Done and Undone

Visual Studio 2012

By Ken Schwaber and David Starr, Scrum.org

January 2012

Delivering a done Increment is critical to being successful with agile software development. Using real-world and theoretical examples, the authors demonstrate the difference between perception of "done" and the reality of "done," and how that affects the success of a project. Using these examples, the authors go on to demonstrate tools and strategies that can help teams start with a definition of done that makes sense for them, and methods to help teams communicate dependencies, status, and the meaning of "done."

Applies To

Application Lifecycle Management; Team Foundation Server

In this article:

- Introduction
- Transparency Lost at Anna's Company
  - What People Thought Was Happening
  - What Actually Happened
  - The Lesson
- Technical Debt at Nanomedtronics AZ
  - What People Thought Was Happening
  - What Actually Happened
  - The Lesson
- Technical Debt Amplifies with Multiple Teams
- The Release Plan at A Datum Corporation
  - What People Thought Was Happening
  - What Actually Happened
  - The Lesson
- Large Scale Techniques for Getting Done
  - Scrum of Scrums
  - Continuous Integration
- Conclusion

Scrum is an iterative, incremental agile framework. Teams use it to rapidly deliver Increments of “done,” or potentially usable software functionality each iteration, or Sprint.

“Done” is a simple yet often misinterpreted word. To me, it means finished, finalized, and completed. Being done means that there is nothing left to do. Done is simple to define; however, delivering a done Increment remains one of the fundamental and more elusive requirements of Scrum and agility.

Agility requires delivering done, ready-to-use Increments of working software each Sprint. Yet most Scrum and agile teams generate partially done, incomplete Increments. When a Scrum Team is asked why Product Backlog requirements were not completely done in a Sprint, team members often reply, “We didn’t have time.”

This paper addresses issues related to done Increments through the example of true case studies from the early days of Scrum. The names and places of those involved have been changed but I am sure the individuals will recognize themselves and their hard work. Throughout this article all of the Sprints are a monthly iteration, unless noted otherwise.

Transparency Lost at Anna’s Company

Anna needed to automate her department’s receipt of property title changes. The company she worked for built and operated gas pipelines across most of North America. Her department processed and paid royalties to the people who owned the land it crossed. The title information Anna’s department received was printouts or property change papers. The volume was becoming overwhelming, so Anna wanted to automate the feeds and royalty payment process.

Our Development Team proposed that we build the royalty system for Anna using Scrum. This ensured she would have a usable piece of software to inspect each month. She also had the right to change her mind each month about what our team would do next.

By the end of the third Sprint, we had a feed from one of the provinces up and integrated with older data. We demonstrated it using a simple SQL solution. Anna was delighted, because most of her staff’s Product Backlog was from this province.

She wanted us to teach her staff SQL so they could immediately begin using the software the Development Team had delivered.

What did she mean that she wanted to use it? Surely she hadn’t interpreted being finished with a Sprint as being done with the software!

We told Anna this as carefully as possible. She was livid. “What do you mean it isn’t done? I saw the first Increment, the second Increment, and now I want to start using it. Deploy it and teach us SQL so we can use it.”

We started feeling uneasy. We told Anna that, yes, it was done. But, it wasn’t that type of done. It was done to show to her, but there were issues that still needed resolution before the software could be used:

- Some incoming records could not be processed. We needed a facility to store and manage them, rather that throwing them away.
- Several of the fields in the incoming records appeared to be used for purposes other than documented. What should we do with them?
- Fields in the existing database contained pointers or information that looked like reference information. This was all over the database.
- When we were running the incoming feeds and querying the database, the system crashed several times. It looked like data was corrupted during these crashes.

Anna asked us how long it was going to be before we could make it her type of done, a usable done. We estimated another two Sprints were needed to make the Increments usable. She told us to go ahead and get it ready for her department's use. She then "asked" me to meet with her the next morning.

The next morning Anna, her boss, and the director of development were there. They expressed their disappointment that the transparency that I had touted was not present. They felt that I should have handled the unresolved issues differently than just recording them as bugs. They were disturbed because the progress reflected in the burndown reports provided to everyone was incorrect.

After the meeting, our marching orders were:

1. Investigate and correct the four bugs.
2. Finish and deploy the first three Increments so Anna's department could start using it.
3. Improve our engineering skills and test automation so our definition of done was the same as Anna's definition (immediately usable by the business).
4. Change the way we recorded defects so that the requirement would not be considered done unless they were fixed.

   We were told that this would be a good learning opportunity, if we all learned our lessons.

What People Thought Was Happening

When we developed a baseline plan for the system, it represented what stakeholders and Anna thought would happen. The Development Team reported progress that looked as though the release was on track, and people trusted the report.

The Development Team actually thought they were doing the right thing by showing that work was proceeding as prescribed in the plan.

What Actually Happened

Velocity is the measure and historical record of a Development Team's productivity per Sprint. Velocity is measured each Sprint and used to establish patterns of productivity.

Our Development Team needed a sustained velocity of 8 completed units of work each Sprint to meet the plan. When something threatened to slow our velocity below 8, we didn't finish all of the work on those items.

We delivered functionality that worked pretty well, but wasn't complete enough to use or build upon. We intended to improve it later. When we went back to estimate the undone work, it added another 14 units of work.

Given how difficult the initial title feeds were, we reworked the entire plan to reflect a likely twenty-month schedule. Anna's department released Increments every two or so months, enabling new feeds. The new automated feeds reduced overall manual work so greatly that it was anticlimactic when the system went live twenty-two months after it was started.

The Lesson

True transparency requires that everyone viewing the Increment must see and understand the same thing. Transparent inspection of the Increment should have allowed Anna to manage risk and gain predictability. However, since the Increment was not transparent, she couldn't effectively plan.

At the start of the project, Anna established a release goal. After Sprint 1, she assessed progress toward the goal by inspecting what she thought was a usable Increment. She made a decision about what to do in Sprint 2 based on the incremental progress toward the goal. If she had thought our progress was inadequate, she could have cancelled the project.

At the end of Sprint 3, Anna believed that 3/10 of the total was done, which was obviously incorrect.

Unfortunately, we had only done enough of each Increment to demonstrate. The undone work made the Increments unusable to Anna's department and opaque to inspection.

Opacity when inspecting an Increment is like covering a thermostat with a cold, wet washcloth. The thermostat doesn't have the correct understanding of the actual room temperature, and would incorrectly initiate heating when it should be cooling.

Without transparent Increments, the stakeholders don't have a correct understanding of what is actually happening, and may incorrectly take actions that don't make sense.

In short, without full transparency, the ability of the teams to inspect and adapt effectively is lost.

Technical Debt at Nanomedtronics AZ

Technical debt is the deferred work that must be completed before software is considered done. Technical debt takes many forms such as poor design, code duplication, and untested features. The following example demonstrates the cause and impact of technical debt as undone work over the course of a project.

Nanomedtronics AZ was a small startup software company. They had one Scrum Team working on a new release of their life-critical product; software used by nano robots to clean the clogged arteries of patients suffering from high blood-pressure.

What People Thought Was Happening

When the team got started, they were tasked with selecting Product Backlog items to turn into something done (no more work remains, useable, potentially shippable) in a one-month Sprint. The Development Team had all the skills to fully develop the requirements into a done Increment.

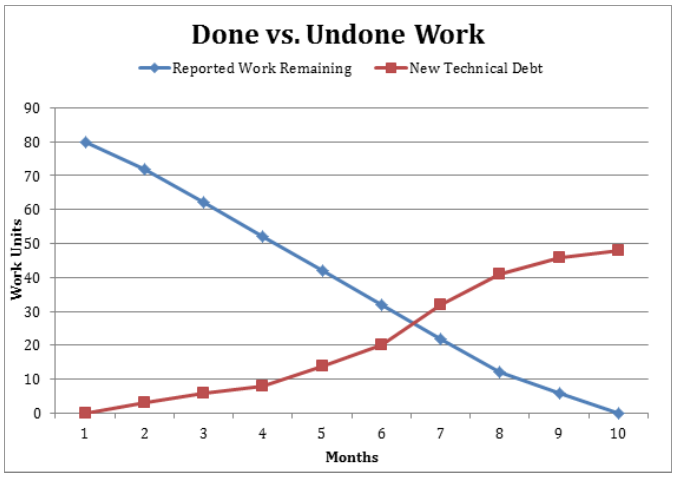
As the Scrum Team started the first Sprint, they saw that there were 80 units of work that must be completed in 10 months. Accordingly, the Development Team dutifully selected 8 units of work each Sprint. They reasoned that if they merely pulled 8 units per Sprint, they would be done in the 10 months mandated by the company.

The Development Team showed a working Increment at the end of each Sprint. By all outward appearances, Scrum was working and Nanomedtronics AZ was on track to deliver their product as planned.

What Actually Happened

What wasn't clear beyond the Development Team was each Increment delivered actually included some poor implementations, non-critical bugs, repeated logic, and generally messy code. Further, the Increments hadn't been fully tested because the Development Team stopped testing when pressed for time in a Sprint. The Development Team committed to meeting the schedule, and cutting quality was often the way to stay on course.

The team worked hard and built their product over the 10 months. The customer was delighted and prepared to implement and use the software. However, the Development Team had accumulated 48 units of undone work, shown in the following figure as new technical debt.



The team at Nanomedtronics AZ did not consider all the activities and work it would truly need to accomplish to be done. The following includes things the team failed to consider, and is by no means exhaustive. There are many more things that could be included.

- Analysis
- Design
- Dependency analysis
- Performance testing
- Stability testing
- Refactoring
- Immunological response testing
- Integration with the work of any other teams working on a product
- Integration testing with the work of any other teams so the Increment is the totality of all of the team's contributions
- Release notes
- Internationalization to the six cultures where the product will be sold
- User acceptance testing
- Regression testing
- Code reviews

The above work must be done to create a complete, integrated, Increment by the end of the Sprint. However, most Development Teams do not complete all of the above work. They leave behind some "undone" work every Sprint. This creates Increments with some poor design, duplicated code, overly complex logic, untested functionality or capability, or other form of incompleteness. This is how teams create technical debt in software.

Nanomedtronics AZ learned their product included all features needed to deliver it to customers, but also included many defects and lacked the packaging and finishing work needed to actually take the product to market. The Development Team had inadvertently created a backlog of additional work that would take another 6 months to complete, assuming an already incorrect velocity of 8 units per Sprint.

Waiting 6 months to ship the product was not acceptable to company leaders, and the product shipped with undone work remaining after only 3 months. The lost potential was not just the 3 month delay in shipping the product, but in the slowed ability to add new features as the Development Team now had to struggle with technical debt in future Sprints.

The Lesson

Technical debt obscures true progress and clouds the transparency required for empirical decision making by the Product Owner and stakeholders. Technical debt will be paid back, either in deliberate time spent correcting technical issues, or in loss due to shipping late or with poor quality.

In most situations, at least 50% of undone work remains in products when they are released. Accordingly, undone work becomes institutionalized as ongoing debt. Technical debt rapidly causes product fragility and ultimately may force negative business decisions like investing in rewriting software or abandoning a product.

Technical Debt Amplifies with Multiple Teams

A Development Team must carefully choose only as much work as it can do in a Sprint. The Development Team learns how much work this is through experience. Still, a team has to employ modern engineering practices like automated build and regression testing to accomplish much of anything. If these are not employed, the manual work tends to overwhelm a team by the fourth or fifth Sprints.

Consider a Development Team of three programmers and two QA specialists. Every day the programmers are checking code into the source code system. It is being tested and bugs are detected and given to the right programmer. Time elapses between checking in code and defects being detected and reported. During that time, the other programmers may have checked in code on top of the faulty code. The effort required to fix the initial problem is now greater. If the Development Team had a continuous build and test facility, the error would have been detected immediately. People could have investigated it, fixed it, and then proceeded. Extra work and waste could have been avoided.

Many organizations use more than one Scrum Team to build software. When this happens, the technical debt problem described in the section above becomes greatly amplified. The opportunities for checking in code on top of faulty code are significantly greater. The cost to remediate the increasing fragility of the software grows exponentially as work progresses.

The Release Plan at A Datum Corporation

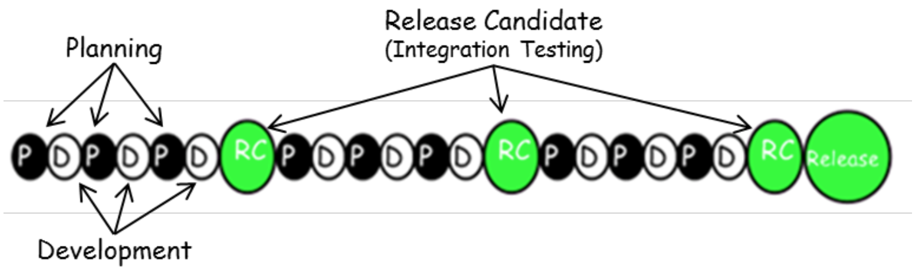
I recently worked with another company I'll call A Datum Corporation, an infrastructure software company. The main product line employs over 800 developers, including 250 programmers. The development infrastructure was partially automated and partially manual. Testing often lagged coding by days. The time between a programmer checking in defective code and it being detected and reported often was ten or more days.

To minimize the complexity of so many programmers, each Development Team worked on its own code branch. The development managers helped organize the Product Backlog requirements to minimize the dependencies. If each Development Team merged its code into the main code line every day, the amount of potential rework would be minimized.

However, management felt this would take too much time. Management decided to merge all code branches into the root every third Sprint. Integration tests would find any defects, which would then be fixed. A release candidate would be ready by the end of every third month.

What People Thought Was Happening

The following figure shows the planned release schedule and cycle.



The original plan assumed:

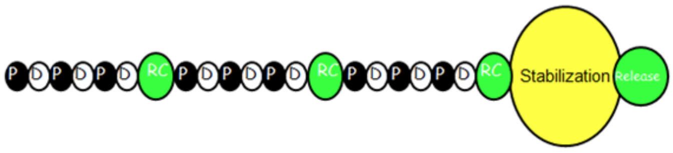
- 9 Sprints
- 3 release candidates and then a full release
- An 800-person development organization

What Actually Happened

When this organization got to the release date after nine monthly Sprints, the product was not ready for release. The sixth release candidate had over 5,000 defects and over 2,000 Product Backlog requirements were incomplete. We wondered how this could be. We had seen a release candidate every third month. When we investigated, we found that the demonstration had been from each Development Team's code branch. No integration had occurred. No integration testing had occurred.

To maintain the velocity required for the release date, the Development Teams had deferred any integration work, thereby creating a large amount of technical debt. The result was an eight-month slip from the scheduled release date. The words "release candidate" had no meaning.

The following figure shows the real project plus time needed for stabilization.



The release candidates had partially working functionality from the code branch for each team. Five months of "stabilization" was required prior to release. One particularly pesky defect that delayed delivery more than others was "poor performance." This issue was logged in the first Sprint.

The Lesson

Different teams working on the same software will eventually merge their work. Integrating software to ensure it works reduces the risk of the integrations and should occur frequently.

Waiting to merge the work of multiple teams is tempting as it allows delaying the cost of the merge. This ultimate cost of the delay however, is expounded by the number of teams involved and the number of branches that must be integrated.

Large Scale Techniques for Getting Done

Reaching a “done” state across multiple teams is hard. The complexities involved are numerous. Dependencies between teams and across code branches exist. Although this complexity of scale has cost, it is achievable and offers tremendous value when synchronized teams deliver in a rhythm together.

There are several techniques I’ve found helpful when multiple teams work together.

Scrum of Scrums

When many Scrum Teams are working on the same project, they need a technique to coordinate their work. I have recommended a “Scrum of Scrums.” This is a daily event, held immediately after each team’s Daily Scrum. Someone technical from each team attends. Each team representative describes what his or her team just worked on. He or she then describes what they plan to work on during the upcoming day. Based on this information, the representatives attempt to identify any rework needed, any upcoming dependencies, and any work that may need to be rescheduled.

The Scrum of Scrums has been useful to many organizations. However, it is inadequate. Dependencies and rework may or may not be successfully identified, because the problems are anticipated rather than known. Unanticipated dependencies caused rework and failing tests. Some teams spend more than 50% of each Sprint working and reworking the dependencies.

Continuous Integration

Extreme Programming (XP) requires continuous integration and integration testing of a team’s work. Why not extend this to all teams? Whether there are two teams or fifty teams, the teams are required to produce an integrated, integration-tested Increment at the end of every Sprint. To do this, the teams have to frequently integrate their work. Since any un-integrated work may potentially contain unresolved dependencies, continuous integration is the best solution.

Continuous integration of all work is similar to lean production techniques. In lean production lines, many techniques are used to assess the quality of a product throughout the production process. When deviations or quality issues occur, the production line is halted. Continuous integration and integration testing provide the same techniques for software product development.

Hard as it is, I recommend that every team and team member stop coding if a continuous build or test fails. Any continuing work is potentially building on top of the faults, causing a ripple effect of errors. If continuous integration is used, teams work closely together to avoid integration failures. Their work habits improve, waste is reduced, and quality increases.

Most organizations adopting Scrum do not begin with all of the engineering skills and tooling to build a done Increment. A rigorous program to achieve done Increments must be started and pursued. Groups of fewer than fifty people can rapidly reach a state where they are completing a done Increment within the Sprint. Organizations of over five hundred developers often require several years to get to that point.

Undone Increments create waste and prevent teams from reaching true agility. The true cost of undone work is far more than the lack of a feature or function in the Increment. The cost includes the wastes of re-planning and rework necessary when an Increment is not truly done, as well as costs of lower predictability and higher risk.

Many organizations want the benefits of agility, and employ Scrum to get them. Delivering a done Increment is critical to being successful with agile software development. Teams should start with a definition of done that makes sense for them, and then deliberately expand the definition over time. Only then, will they achieve genuine agility.

See Also

Concepts
Agile Planning and Iterations
Get Started as a Team
Create or Add to the Product Backlog
Groom and Estimate the Backlog
Run an Iteration
Finish an Iteration