Eleven Lessons Learned about Agile Hardware Development

Kevin Thompson, Ph.D.
www.kevinthompsonphd.com

January 11, 2016

Abstract

A previous paper by the author, *Agile Processes for Hardware Development*, argued that a Scrum process should be effective for development of electronic and electromechanical devices. This paper presents a case study of Thermo Fisher Scientific's experiment with using Scrum to develop equipment for analytical chemistry. The paper describes the background, the transition to Scrum, and lessons learned from this very successful experiment.

Acknowledgments

The author would like to thank Marisa Richardson, Jean-Jacques Dunyach, August Specht, Michael Bedford, Ray Chen, and all members of the Scrum Team that pioneered Agile Hardware Development with Scrum so effectively at Thermo Fisher Scientific.

1 Background

From late 2013 to mid-2015, I conducted research into Agile techniques for hardware development with two partners, John Carter (TCGen) and Dr. Scott Elliott (TechZecs). We interviewed numerous companies to discover what Agile techniques they used, and what issues and constraints they faced.

The information we collected led me to a conclusion that was quite different from my expectations: A Scrum process should work well for hardware development. This research led to my paper, *Agile Processes for Hardware Development*,[1] which presented the research findings, and provided explicit guidance on how to develop hardware with a Scrum process.

I'll leave the fine details of how Scrum works to the above paper, but need to summarize a few concepts that are important for the remainder of this paper. This paper will use the following terms, as defined below.

Development Cycles

- Sprint: Short period (2-4 weeks) in which a Team starts and completes a set of tested and working deliverables
- Release: A longer planning horizon (months) consisting of a set of contiguous Sprints

Roles:

- The *Product Owner* defines the product and drives requirements development.
- The *Scrum Master* oversees and enforces the process, and generally does whatever is needed to keep the Team productive and running smoothly.
- The *Team* members develop and validate deliverables.

Artifacts

- Stories: Specifications for small deliverables to be developed and validated
- Epics: Specifications for larger deliverables that must be decomposed into Stories prior to implementation

Ceremonies (recurring meetings with standard agendas)

- Backlog Refinement: All review, provide feedback on, and revise draft Stories and Epics, to ensure these items, and Team members, are ready for use in Sprint Planning
- Sprint Planning: Select rough scope (Sprint Backlog) based on priorities and estimates for Stories and estimate for how much work the Team can do in a Sprint. Decompose work into Task Breakdown for each Story and refine scope based on estimated Task hours.
- Daily Stand-Up: A brief daily meeting in which Team members update each other on status of work and plans for the near future, and identify who will follow up on issues that are slowing work
- Sprint Review: End-of-Sprint meeting at which Team members demonstrate all deliverables completed in the Sprint
- Sprint Retrospective: End-of-Sprint meeting at which all participants identify what practices to continue, identify what needs to be improved, and decide which improvements to make, and who will make them

Tracking

- Scrum Taskboard: A tabular display of Sprint status information, with (commonly) four columns. The first column contains the planned Stories for the Sprint, in rank order from top to bottom. The next three columns show status of Tasks, such as Not Started, In Progress, and Complete. Team members move each Task through those states on starting or finishing the work of that Task. This Taskboard may be a physical one, with sticky notes (say) on a wall, or a virtual one, in Web-based Agile Project-Management tool.
- Burndown Chart: A chart of hours remaining in the Sprint for uncompleted Tasks, versus day in the Sprint. The plan line shows the desired trend, and the charted data points show progress that can be compared to the plan.

I was putting the finishing touches on the paper when we received a call from someone at Thermo Fisher regarding Agile hardware development.

2 The Company

Thermo Fisher Scientific, Inc., is a major biotechnology company that makes a wide variety of analytical and laboratory products, such as mass spectrometers, chromatography equipment, and so forth. The company has offices all over the world.

In early 2015, Thermo Fisher's office in Bremen, Germany, began applying some Scrum concepts in the context of hardware development. The basics included the concept of a dedicated Scrum team (i.e., not assigning one person to multiple projects), daily stand-up meetings, and some of the other Scrum meetings. They observed good morale, and faster prototype development than expected.

Encouraged by stories of success in Bremen, management in the in San Jose, California, office wanted to know if similar improvements might be possible in their part of the world. Marisa Richardson, who heads up the PMO for the Life Sciences Mass Spectrometry division, began looking for Agile consultants with hardware expertise, and quickly found cPrime. Upon hearing about my research into Agile techniques for hardware development, Marisa asked to meet with me.

I introduced myself to Marisa in the San Jose Office as "a physicist, working in the area of Agile process consulting." (Since Thermo Fisher employs numerous physicists, chemists, and engineers of all kinds, I

felt very much at home there.) It was a short and productive meeting, which led to other meetings, which led to planning their first Agile hardware project.

On July 10, I met two of the key decision makers in the Life Sciences Mass Spectrometry division, Jean-Jacques Dunyach (Director, R&D) and August Specht (Director of Global Research and Development), in San Jose, to discuss the concept of Agile hardware in depth. (This was probably the first meeting in my time at cPrime that was populated mostly by people with advanced degrees in physics and chemistry!)

I had the impression that this meeting would lead to the go/no-go decision.

I was candid in our discussions.

“Do you have experience in Agile hardware?” August asked me.

“No,” I said. “I don’t, and I don’t know anyone who does. I’ve done research in the area, and I believe the approach will work, but I don’t have that experience yet.

“You will be pioneers,” I added. “We have a lot of experience to draw upon for Agile software development, but Agile hardware development is new. If you want to proceed, everyone needs to understand that there may be challenges.

“I recommend picking a medium-size project which is important, but not a life-or-death matter for the company. I also recommend picking people who are willing to be pioneers, and have some tolerance for risk and mistakes.”

I learned that Thermo Fisher was willing to be a pioneer in the process of developing hardware, as well as in the hardware itself, when they made the call to go forwards with the Agile initiative.

3 The Plan

Our July 17th planning session was focused and productive. By the end, we had a plan for the engagement.

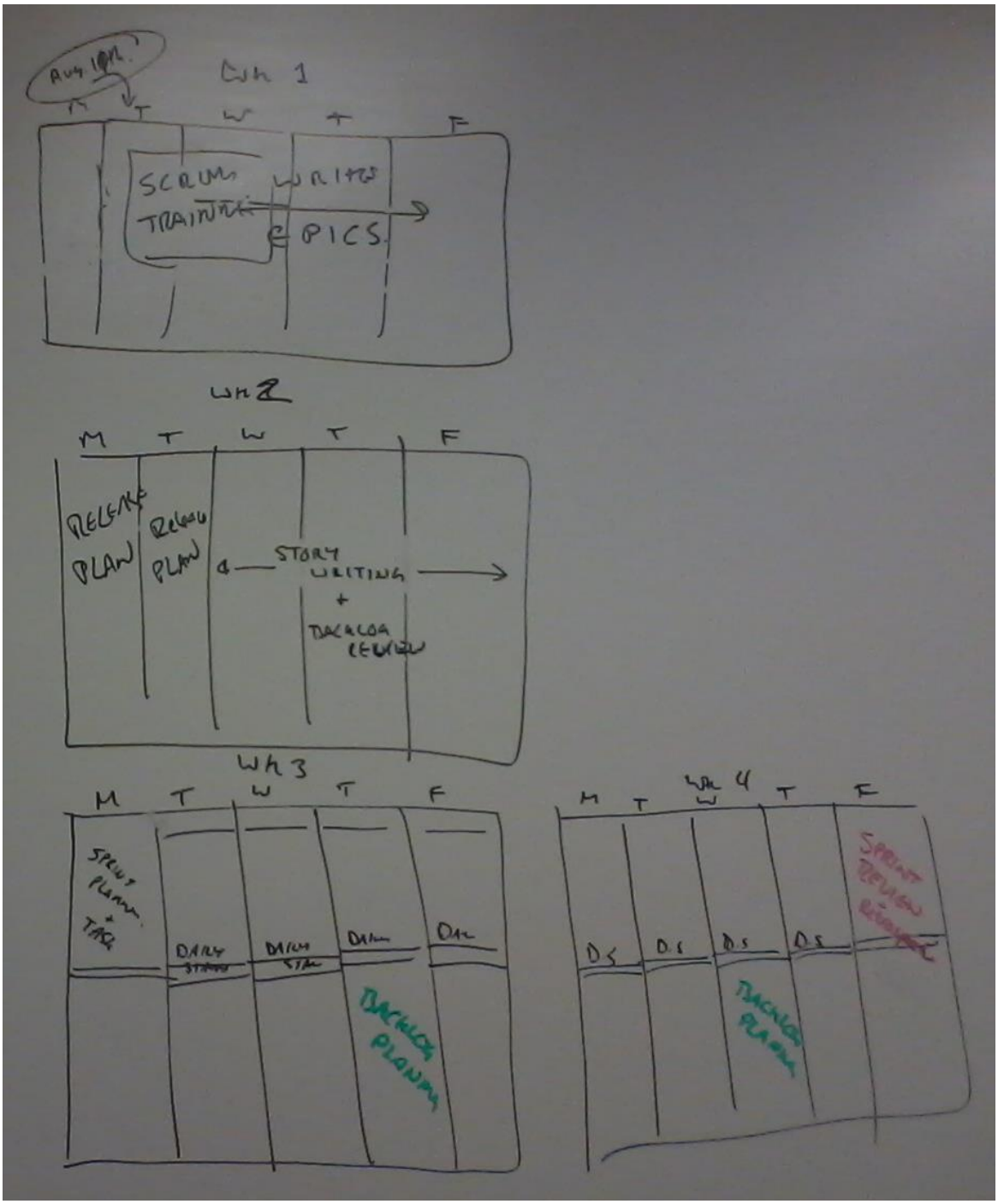

Figure 1 The Plan

Our schedule would be as follows:

Week 1:

Tuesday and Wednesday: Scrum Training

Thursday and Friday: Write high-level requirements (Epics)

Week 2:

Monday and Tuesday: Draft Release Plan, including estimates for all Epics
Wednesday through Friday: Write User and Technical Stories to prepare for Sprint 1

Week 3 (first week of Sprint 1):
Monday: Sprint Planning
Tuesday through Friday: Develop and test deliverables

Week 4 (second week of Sprint 1):
Develop and test deliverables

We also had a product. “Trailblazer” (not the real name) was to be a new product, and just the kind I’d suggested: Important, and of medium size.

The Trailblazer product was to be developed with a new team of nine people who were experienced in developing analytical equipment, but who had not worked together in this fashion before. The managers assured me that these were the right people to innovate, and history proved them correct.

4 The Scrum Training

I did not find out until sometime afterward, but the Team members who showed up for Scrum training on August 11 had not learned they were about to become pioneers until shortly before the class started. For them, the good news about the project approval was accompanied by the startling news that the Team would be developing the product with a Scrum process.

The Product Owner was to be Dr. Michael Bedford, Senior Scientist, acting in a role more commonly described as “Technical Lead” at Thermo Fisher. He was going to wear two hats: Half-time Product Owner, and half-time Team member. As Product Owner, he would drive the product definition over time. As Team member, he would also be responsible for some of the hands-on work of developing the product.

The Scrum Master was to be Ray Chen, an experienced Program Manager at Thermo Fisher. Ray had extensive background in managing large hardware-development projects at the company, and knew the ins and outs of the company’s overall governance model thoroughly.

Most of the Team members had barely heard of Scrum, and none of them had experience with it. The news, therefore, was quite a surprise. So I must give everyone in the class a lot of credit for their open-mindedness. Several people were skeptical that Scrum was a good idea, but everyone was willing to learn. Some started to think that Scrum just might have something going for it before the class ended, and others came to think this way over time.

The class itself went smoothly, with lots of good discussion. I changed exactly one slide in our standard Scrum training, to show an example of how to write hardware requirements. Everything else in the slide deck remained unchanged.

4.1 Lesson #1: Product Owner as Team Member

One of the conclusions of the Agile hardware paper was that the people who would most commonly lead the definition of a product, as Product Owner, are likely to participate as Team members as well. This was indeed the case at Thermo Fisher.

There is no policy in Scrum regarding whether one person can act in these two Roles. In software development, the question usually does not arise. For the most part, the people who become Product Owners do not have a hands-on software development background, or interest in actually writing software. Instead, they tend to be more focused on market and business needs, and often have formal titles such as Product Manager or Business Analyst.

In hardware development, we predicted (and I was now observing) a different pattern. The people who drive the definition of hardware products often do come from a hardware-development background, and will participate in development.

There is nothing wrong with either model, but there are trade-offs to consider. Someone who functions solely as a Product Owner can normally do this for two or three Scrum Teams concurrently. Someone who functions as both a Product Owner and a Team member will most likely be able to function as Product Owner for only one Scrum Team.

5 Preparation for Release Planning

Thermo Fisher's management quickly validated one of the findings of the Agile Hardware research, namely that *Release Planning* was important for hardware projects.

The fundamental development cycle of Scrum is the Sprint. Scrum Teams plan what to do in a Sprint in a Sprint Planning Meeting, at or shortly before the beginning of the Sprint. Sprint lengths are usually 1—3 weeks, with 2 weeks being the most common. Thermo Fisher selected two weeks for the standard Sprint length, although scheduling issues resulted in the first Sprint being about 2 ½ weeks long.

The next larger time horizon for planning is the Release cycle. It is a time scale intermediate between a Sprint (two weeks) and a Roadmap (6—24 months). In software development, the Release Plan produced by this kind of planning exercise often corresponds to a production release of the product to customers. However, "often" is far from "always," and it is fairly common to have the production release at longer or shorter intervals than the span of the Release Plan.

The need for Release Planning in Software varies. Some organizations need it, and some do not. The key drivers for Release Planning are the importance of planning cross-Team and external dependencies over time, and to set expectations (and make trade-off decisions) about features developed in the interval of interest.

Hardware development has a higher cost of change, and a greater time-to-usability, than software, which implies that Release Planning is more of a requirement than an option.

The biggest challenge with Release Planning is to write the product requirements with enough coverage, and in enough detail, to enable meaningful planning. This kind of requirements development is most commonly performed over a period of a few months prior to a planned Release Planning session, but that kind of time was not available at Thermo Fisher. Instead, we had a few days.

5.1 Organizing the Scope

The first challenge was to develop any understanding of product scope at all. While the product concept was well-understood by the Team, the scope definition was unclear in any practical sense.

I knew how to proceed for a new software product. I would begin by asking about the major user-facing areas of functionality that were important for the first (or next) product release, and guide the Team to identify these areas well enough that we could being to write specifications for them.

User-facing capabilities, and any non-functional requirements already known to a Team, both imply some conception of infrastructural capabilities that must also be developed. I would guide the Team to identify these aspects of the product as well.

With the basic developmental areas defined, I would guide the Team to write Epics (high-level scope descriptions) to cover all aspects of the product as currently understood. A key part of this effort would be to identify how to divide up the work of writing the Epics across the Product Owner and Team members.

However, this was not a software product. Not being an expert in hardware or analytical equipment, I had no idea what the relevant pieces were, or how to organize their definition. So I asked a simple question:

"How do you think about the major pieces of this device? How do you organize your understanding of the design of this thing that you want to build?"

Then I sat back, watched, and listened.

I noticed that the discussion turned immediately to the physical components of the product, with almost no focus on user experience. This was a different behavior than I typically observe for software groups, which focus more on what the product *does* than what the pieces of the product *are*. Yet this group assumed a common understanding of what the product did, and focused almost exclusively on what the components were.

Was this a good way to proceed, or not? It would not be a good way to proceed for a software product, but I didn't know if this behavior here was appropriate, pathological, or somewhere in between.

I decided to experiment by bringing up the user experience aspects periodically, and seeing what the group did. They were more than happy to talk about these aspects, yet the discussions seemed curiously unproductive in terms of bringing clarity to developing the specifications. Also, the discussion always migrated back to the physical components of the product.

I concluded that the people in the room knew what they needed to focus on, and went with the flow. This turned out to be the right decision, even if it did seem counter-intuitive to me. The discussion led to a clear definition of the categories of scope needed to develop the product.

User Experience	Hardware	Software	Electronics
Device Operation	Cabling	User Interface	Power supply
	Electronics box	Command-line interface	Temperature control

Firmware

Figure 2 Scope Categorization

Figure 2 shows the organization that flowed from the discussion, with some of the specific scope areas listed (confidential product information has been omitted).

At this point, we knew enough to begin writing the specifications for the product.

5.2 Product Definition and Product Lifecycle

Thermo Fisher's concept of a product lifecycle is a very common one. Product designs go through three phases, which I'll call Engineering Prototype, Manufacturing Prototype, and Final Design.

- Engineering Prototype: A device that implements the desired attributes of the product, including the desired behavior, for the first time
- Manufacturing Prototype: A refinement of the Engineering Prototype, intended to be manufacturable
- Final Design: The design that will be manufactured

This three-phase pattern has been a successful one, but the relationship between this pattern, and the lifecycle as seen from a Scrum perspective, remained to be determined.

It was tempting to collapse all three phases into one Release cycle, by declaring that we would build something that could be manufactured. Tempting, but arrogant.

In software, we can work this way, but in software, we can refine and iterate in very small increments all the way up to the production release of the product.

This pattern does not hold in hardware development. Too much of what needs to be refined, in the design, is discovered once many or all of the components are assembled, and this assembly cannot be performed until the components exist in a sufficient state of completion. As a practical matter, this means that some kind of long-scale iteration of a workable design is required. One might argue that the iteration need not map exactly to the three phases listed above, but that is not the same as saying that the above phases are inherently wrong. They are not.

We took the conservative approach of hewing to Thermo Fisher's standard phases. The Team felt very comfortable with this approach in our new context, and this approach fit neatly within the expectations and overall governance model of the company.

With this clarification, we now understood what the term "Release Cycle" meant for this product. The Release Cycles would map exactly to the three main design iterations. Our first Release Cycle, therefore, was to complete the design of a working Engineering Prototype, and the Team wrote the Epics accordingly.

We discussed how to allocate the different topics to different people, and they got started. Over the next few days, the Team members wrote as many Epics as they could, trying to get full coverage of the product definition. We held daily review revision sessions, and I gave advice on how to improve the Epics until they had the level of clarity that everyone considered appropriate.

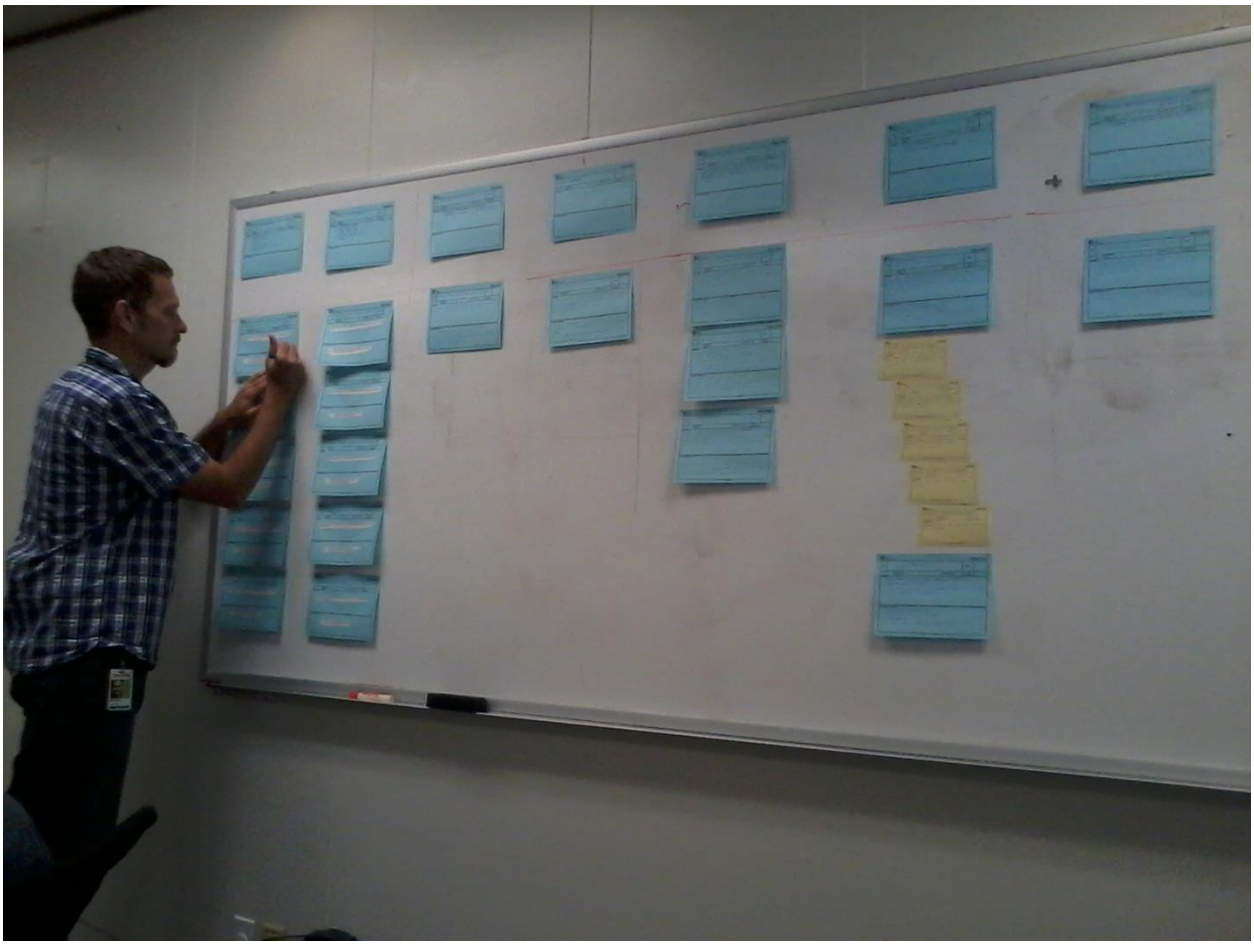

Figure 3 Product Owner, Dr. Michael Belford, Editing Stories

5.3 Lesson #2: Software Accretes Functionality. Hardware Accretes Components.

Software products evolve over time by accreting small increments of usable functionality. Hardware products cannot evolve in this fashion, because we must build each component in a way that enables the desired range of user experiences.

Thus the intended device will 'accrete' components for some time before any product capability exists. It is still possible to do some iteration of functionality via prototyping, but the cycles will be longer, and increments of functionality larger, than for software products.

5.4 Lesson #3: User-Experience Information may not Lead to many User Stories

In Software, we think of implementing a user experience by writing code at many layers in the product: User-interface code, application logic, persistence layer, database, and so forth. Roughly speaking, each user experience can be considered atomically, and adding many of them adds code to all layers of the product. Thus "User Stories," which provide specifications for these user-facing deliverables, are commonly regarded as the dominant type of specification.

This pattern does not apply to hardware development. Instead, the user requirements largely serve to identify the capabilities that must be supplied by hardware components. It is taken as a given that

“wiring the front panel” to the rest of the product can be done, and will work, and this is largely true. The ‘front panel’ itself is another piece of hardware, and the only piece directly driven by the desired user-facing behaviors of the product. User Stories are thus directly relevant for the front panel, but not for most other aspects of the product.

The bulk of the specifications for deliverables are about the internal components of the product, which provide no user experience. These may be represented as "Technical Stories," which describe the deliverable to be produced, but without reference to a person's direct experience of product functionality. Thus the bulk of the Stories written for hardware development are likely to be Technical Stories, not User Stories, in direct opposition to norms in the software world.

For example, the design of a printed circuit board in the Trailblazer product was the aggregate outcome of implementing many Technical Stories, with whose deliverables no human user would ever interact directly.

The picture blurs when the device is controlled by software, in which case the usual software patterns apply to the software portion of the product, but only to that portion.

5.5 Requirements Artifacts: Stories and Epics

In Scrum language, a “Story” is a specification for a deliverable. The common term of *User Story* refers to a deliverable that is a behavior in the product, i.e. which reflects a user interaction of some kind. The somewhat less common term of *Technical Story* refers to a deliverable that is not perceived, or used directly by, a user of the product.

By definition, a Story is implemented in a single testable (and tested) chunk, and must represent a deliverable that is small enough that a Scrum Team can implement many (say, five to fifteen) in one Sprint of two weeks.

Any description of scope that exceeds the size limit of a Story is called an Epic. Epics useful for longer-term planning, and are always decomposed into Stories prior to Sprint planning and implementation. (The Epic-Story relationship is essentially that of the *Work Breakdown Structure* of classical Project Management.)

Epics and Stories have the same written format. A very common format for User Stories contains a Title, a Narrative, and Acceptance Criteria.

- The Title is a one-sentence summary of the deliverable.
- The Narrative contains one to a few paragraphs that describe the user experience.
    - The first sentence is of the form, “As a <Role>, I want to <perform an action>, so that <this benefit occurs>.” Any following text expands on the user experience.
    - The Narrative section often contains non-narrative information as well, such as links to user interface designs, and other external documents of interest that relate to the user experience.
- Acceptance Criteria summarizes things that must be true, and validated, about the completed deliverable.
    - Many Acceptance Criteria provide a starting point for test-case definition during development work.

- This is also a good place to provide other useful requirements information that does not fit in a narrative description.

Figure 4 shows a sample User Story that reflects the above description.

Title	Buyer views statistics for transactions with Vendors				**Rank**	3
ID	22		**Estimate**	8	**Total Task Est.**	13
Narrative						
As a Buyer, I can view my statistics about my transactions with Vendors, so that I can understand how my history looks to Vendors. When I click on a buyer-statistics link, I see my statistics. (This link is on the home page, under account information.) Prototype / UI References: Landing Page: Attached (landingPage.html) External Documents: Company Style Guide for UI: http://wiki/UI/UIStyleGuide.doc Statistics to be computed: http://wiki/apps/buyerstats.doc						
Acceptance Criteria						
• When the user clicks on the link, the application should display the statistics. • User can create fictitious buyers and suppliers for use in testing. • When the user submits or responds to RFPs, report shows updated statistics that reflect the user's activities. • The screen should appear within one second, under a load of 20 concurrent requests pulling data from 20 different static-content files.						

Figure 4 Sample User Story

The attempt to write requirements proved harder than I anticipated. This was the first lesson I learned about employing Scrum in a hardware context.

5.6 Lesson #4: Writing Hardware Requirements is Difficult to Learn

The problem is not that writing requirements for hardware is inherently more difficult than writing requirements for software. The "Story" format used in Scrum can be the same in either case. What makes the experience of writing requirements difficult is the force of habit.

A Story, in Scrum terminology, is a specification for a deliverable that the Scrum Team will develop. In software development, Stories are mostly about features in the software product. In hardware development, Stories are mostly about the deliverables that comprise the design of the product (which, ultimately, goes to manufacturing for production).

In my experience, Software teams find Story writing fairly easy to do. Like any skill, this one takes time to master, but Software team members usually grasp the basics quickly. However, the Hardware team found Story writing substantially harder to learn than is typical for Software teams.

The reason for the challenge has to do with the different ways Software and Hardware developers perceive requirements. At the risk of overgeneralizing, I've noticed the following:

- Software developers perceive requirements as primarily reflecting product behavior (functional requirements), and with relatively little in the way of nonfunctional requirements. The concept of a Story as a specification for a deliverable is therefore fairly intuitive.

- Hardware people perceive requirements as one big ball of "things that must be true when the product design is finished," with less awareness of the distinction between functional and nonfunctional requirements, and a much larger set of 'things to build' that have no visible behavior from the user's perspective. The undifferentiated mix of 'requirements' means that the first attempts to write Stories often had no concept of deliverables at all!

The result of this challenge was that the process of writing Epics for the Release Plan went more slowly, and required more revisions, than I expected. We discarded a number of Epics that described criteria that had to be met, but which did not reflect deliverables to be developed.

5.7 Technical Stories

Technical Stories have the same elements as User Stories, but differ in that the Narrative is simply a description of the deliverable, not a description of a user experience. Figure 5 shows a sample Technical Story for a hardware product.

Title	16-bit Digital-to-Analog Converter Motherboard Integration				**Rank**	9
ID	25		**Estimate**	8	**Total Task Est.**	60
Narrative						
Design into the motherboard a Digital-to-Analog converter (DAC), in order to provide the high-level analog voltage required to drive the analog display. Assume that the digital drive presented at the input of the DAC module is of sufficient amplitude to activate the unit properly, but not so large as to damage the input gates. Since no DAC has been chosen for this purpose yet, select an appropriate one for incorporation into the motherboard design.						
Acceptance Criteria						
• 16-bit output with +/- 0.5 bit nonlinearity • Single supply operation: 2.7 to 5.5V • The maximum current to drive the unit is 500mA. • The maximum clock rate is 3.4 MBit/sec. • The output will be a time-varying analog signal varying from 0V to 5V at steps of 7.5 µV, with a maximum output impedance of 1 Ohm. • Device meets the other requirements of the I2S serial bus						

Figure 5 Sample Technical Story

5.8 Lesson #5: The Product Owner Writes Few Stories

In Scrum, the Role of Product Owner drives the product definition. For Software products, the Product Owner commonly writes a large number of User Stories, while the Team members normally write a smaller number of Technical Stories for deliverables the Product Owner does not understand well enough to define.

These statements are true for hardware development as well, but the shift in Story types means that the Product Owner wrote a much smaller number of Stories than did the Team members. For the Trailblazer

product, the Product Owner did indeed drive the product definition, but his day-to-day work focused more on reviewing Technical Stories written by Team members than writing Stories for review by Team members.

5.9 Lesson #6: Stories are Tested by Implementers

The software world assumes that Quality Assurance specialists are responsible for making sure that software is tested adequately. The developers who write the software are generally seen (by themselves, and others) as not being the best people to develop test cases that cover each User Story's functionality well.

The situation in Hardware development is different. The fine-grained hardware deliverables generally do not have functionality that is hard to understand. At the same time, the nature of deliverables varies so widely that the concept of a general-purpose "Hardware tester" makes little sense. Thus the deliverable of a Story generally is tested by the person or people who developed it, simply because no practical alternative exists.

System-level testing of a completed hardware product is different. Test teams commonly do exist to put the completed product through its paces.

5.10 Lesson #7: Scope Decomposition is Component-Oriented, not Functionality-Oriented

The preponderance of Technical Stories over User Stories reflects the reality that the natural scope decomposition is into components, not areas of functionality. I expect this reality will shape larger, multi-Team organizations as well, with Teams focusing primarily on different subsystems and interfaces.

The Component-Team organization is not unknown in the software world, but is generally considered to be the least-desirable form of organization. The reasons for preferring other styles of Team definition (e.g., Feature Teams) is because other styles more closely reflect the user experience, and so leads more easily to usable increments of functionality over time, and because the other styles have less intense cross-Team dependencies.

In hardware development, the Component-Team style of organization appears unavoidable, and the cost of cross-Team dependencies must simply be accepted and managed as well as possible.

6 Release Planning

Our Release Planning session suffered from the limited time available for preparation. The four days allotted for the preparatory work and the planning session were not sufficient to produce a Release Plan in which the Team could have much confidence. Still, the effort of Release Planning would be useful from a training perspective, and the Release Plan to be produced would have some value as a rough forecast of achievable scope versus time.

Release Planning is simple in concept. A Scrum Team can accomplish a certain amount of work on its deliverables in any one Sprint. The Scrum term for this numeric limit is *Velocity*, or, more specifically, Sprint Velocity. It is typically the responsibility of the Scrum Master to provide an estimate for this value, for use in planning work for a Sprint, or for a Release cycle containing some number of Sprints.

Teams also estimate the Stories or Epics to be implemented, and Product Owners make the final decision on the ranking (sequencing) of these items.

When the above work is done, it is possible to develop a forecast of what can be accomplished in a given period of time, or of how much time is required to complete a certain amount of scope. This information is commonly incorporated into a Release Plan, which describes the planned scope of the selected Release Cycle's duration.

6.1 Estimation of Work

Estimating the size or work to be done for an Epic or Story requires two things: A unit of measure, and a technique for producing the estimate.

The most common units of estimation encountered in Scrum are Story Points and Person-Days, although other terms that mean approximately the same things are often encountered (indeed, it is not uncommon to find Teams that define the term "Story Point" to mean a Person-Day).

A Story Point is defined to be a dimensionless measure of size, which is not defined in terms of time in any direct fashion. Teams using Story Points will develop a sizing scale, normally based on the Fibonacci sequence, so that the possible sizes for Stories (or Epics) will be 1 Story Point, 2 Story Points, 3 Story Points, 5 Story Points, and so forth. They then take a selection of Stories of varying size, and identify a subset whose sizes they believe scale approximately as the numbers in the sequence. This subset becomes the set of reference Stories that define the meaning of the numeric sizes. Subsequent Stories are then estimated by comparing them to the reference Stories, e.g., by saying, "This Story has a size of five Story Points because it is about the same size as our reference Story of five Story Points."

A Person-Day is defined to be eight hours of a person's time. It is a measure of effort, or aggregate time devoted to working on something, not duration. If I spend four hours working on something today, and finish it tomorrow after working on it another four hours, then my total working time is eight hours, or one Person-Day. Similarly, if two people work on the same chore, and each spends four hours on it, then that pair has also expended one Person-Day of effort on the chore. While the duration of the two cases is different (two calendar days for the first case, and one for the second), the effort is the same: One Person-Day.

At Thermo Fisher, the Team members overwhelmingly preferred to estimate effort in Person-Days, rather than Story Points, and this turned out to be a good choice. In hindsight, I believe it is also likely to be the only viable choice for hardware development, for reasons that will be made clear below.

The two estimation techniques most often used in the Agile world are Planning Poker and Affinity Estimation, both of which pool the expertise of a group of people to develop the estimates. The former is an excellent approach for providing in-depth analysis and the most reliable numbers, while the latter is generally a better choice when one needs to estimate a large number of items quickly. Thus Planning Poker is commonly used in Sprint Planning, while Affinity Estimation is more often used in Release Planning, and we did indeed use it for this purpose at Thermo Fisher.

A key contributor to success in group-estimation efforts, but which is often unrecognized, is to select units that yield estimates as one-digit or low two-digit numbers. Most people have an intuitive feel for numbers in this range, but not for numbers that are much larger. Thus for estimating the large Epics the

Team wrote for Release Planning, we decided to estimate in units of Person-Weeks, where one Person-Week was defined to be five Person-Days, or 40 hours of one person's time.

We set up an estimation wall with column headings for size (on yellow sticky notes), as shown in Figure 6. The numbers in the column headings were drawn from the Fibonacci sequence.

Figure 6 Affinity Estimation for Release Planning

Each Team member took an Epic from the pile on the table, and taped it to the wall under one of the size headings. Team members continued in this fashion until all Epics were on the wall, and also discussed whether to move Epics to different columns. Within a few hours, the Team had a consensus belief that the estimates were as reliable as they were going to get, and we were done.

Ray, the Scrum Master, had estimated the Team's Sprint Velocity at 34.1 Person-Days for an average two-week Sprint. The Epics on the wall totaled 98 Person-Weeks, or 490 Person-Days. These numbers led to an estimated duration of 15 Sprints, or 30 weeks, which is about 7 months.

Normally, the next step would be to complete the Release Plan by working out the sequencing (ranking) of the Epics, and thus forecasting the planned scope over time. Due to time pressure, and the practical reality that the information wasn't sufficiently detailed to make the exercise very useful, we did not do that work at this time.

6.2 Estimation of Velocity

Estimation of Velocity is a fairly simple process.

If the units of estimation are Story Points, one forecasts the Velocity of the next Sprint based on the history of the Team. For example, if, in the last Sprint, the Team completed Stories whose estimates add to 35 Story Points, then one might forecast the Velocity of the next Sprint as 35 Story Points. It is often common to forecast the next Sprint's Velocity as the average of the last three (say) Sprints' Velocities, to smooth out variations

If the units of estimation are Person-Days, it is possible to use the same technique as for Story Points, but not common. It is more common to forecast the Velocity based on how much time the various Team members are likely to be able to spend on implementing and testing the Stories' deliverables in a Sprint, which was what we did at Thermo Fisher.

6.3 Lesson #8: Time-Based Estimation is Essential

Although the practice of relative sizing with Story Points is popular with many Agile coaches, this technique was not practical at Thermo Fisher, and I doubt that it will work in general for hardware development. Time-based units, such as Person-Days, did work well at Thermo Fisher, and I expect will generally be required in the hardware world. (I remember one person expressing considerable relief that I hadn't tried to force Story Points on them.)

Relative sizing with Story Points requires that the Team be able to develop a consensus norm about how big things are, without explicit reference to time. The expression of this consensus norm is often captured in a set of reference Stories, each of which defines by example what a particular size means.

If the Team cannot develop a consensus norm of sizing that is not based on time, then the concept of relative sizing, and the use of Story Points as units, is not applicable.

Software-development Teams require a mix of skill sets, such as HTML, JavaScript, Java, test development t, and so forth. Most members of a software-development Team can develop enough facility with skills outside their area of specialization to do some work in these other areas, and have reasonable insight into what such work entails.

Hardware-development Teams also require a mix of skills, such as electrical engineering, mechanical engineering, physics, chemistry, and so forth. Most members of a hardware-development Team *cannot* develop enough facility outside their areas of specialization to do work in other areas, or have reasonable insight into what such work entails. For example, a Team member who has a BS in Mechanical Engineering is not likely to have a useful understanding of printed-circuit board design work.

The high specialization of skills in hardware development means that the concept of a reference Story for a particular size becomes meaningless in the absence of explicit reference to time, because no one Story can serve as a sizing reference point for the whole Team. Time-based estimates are therefore required.

The logic behind this statement may seem abstract and disconnected from the experience of many readers, but the reality becomes evident in the field, and quickly.

7 Sprint Planning

Sprint planning went more smoothly and reliably than Release Planning. This was partly because we had sufficient time to prepare, and partly because the Release Planning experience provided useful practice for Sprint Planning as well.

The Product Owner provided guidance on Stories that needed to be written for the first Sprint. Team members, as before, wrote most of the (Technical) Stories for the simple reason that they were the only people who could write them. A beneficial consequence of this distribution of labor was that no one person was overloaded with work.

We held a Backlog Refinement meeting each day to review and discuss each new or revised Story, just as we had done for Epics when preparing for Release Planning. The Product Owner facilitated the meeting, and everyone provided feedback on the draft Stories. The various authors would then incorporate the feedback from the rest of the members either by making quick edits in the meeting, or by revising their draft Stories afterwards.

These meetings were painfully slow at first, but the pace picked up as everyone became more skillful at drafting better Stories. As usual, as expected, and as desired, the practice of Backlog Refinement not only produced well-written specifications for Team deliverables, but also provided dramatically improved understanding by Team members as to just what it was that they would be developing.

The process of Sprint Planning requires the following be done:

- The Sprint Velocity must be estimated
- All Stories under consideration for the Sprint must be estimated
- Initial Scoping is done by defining the order of implementation of the intended Stories (ranking) and selecting the ranked set that fills the Sprint up to the Velocity ceiling
- Each intended Story has its work decomposed into a Task Breakdown, with each Task estimated in hours (Person-Hours, to be precise)
- Final Scoping is done by revisiting the scope decision based on the Task estimates and any relevant constraints or issues

Story estimation is most commonly done in Backlog Refinement or Sprint Planning meetings, with the decision made by the Team based on their specific needs and preferences. Thermo Fisher chose to perform these estimates in the Sprint Planning meeting.

We conducted the first half of the Sprint Planning meeting in the morning of Friday, August 21st. By 2:30 PM, the Team had estimated all 20 Stories under consideration.

Ray had already estimated the Sprint Velocity, so it was easy to develop a set of ranked Stories that would fit neatly into that figure. However, the high degree of specialization involved meant that several of these could be done by only one person, and he could not do all of them in one Sprint. So we had to revise the candidate Sprint Backlog so that it appeared achievable with the mix of skills and people that were available.

After a break, they began developing the Task Breakdowns at 3:15 PM. Team members worked singly or in pairs, as appropriate for the distribution of expertise among Team members, and the content of the

Stories. The work was mostly completed by the end of the day, and we finalized the scope Monday morning of the following week.

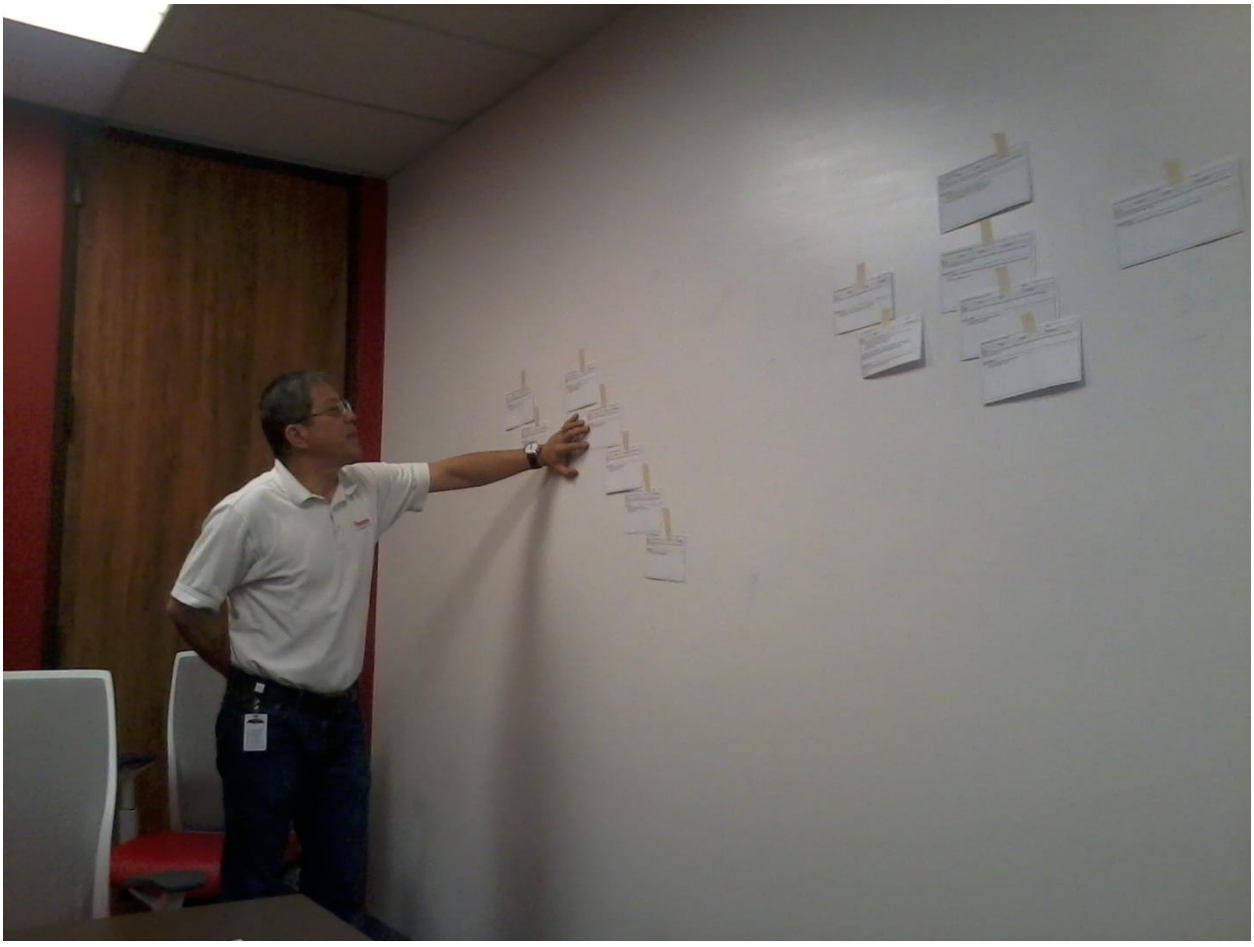

Figure 7 Scrum Master Ray Chen, Examining Task Breakdowns

The Task-hour total for the final Sprint Backlog was 237.5, while Ray's forecast indicated that the Team had a capacity of 310.5 hours. On discovering that some capacity remained, two of the members realized that they could allocate some time to studying programming techniques for embedded control software, so we added those Stories and Tasks to the Sprint Backlog.

Sprint Planning finished, the Team members selected the initial Stories to be developed, and got to work.

7.1 Lesson #9: Swarming is Always Good, but Ability to Swarm is Decreased

"Swarming" is a style of working in which Team members self-allocate as many of themselves as possible to each Story at a time, to minimize the duration of the Story's completion. This strategy minimizes the risk of completing less than the maximum possible value in a Sprint (where "value" is defined as the maximum number of highest-ranked Stories' deliverables).

The ability to put two or three people on a Story depends on certain factors:

- The scope of the Story must be large enough for the work to be decomposed into pieces for multiple people to work on it
- The skill sets of Team members and types of work required for the Story must be such that two or more people can work on the pieces at the same time

Thus while Swarming is always possible, the desired Swarm size of two or more is achievable only if the above characteristics are met.

Those characteristics commonly are met in Software development, but very often are not met in Hardware development, as noted in Section 6.3. Thus the Swarm size often will be one.

7.2 Lesson #10: Ranking is Important, but not Literally Achievable

Ranking of Stories by value (and dependencies) is necessary in order to plan Sprints so as to maximize the value that is produced in the Sprint.

In Software development, to a large extent, the Teams literally work on the Stories in the Sprint Backlog in rank order, starting with Story #1, and working down through the list over time. This approach yields the maximum achievable value in those too-frequent cases where unplanned disruptions result in less work being achieved than planned, as the top-ranked Stories are likely to be completed.

The greater specialization of skills in hardware development means that many Stories are likely implementable by only one person on the Team. Thus more Stories are likely to be in process at any one moment in Hardware development than in Software development, and unforeseen issues or disruptions are more likely to cause some Stories that have been started to not complete in the Sprint. This is obviously an undesirable situation, but it is a real issue.

7.3 Lesson #11: Velocity Matters, but is not Definitive

Velocity provides an extremely useful way to bound the scope of a Sprint. It is a great tool for this purpose, and accomplishes this purpose for both Software and Hardware development.

However, Velocity is less definitive as a planning tool in Hardware development. The greater ability to Swarm that we find in Software development means that Sprint Planning is often no more involved than putting Stories in the desired order, and ascertaining where in the ranked list we hit the Velocity ceiling.

In Hardware development, where Stories are often one-person exercises, we must pay much closer attention to how loaded each person on the Team is. It is easy to select a Sprint Backlog that satisfies the Velocity limit, but which overloads some people and leaves others idle.

Thus we will continue to use Velocity as a key planning tool, but must also pay careful attention to how work is allocated across people, in order to formulate a plan that can actually be executed.

8 Sprint 1 Execution

Execution of the Sprint was eerily smooth. I expected novel problems to arise, but that didn't happen. Instead, Team members updated the physical Scrum Taskboard daily, moving Tasks for their Stories from Not Started, to In Progress, to Complete, and picked up new Tasks on finishing old ones. Collaboration was excellent throughout, and highly visible in the Daily Stand-Up meetings. Team members made suggestions and offered to help when issues arose, exactly as desired.

Figure 8 Scrum Taskboard. Stories are blue notes, Tasks are yellow.

Ray updated the Burndown chart the first thing in the morning, each day. Progress throughout the Sprint was surprisingly smooth, with the remaining Task hours tracking the plan closely.

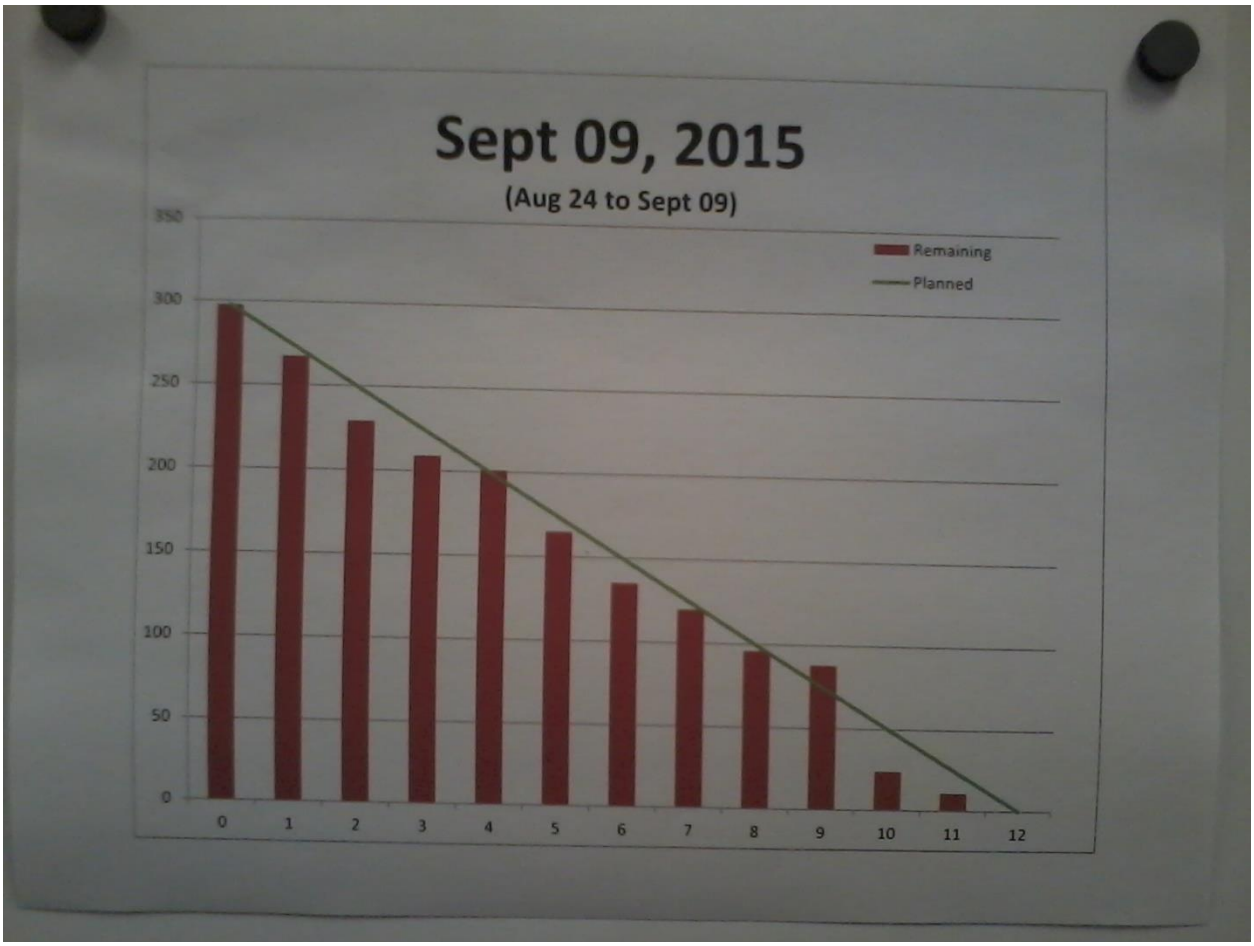

Figure 9 Burndown Chart at End of First Sprint

The only real challenge encountered in the first Sprint was in writing and refining Stories at a pace sufficient to prepare for the next Sprint Planning meeting. At about the half-way mark, we realized that there would be a shortfall, and so devoted more time to writing Stories, and more time to the refinement meetings. This effort paid off in providing well-groomed Stories (and well-informed Team members) prior to the planning meeting for Sprint 2.

8.1 Daily Stand-Up Meeting

Daily Stand-up meetings were concise and productive. Occasionally, people brought in things they were working on.

Figure 10 Some People Knit. Ann Winds Coils.

It was great fun, for me, to see such physical deliverables as coils and circuit boards.

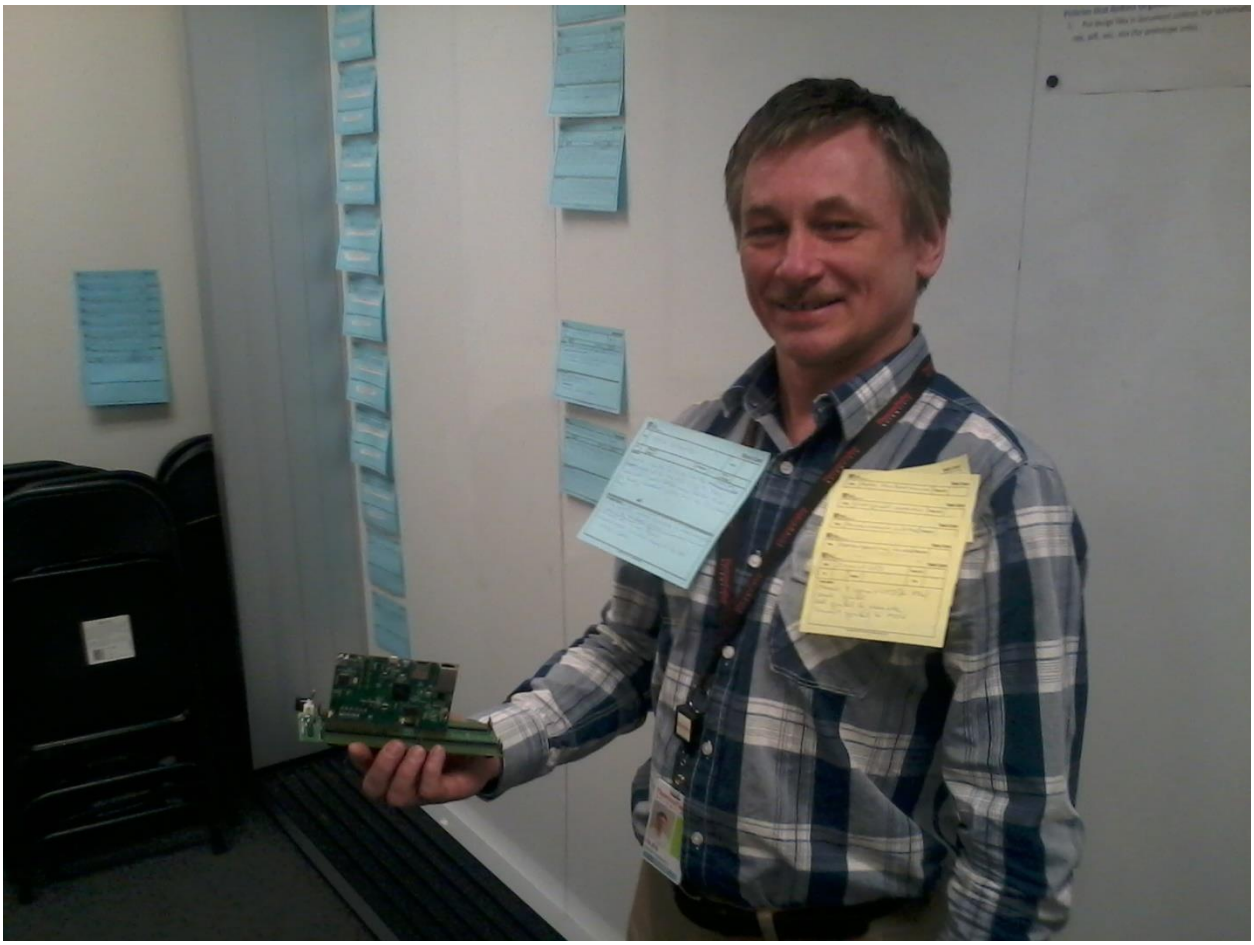

Figure 11 Scrum Master Misses Taskboard, Updates Oleg with Status Information Instead

8.2 Sprint Review

In a Sprint Review meeting, the Team members demonstrate the deliverables from all Stories (and bug fixes) completed in the Sprint. Part of the purpose is to ensure that everything that we say is done actually is done, part is to give the Product Owner a final opportunity to decide if a deliverable is what it should be, and part is to give people a fun opportunity to show off what they've built.

The major deliverables for Software development are usable capabilities in the software. The major deliverables in Hardware development are design artifacts which, in the aggregate, provide the information required to manufacture the product. Thus the Sprint Review mostly consisted of showing designs for parts of the product. Aside from this shift in focus, the Review meeting was quite normal.

Of particular interest to me was how much the Team members were enjoying the meeting. They were very pleased to show off their work to the only audience that was capable of appreciating it, and the meeting was definitely upbeat and fun.

8.3 Sprint Retrospective

The Retrospective meeting was similarly smooth and ordinary. Team members wrote, on sticky notes, short descriptions of "Things that went well in the Sprint, that we should continue doing," and "Things that didn't go well, that we would like to be better in the future." They wrote one item per sticky note, and stuck the notes on a white board under two columns with the above headings.

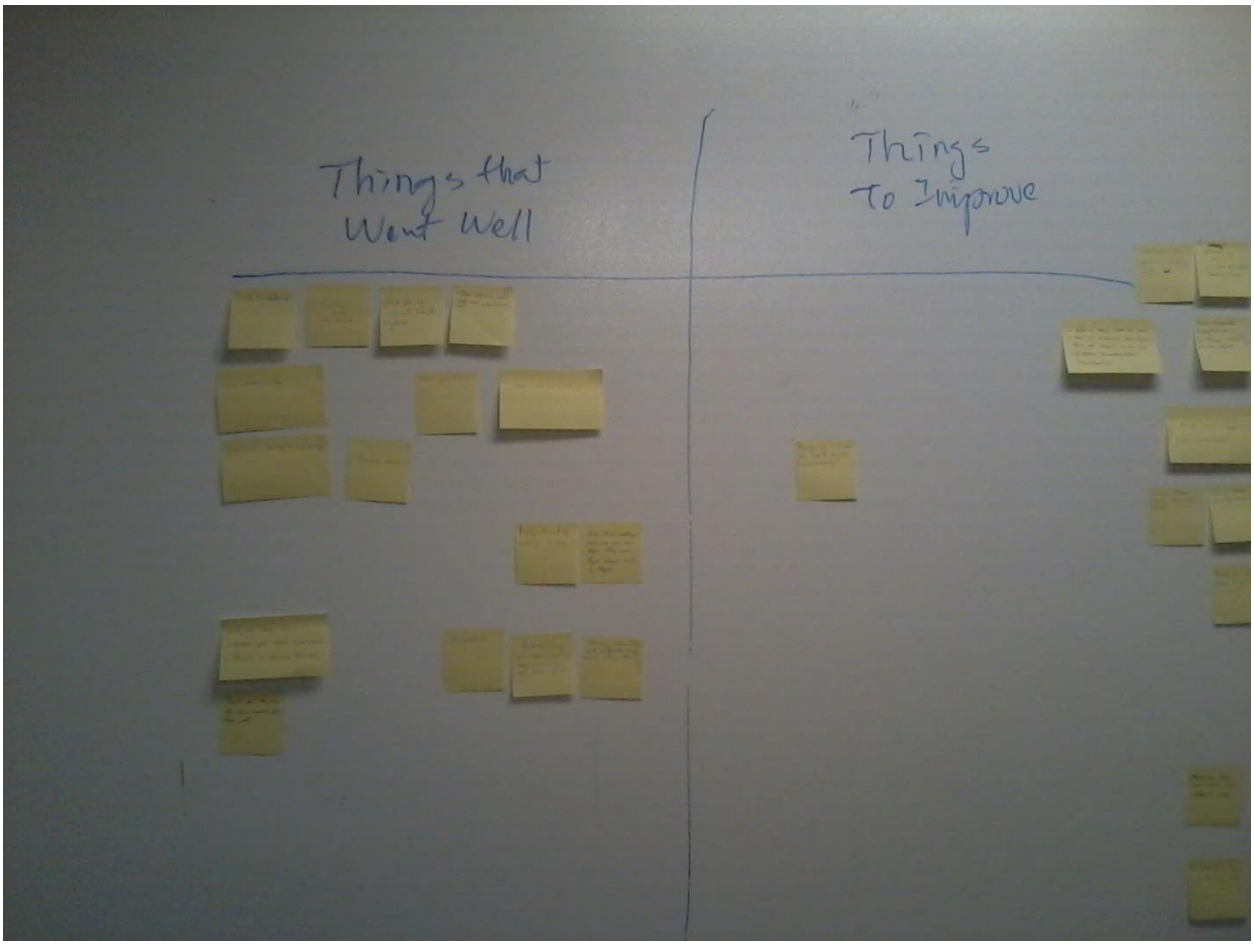

Figure 12 Retrospective Notes

We read through the "went well" items, and took a little time to enjoy the successes. Then we consolidated the "want to be better" items into clusters of duplicates, and talked through the issues identified. All of these issues were addressed in the meeting by making policy decisions, or simply reflected the newness of the Team to Scrum.

9 Conclusion

We have to stop this tale at some point, so I'll close the narrative by saying that the second and third Sprints built on the lessons learned over time, and the Team continued to improve. I considered the engagement to be a success, and was very pleased with how well it went.

The big question, of course, is what Thermo Fisher thinks of this new Scrum process for Hardware development. For that question, I think it is only fitting to leave the last word to the Product Owner, Dr. Bedford, who provided the following in an email to me on January 4, 2016.

> *"First and foremost, running this project with Scrum is an ongoing success. The team has formed nicely and we have proper buy-in from each team member. I make technical decisions with more input from the team than any other project that I have worked on. Refinement is a hassle because it takes so much time and requires us to stop doing and think, but the drawbacks I*

mention are also the clear benefits. We take time to plan. And we plan together. The daily standup is a life saver. As is the board. I can quickly understand where we are and what needs to be done. Release planning has been beneficial because it really focused the team onto what our big goals are as opposed to smaller engineering goals. I feel that management here in San Jose is excited about Scrum. I have had many of the bosses and decision makers stop by and ask about the process. I am curious to see if another team pops up in the near future."

[1] Kevin Thompson. *Agile Processes for Hardware Development.* 2015.