownership Needs Explicit Agreement

With ambiguous product ownership, there was no clear prioritization of the backlog and no way to resolve conflicting viewpoints. We did not yet have a consistent organizational process, and we needed to renegotiate the social contract.

Planning and Grooming the Product Backlog Cannot Be Skipped

If you don't have a backlog that provides a clear line of sight to customer value, all prioritization decisions seem arbitrary. As a result, individuals revert to the local tribes that they know best.

The Backlog Needs to Ensure Qualities of Service

A particular oversight was the lack of suitable requirements in the backlog around the fundamentals, such as performance and reliability, and lack of clear product ownership for these. With both betas of VS 2010, we earned significant negative feedback regarding product performance. Figure 9.12 shows sample results of the performance instrumentation that we introduced after Beta 1 to make performance visible for common customer experiences.

Fortunately, we recovered by the time of release to manufacturing (RTM), but at considerable cost (including some schedule delay). Had we set the fundamentals early, established the ownership, and put in place the instrumentation and transparent reporting at the beginning of the release cycle, we would not have had to pay for the recovery.

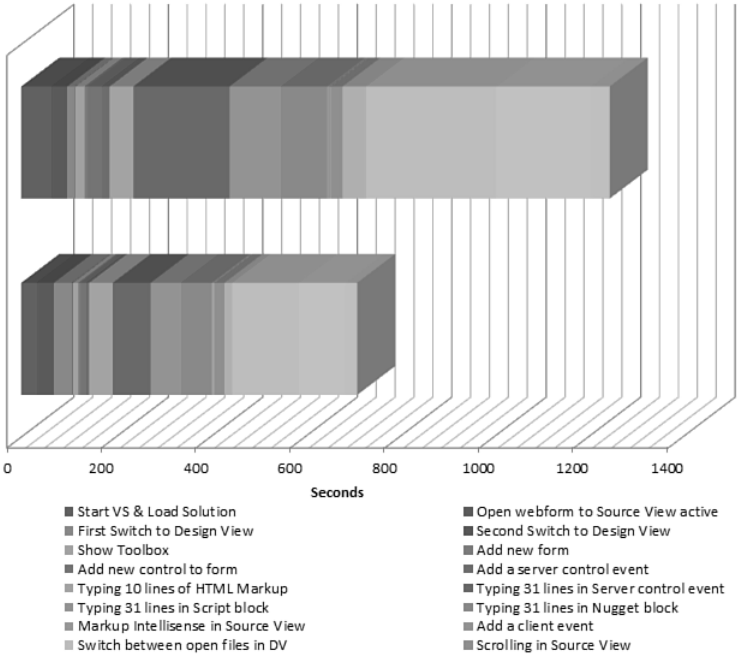

FIGURE 9.12: The chart compares an early build of VS 2010 and VS 2008 SP1 through a common scenario from starting the IDE to producing a simple application and closing the IDE. This is an example of transparent measurement raising awareness and focusing action.

One Team's Enhancement Is Another's Breaking Change

In a product line this complex, it is easy for one team's great enhancements to be crippling changes for another team. We had put a clear definition of *done* in place for the previous release, and automated much of it, but we didn't maintain the practices or the automation cleanly. Most visibly, our integration tests were not acting as a suitable safety net. As a result, we had significant friction around code integration.

Test Automation Needs Maintenance

We had invested heavily in test automation to validate configurations and prevent regressions, but we let the tests get stale. They effectively tested 2005 functionality, but not the new technologies from 2008. Without an MQ

to update the integration tests in particular, we discovered we could not predict the effects of integration. As a result, a team promoting changes had no way of determining the effects on other teams until the recipients complained.

Complicating the problem was the false sense of security given by automated test runs. If the tests are not finding important problems fast, and catching quality gaps *prior* to integration, they are the wrong tests.

Broken Quality Gates Create Change Impedance

There is a side effect to having the engineering infrastructure in this broken state. There were still quality gates, but they weren't ensuring the intended quality because we hadn't maintained them. As a result, they became impediments to change rather than enablers. As we realized this, we cleaned up the problem, but again much later than we should have. This pointed out clearly not only why it was important to *get clean* at a point in time, but also why we then needed to *stay clean.*

Celebrate Successes, but Don't Declare Victory

The overriding management lesson for me is to celebrate successes but not declare victory. In our case, we forgot the pain of the VS 2005 release, and after the success of VS 2008, we decided to skip MQ, neglect our backlog, and underinvest in our engineering processes. We have since recovered, but with the reminder that we have to stay vigilant and self-critical.

It takes strong leadership, a strong social contract, and consistent language among the tribe to counteract this tendency. Part of this is the move from dysfunctional to functional tribalism, or in Dave Logan's terminology, from Stage Three to Four. People in a Stage Four organization do the following:

> Build values-based relationships between others. At the same time, the words of Stage Four people are centered on "we're great" … When people at Stage Four cluster together, they radiate tribal pride.[7]

The Path to Visual Studio 2012

The Visual Studio 2012 wave represents the third major phase of improvement for DevDiv. We had worked on reducing waste, increasing trustworthy transparency, and expanding flow of value. Once you have mastered the three principles of the Agile Consensus, the next challenge you face is to get better at expanding continuous flow. We believe that there are two key actionable metrics here: cycle time, how long it takes to turn an idea from the product backlog into working software in customer's hands, and mean time to repair (MTTR), the interval from an unwanted event in production to the root cause being fixed and the service redeployed and in use by the customer. These metrics are shown in Figure 9.13.

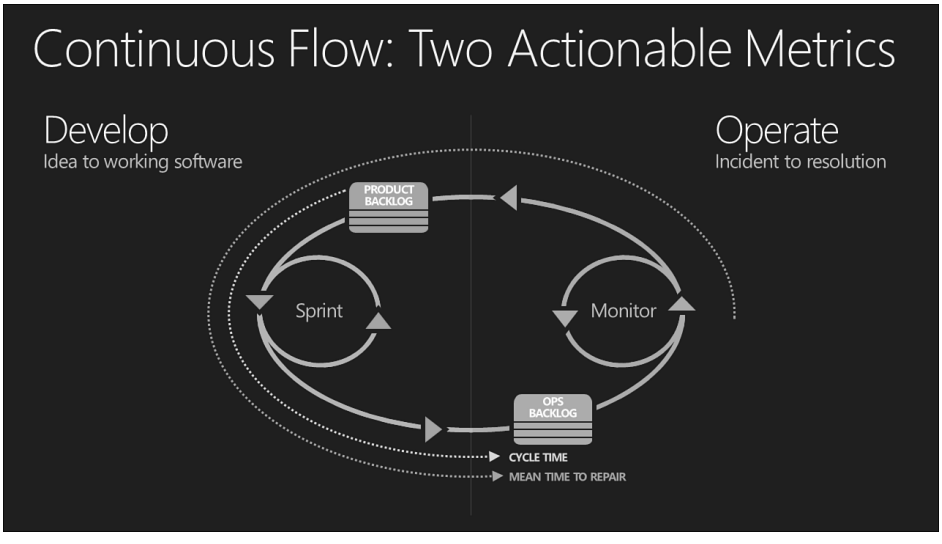

Figure 9.13: The two key metrics for improvement of continuous flow are cycle time and MTTR.

During the development of Visual Studio 2012, teams worked in synchronous sprints of three weeks. At the end of each sprint, we would update our "pioneer" server of TFS, and new builds of the client IDE were available daily intrasprint. Because of the earlier practice improvements, builds were just expected to work.

We coordinated across geographies by recording the sprint review demos as videos, individually accessible by PBI. That way, colleagues in

Hyderabad, India, could see progress in Redmond, Washington, without the entire cohort having to stay up late into the night. (One representative from each site would be present at the other site's review.)

We would also extend our sprint reviews to include customers. Through a regular cadence of Web calls with our customer advisory councils, we would use the video demos and extend the sprint review into a broader panel to drill into areas where the team needed more feedback.

The Visual Studio 2012 wave coincided with a shift in the industry from on-premises deployment of software to the early use of the public cloud. Microsoft's public cloud, Windows Azure, is one of the major "Platforms as a Service" (PaaS), and we used that to pioneer a new form of TFS, Team Foundation Service, which became available as a developer preview in September 2011.

Team Foundation Service raised the bar on our expectations once again. In the past, at the end of every three-week sprint, code was ready for internal "dogfooding." On the Service, every three weeks it is deployed live for customer use. And because we are connected to our users, we monitor Twitter and email continually as a support channel. Although we have extensive monitoring through instrumentation, synthetic transactions, and points of presence, an individual customer may still be the first to experience a problem. As soon as the customer notifies us, we can ask for permission to attach to the customer's instance and remediate, and we can measure our ability in minutes.

Endnotes

1. For simplicity, I refer to these as VS 2005 and 2008, without differentiating the .NET platform components, the VS IDE, TFS, or the ALM components formerly known as Team System. I also skip the dozens of power tools and releases of Internet Information Services (IIS), ASP.NET, Silverlight, and so on that shipped in between the major releases.
2. There are approximately 30 other instances in Microsoft, but I'm writing here about DevDiv, where I have firsthand experience.

[3] You can download the process template we used internally from http://mpt.codeplex.com/. However, the process templates that we ship are much leaner, take advantage of the 2010 features, and aren't tinged by the internal constraints.

[4] Melvin E. Conway, "How Do Committees Invent?" *Datamation* 14:5 (April, 1968): 28–31, available at www.melconway.com/research/committees.html. Amazingly, Conway's law was completely anecdotal for 40 years, until empirical validation by Microsoft Research in Nachiappan Nagappan, Brendan Murphy, and Victor Basili, "The Influence of Organizational Structure On Software Quality: An Empirical Case Study," January 2008, available at http://research.microsoft.com/apps/pubs/default.aspx?id=70535.

[5] Dave Logan, John King, and Halee Fischer-Wright, *Tribal Leadership: Leveraging Natural Groups to Build a Thriving Organization* (New York: HarperCollins, 2008), 77.

[6] For example, see Gartner ALM Magic Quadrant, published May 2012, available from www.gartner.com.

[7] Logan, *op. cit.*, 255.