

Agile Requirements, Estimation and Planning -- Iteration Zero --

James W Grenning

james@wingman-sw.com

Presented at the Embedded Systems Conference

San Jose, CA 2012, Boston, MA 2013

Introduction	2
What is Iteration Zero	2
Preparation for Iteration Zero	3
The Bright Idea	4
Critical Dates	4
Product Needs	4
Technology and Architecture Goals	4
Who Participates in Iteration Zero?	4
Outputs From Iteration Zero	5
Product Vision	5
Architectural Vision	5
Product Team	5
Product Stakeholders	6
Product Story Backlog	6
<i>MuSCoW Analysis</i>	8
<i>INVEST in Stories</i>	8
Agile Planning and Tracking	11
Release Plan	12
Velocity 13	
Burn-down	14
Early warning means you have options	14
<i>Add people</i>	14
<i>Change the date</i>	14
<i>Adjust the scope</i>	15
Estimation	15
Planning Poker	17
Planning Poker Party	18
<i>High-low Showdown</i>	18
<i>Deal and Slide</i>	18
<i>Planning Poker for Groups</i>	19
<i>Developer Guts</i>	19
<i>Customer Guts</i>	19
Replanning	19
Get Ready to Start Iterating	20
Iteration Planning Meeting	20
Iteration Race Track	21
Team Agreement on Working Practices and Learning Goals	21
Other Iteration Zero Activities	22
Summary	22
Bibliography	24

Introduction

Agile product development is designed to improve visibility and predictability of schedule performance, as well as overall product quality. A product team transitioning from a phased and plan driven product development model will find it challenging to change to the iterative and incremental development model of Agile. This paper shows readers how to get started with Agile, and prepare for their first development iteration. The techniques described in this paper are useful for getting ready to do your first iteration, hence the name Iteration Zero.

This paper mainly focuses on the planning aspects of Iteration Zero, but touches on other important activities to prepare the team for iteration one and beyond. We start by looking at the input needed by iteration zero and the outputs produced.

The paper describes Product Stories, and how they are used to manage project scope to plan and track the product development effort. You will also learn what makes a good Product Story. We explore a series of techniques a team can use to estimate the backlog of stories. With estimated stories a plan is formed to incrementally develop the product and track their progress, giving the team the information to manage to a successful completion.

Because iteration zero prepares the team for iteration one, I'll briefly describe the mechanics, planning and tracking of iteration one and beyond. A big part of Agile development is the self-organizing team. So we briefly look at how a team has to make an agreement about work practices in iteration zero and must continually refine their work practice each iteration thereafter.

Although this is written from the perspective of a new development effort, this is applicable to any team transitioning to an incremental and iterative agile planning model.

What is Iteration Zero

Iteration Zero is a focused set of activities that a team does to get ready to begin a series of product development iterations. Each product development iteration delivers some demonstrable part of the system.

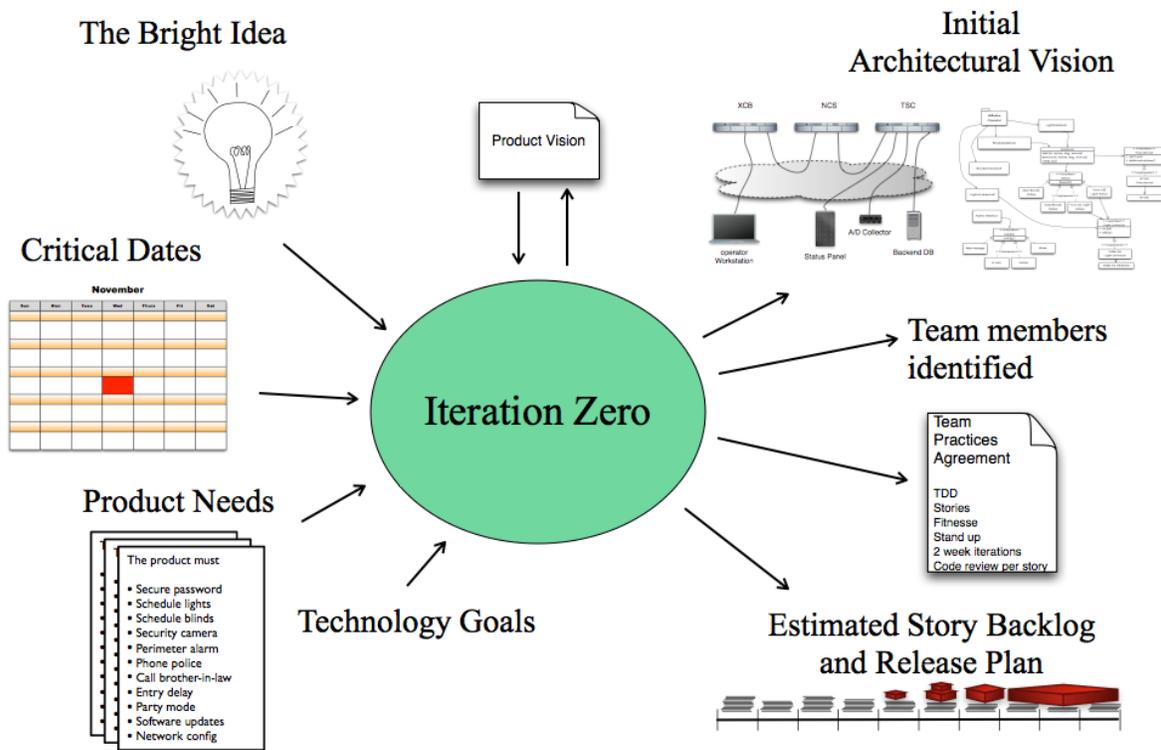
Needed documents are created incrementally, as we learn what is needed and how to build it. Each iteration should deliver some functionality. In an embedded development effort, when there is not hardware early in the development cycle, the early demonstrations might be executing test cases in a virtual platform environment, or on prototypes.

In making product development visible, agile relies on demonstrable progress rather than a document trail. This lets the team and business stakeholders see and adjust the evolving product.

In Iteration Zero the team explores the product ideas, customer needs, development practices, hardware and software architecture. The team breaks down their vision of the features needed and work to be done. They get a common understanding on the goals of the development effort, including the market need, the business needs, the product content and the effort needed to develop the product. They develop an initial plan.

The activity should be focussed. It should not drag on. For many development efforts, effecting teams of 10 or so people, a focussed Iteration Zero should take only a few days or a couple of weeks. Doing some of the intense activities might be best off-site away from day-to-day distractions.

Some organizations delay getting product development started because they don't know *everything* yet. Don't let crystal clarity keep your organization from starting. You can't know everything when you set off to develop a new product or product release, so it is a good idea to avoid spending too much time in the "fuzzy front end"¹. The time, once spent, cannot be recovered. It will be very difficult to catch up later in the development effort.



Preparation for Iteration Zero

To begin Iteration Zero the team and the business needs some key information. The team needs the key driving factors to create the vision that will guide development. Let's look at some of the typical inputs to Iteration Zero.

¹ [MCCONNELL]

The Bright Idea

To begin Iteration Zero you need a bright idea, a product to build, or a product to evolve. This is the initial vision of the product. Expect the vision to evolve over the development lifecycle as the development effort and opportunity become more clear,

Critical Dates

In any development effort it is normal to have target or mandated dates. Business runs on dates. Collect those critical dates and bring them to Iteration Zero. Also, know why those dates are critical. Are dates externally imposed like the trade-show that won't reschedule just because we're not ready? Are dates tied to manufacturing realities, vendor dates, customer contracts, revenue goals, market windows, competitive threats or those motivational stretch goals? Visibility needs to run both directions. It's motivating for development to know why the dates are critical. Management needs to know what is possible and where the time is being spent.

Product Needs

With the bright idea, there is always a list of important product needs. Usually these come pre-trimmed so that all development sees is the absolute must haves. They all are priority one. They are also at a high level of abstraction; they are not too precise. In these imprecise and highly important features lie the detailed features that are needed, along with others that are not needed and everything in between. When the realities of development set in, we will see that some feature aspects are more valuable than others; some are needed before others; and some won't ever be developed at all.

Technology and Architecture Goals

Maybe the new product has some specific technology drivers, like compatibility with industry standards, downloadable code through the cloud, 3-D graphics, or transition to an embedded Linux.

Who Participates in Iteration Zero?

Iteration 0 is a team activity. You'd like the whole team to participate. Some of your situations will not make that very practical, like a very large team, multiple teams building a product or a distributed team.

Generally you'd like to have these people present

- People with a vision of the product being developed
- People that understand why features are needed and how they will be used
- People that will build the system
- People that will test the system
- People that fund the system
- Technology and Domain Experts

The attendees should include everyone involved in developing, specifying, and testing the product. You can probably see that the intention is to be inclusive. In a game company I've worked with, the participants included game designers, producers, software engineers, sound engineers, artists, testers, systems engineers and project managers.

Outputs From Iteration Zero

At the end of Iteration Zero the team is ready to start iterating. Knowledge won't be perfect, it never is. After Iteration Zero we have enough to get started on some of the high priority work. Avoid wasting time at the fuzzy front end. With iteration 1, the team can start working on what is clear and important, which provides time to figure out the things we don't understand. Let's look at some of the outputs from Iteration Zero.

Product Vision

The product should provide initial answers to questions like these:

- Why build it?
- What is it?
- What are the critical dates?
- What problem or need does the product meet?
- What are the key business drivers?
- What are the target markets?
- Who are our external, or internal, customers and suppliers?

Record the product vision on a Big Visible Chart (BVC). It does not have to be pretty, just visible. The team needs the vision to make tradeoffs during product development.

Expect the product vision to change as the development effort evolves. Some of your initial assumptions will be right, other wrong and others will just evolve. As the vision changes, update the BVC. Make sure the team knows when the vision changes. It is motivating to be working toward a valuable end and a shared vision contributes to that. It is demotivating to work towards a goal and discover that it stopped being critical a month ago.

Architectural Vision

The architectural vision is not a formal design document. It is the initial partitioning of the system from a hardware and software perspective. The goal is not to figure it all out up front, but to make appropriate provisional decisions about the partitioning so that the learning can begin and the ideas can be tried. The depth and the effort needed for the vision will vary for different teams and products. A distributed team will need more of the vision documented to communicate to remote members. A collocated team can keep the vision on white boards while it evolves, and commit it to a document as they need.

Product Team

It is important to have the right people on the team to avoid cross team handovers. They cause delay. It's helpful to think of the team as consisting of two major roles, product owner and developer. I'll sometimes refer to the product owner as the customer.²

The development team is made up of the people that design and build the product.

The product owner could be a single person, though usually in product development there is a product owner team, led by the product owner. The customer team specifies the product, and is the customer of the

² Product Owner comes from Scrum. Customer comes from Extreme Programming

development team. In product development this is usually an internal customer, rather than an end user. The voice of the end users must be represented by the customer team. Every development effort is different, but a customer team could be made of these people:

- Product manager
- Systems engineer
- Test engineer
- Tester
- Business or Product analyst
- Marketing specialist

The customer team may change over time. Their duty is to speak with a unified voice to the development team. Multiple people on the team does not mean that developers have multiple voices. Sometimes when a product being developed is a platform or will serve different parts of the market, customer team members have different goals and priorities. The customer team should try to work those differing product and business goals before engaging the developers. I am not suggesting slipping back into making all product decisions before engaging development and throwing it over the wall, keeping development out of business decisions. I am suggesting that the customer team should be respectful for development's time and work out what they can independently. Development will still have their say when the customer team shares their current view of features, priorities and goals.

Sometimes the customer/developer relationship is not so obvious. For example, when hardware is being developed concurrently with software, the hardware developer may act as a customer to the software developers during board bring-up. At the same time the software developer might have a customer relationship with a hardware engineer.

Having Test people as part of the customer team is critical because many detailed specifications are written as test cases. The customer defines done, and done means passes its tests.

Avoid sharing people across teams. Specialties might make this inevitable, but if you do share, the person should work to transfer knowledge so the team can be more self sufficient.

Product Stakeholders

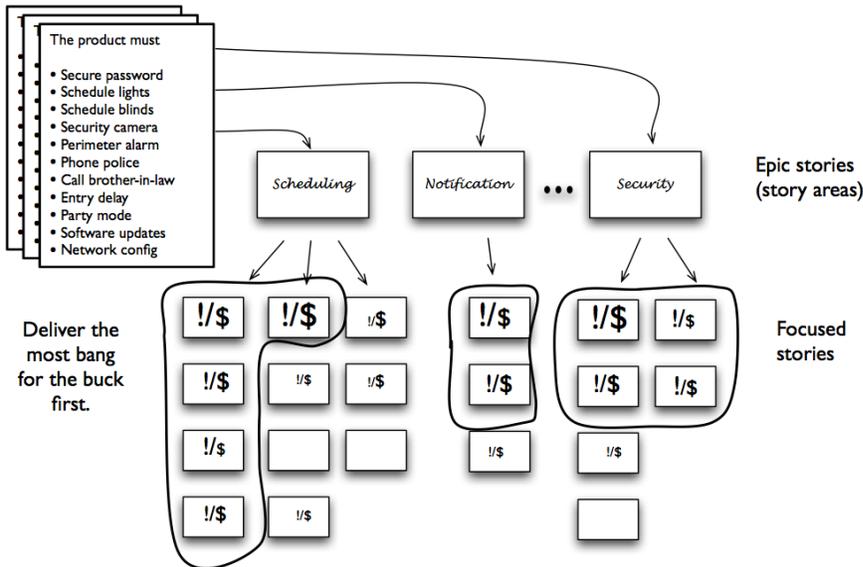
It is important to keep communications open with the stakeholders of the product. Upper management is investing a lot in the product development, so it is critical to make it easy for them to follow the progress. Be proactive. You will see later that having visible planning artifacts, and regularly scheduled planning meetings facilitate open communications with product stakeholders.

Product Story Backlog

The product backlog is made up of all the work that is needed to build the product, and then some. The backlog is made up of Product Stories. Most materials on Agile Development talk about User Stories as the unit of work. Because in product development, many stories are not really visible to the end user, I prefer to call them Product Stories, or simply Stories.

Product stories come in a variety of sizes. Though they need to be fairly small to be chosen for the current iteration. Stories start out big, what Mike Cohn calls epics [COHN1].

Epics might be like the headings in requirements document, or feature specifications. The following diagram illustrates that the epics are broken down into stories.



Copyright 2010 © James W. Grenning. All Rights Reserved. For use by workshop attendees

Each rectangle represents a story, usually written on a note card. Use note cards in story decomposition and planning. They are easy to manipulate, and somehow make the work more tangible. For distributed teams and long term memory, some teams put their stories into a spreadsheet of some sort of agile backlog management software. I recommend that you first learn the techniques with note cards and later find a tool if one is needed.

You are probably used to a work breakdown structure. A work breakdown usually focuses on the tasks to be performed, often leaving big integration activities for later in the development timeline. The story approach is a feature breakdown, where the focus is on delivering pieces of what the product must do. Integration happens much more regularly.

Each story has a value and a cost represented on the cards in the above diagram as !/\$, and read as bang for the buck. The sizes of ! (bang) and \$ (buck) show that some stories are more valuable than others while independently stories have differing relative costs. We know we've broken the epics down to the right level when we start to see work that we won't do, work whose bang is not worth the buck. The fine grained backlog allows the product owner to select the most valuable parts of the epic features for development.

You will see later that we quantify the cost of stories, but let the value be a judgement call without putting numbers on the cards. Some teams will also try to put values on the cards. Consider that advanced agile.

MuSCoW Analysis

When the customer team understands the bang for the buck (I/\$), then well informed decisions on story priority can be made. Anyone who has been in development for a while knows that we always want to put more into a product than we have time or money for. When we look at requirements only from the high level, all the requirements are priority one. When we start to break down features into smaller pieces, we start to see stratification in priorities. Let's look at a technique called MuSCoW analysis. [MUSCOW]

MuSCoW analysis is a very useful technique for prioritizing work. Stories can be classified into these categories

- Mu - Must have
- S - Should have
- Co - Could have
- W - Won't have (very soon anyway)

When choosing stories for an iteration, or planning the next release, MuSCoW comes in handy. Stories let us manage project scope with fine grains. In order to apply MuSCoW, we need well structured stories. Let's look at how we get them.

INVEST in Stories

Mike Cohn describes the acronym INVEST as a reminder of six important attributes of a story [COHN1]

- I - Independent
- N - Negotiable
- V - Valuable
- E - Estimable
- S - Small
- T - Testable

It is difficult to initially get features into INVEST sized stories. The best stories are not centered on one layer of the system, but rather cut across layers of the system. This is counter to how most embedded systems software is developed where individual engineers own specific layers or components. We want layers and components in the system, but we don't want our team to specialize in just one single area. This apparent optimization causes a bottleneck, delays in integration, and a less flexible and knowledgeable team.

Stories cut across layers and components so that integration happens earlier and more often. I am not saying we'll never slice a story at a layer, it's just not the first choice.

When we're trying to get stories that adhere to INVEST, we sometimes need to apply these techniques: Split, Stub, Spike, and Time-box. [GRENNING1]

- Split - cut out some functionality, forming 2 or more stories. Split known from unknown. Split at a component boundary when you must.

- Stub - make a stub implementation of a dependency so that the lack of the dependent item does not block the rest of the story
- Spike - a spike is an experiment. Some stories are large because we don't know enough to estimate them. Use a spike story to go learn something so that the story can be estimated or split.
- Time-box - leave the story as is but agree to limit the amount of time spent on it. This is usually applied to spikes.

Now let's look at each of the INVEST attributes.

Independent

Stories should be independent. The order of development should not matter. Like many objectives, they cannot be met all the time. Although you will probably be surprised at how many stories can be kept independent. Another interpretation of 'I' is immediately actionable. It's something we can do now.

Negotiable

Stories are negotiable. We can negotiate what is in and what is not in the story? We negotiate stories usually when they represent too much work take on at once. When a story is too big, split it. If it is too small, aggregate it with another. If part of a story is known, but part unknown, split known from unknown. Stories can be split by the test cases that they have to support. Consider splitting the happy path of a given feature from the error paths.

Valuable

Ideally stories should deliver immediate value to the customer. This may be unrealistic for the stories for an embedded software product where much of the product must come together to show anything. So we often have to settle for the second interpretation of V, Visible. Sometimes Visible is all you can get.

In embedded development, there are often technology and architecture goals. The connection between the goal and the user is not always direct. The technology and architecture goals often influence the viability of the product. Sometimes the goals can be realized just by having the architectural vision to guide development. I'd prefer to not have a story like "port embedded database" and prefer to have a story about "save and restore product configuration" that pulls in the embedded database porting work.

When the infrastructure work is just too big, find a way to express it as a series of demonstrations rather than having no visible progress for several weeks or months. If you had a system that needed a flash file system and there was not even a flash driver yet, one of the early visible stories to demonstrate progress would be to turn on the flash memory device's LED.

One of my clients is building a system with a robotic arm. We wrote stories about some of the specific movements that need to be made like, open the grabber, close the grabber, move arm up, or move arm to home position. You can see that these are not valuable in themselves, but show visible progress toward the value.

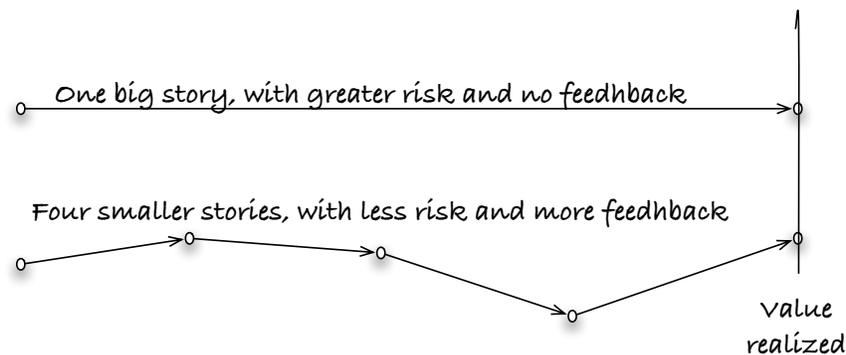
Estimate-able

Stories are the unit of work for agile product development. The stories supports planning and tracking and consequently must be estimate-able. Stories that are too big are hard to estimate for a number of reasons. Stories are often vague and contain too much. The developers may not have the technical knowledge to make an estimate. The product owner may not know exactly what they want. These are all natural and normal when a team is inventing something new, or adding a major new capability to a system.

People are not so good at estimating big complex features and systems, which brings us to the next attribute of INVEST.

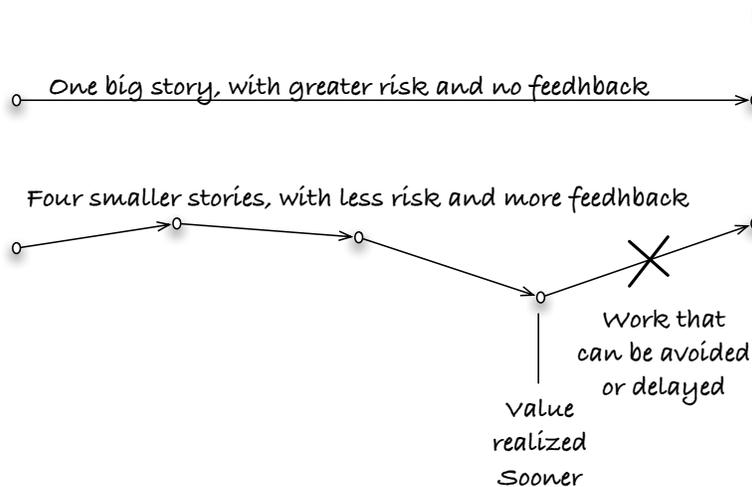
Small

Stories should be small. Small enough so that many fit in the iteration. Consequently, multiple stories should be completed in each iteration. People are better at estimating smaller pieces of work. Let's compare bigger stories to smaller stories using the next two diagrams.



When stories are big, there is more risk and less feedback. If the timeline represents one month, we only get a really good data point once a month on the feature progress. Smaller stories let developers get feedback from customers so the right features get developed. Developers, have you ever spent a month only to find you delivered what was asked for but it was not what was needed? Small stories also provide more regular management information to help determine if development is on track.

Another thing happens when we break stories into smaller pieces as this diagram shows.



After the first few stories are delivered, we might discover the final story is not needed, or something else is needed more. A big part of agile development is finding the work you don't have to do.

Making stories smaller is a challenge for embedded development teams. Use split, stub, spike and time-boxing to cut the stories down to size.

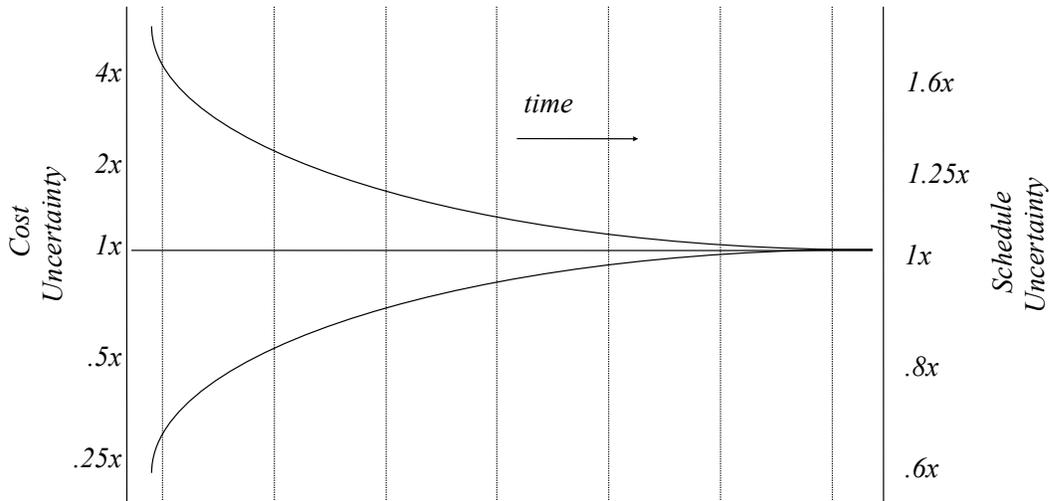
Testable

For a story to be done, it has to pass its tests. That means that stories must be testable. Stories are tokens for the work that has to be done, and they are usually vague and ambiguous. They are the name of the functionality, a promise for a conversation. The tests provide the detailed requirements just in time for the development team. A helpful way to clarify a story is to ask the product owner and associated test people, what tests will demonstrate that the story is done.

Stories are the fine grained work that makes up the product backlog. Let's see how stories are used in agile planning and tracking.

Agile Planning and Tracking

Barry Bhoem, pioneer in software development, drew this diagram to show the uncertainty involved in development efforts. Regardless of the axis being correct or not, the graph shows that early in the development effort there is great uncertainty. It is not until the product is delivered that the actual scope, cost and date are known with certainty.

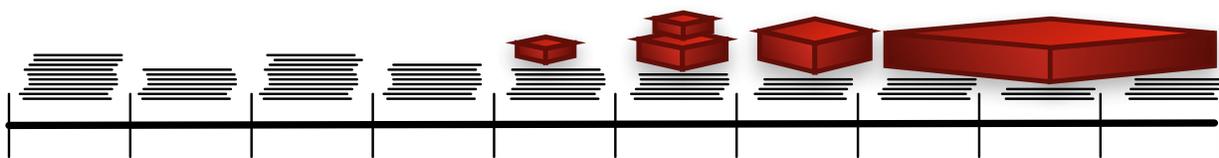


What we'd like is an estimation and planning mechanism that accepts this law of software physics and helps us to close the gap of uncertainty.

Agile estimation and planning is designed to help the estimate converge more rapidly with reality by having feedback in the system. It's not that Agile planning methods ignore dates and the developers have a take-it or leave-it attitude. Dates are critical. We have to manage to the dates while being realistic about what can be achieved by the people doing the work. A release plan is one of the tools. It can be used to manage to a specific delivery date, or deliver a specific content. Most often a planning is a combination of the two.

Release Plan

Stories are laid out in a release plan. This diagram represents the product backlog with specific stories in each iteration. Each iteration is a set of stories, viewed edge on with each story written on a notecard. The boxes represent larger epic stories that have not yet been broken down.



Notice that the first few iterations are broken down into small stories, ready to be developed. Generally, the stories in the next few iterations should be more detailed, as the work is close at hand. Stories later in the plan can be bigger and more ambiguous. This is just like planning in your daily life. You probably know in great detail what's for dinner tonight, and only have a vague idea of what you'll prepare over the next two weeks. Sometimes you will break down the work for a more detailed plan, but often deciding too soon has little advantage.

One key idea is that the system is releasable at any iteration boundary. In a concurrent hardware/software development effort, this probably cannot be true in the early iterations, but can be true in the later ones. So, a limited functionality prototype can be scheduled to use the system at the end of iteration 5, while a field ready product is scheduled for the end of iteration 10. With the give and take from one iteration to the next,

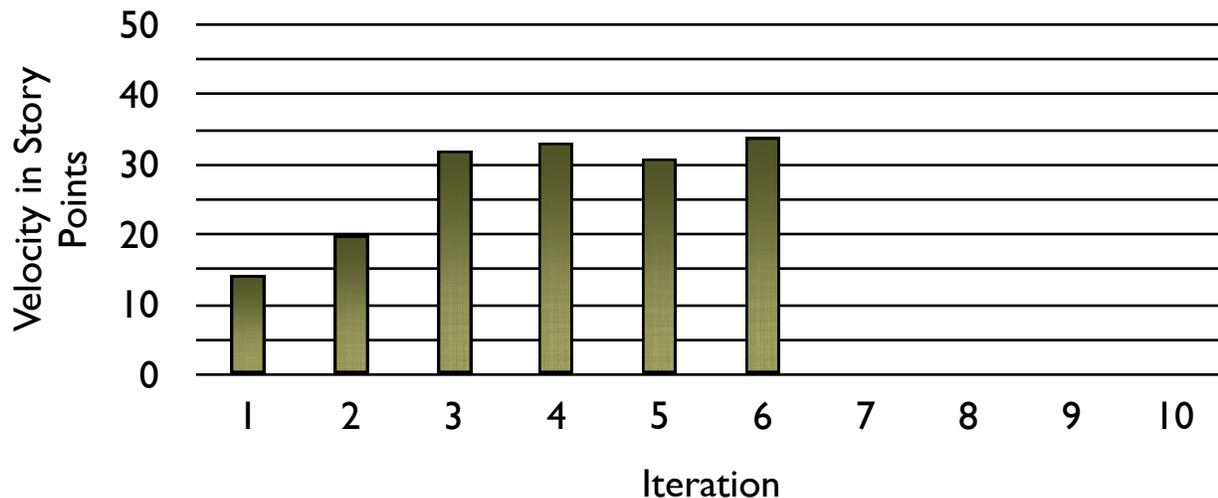
and the big ambiguous epics out in the second half of the plan, the exact content of each release is not predictable, but we can use MuSCoW to have the most interesting releases we can on those dates.

How far to break down the epics is a judgement call. It is important to have a few iterations of work that is ready and important. If your backlog does not have ample small stories, it is likely that the stories with the most bang for the buck are not being worked on.

Break down the epic when you want a better view of the work to come, or reduce schedule risk. Break them down if there is a high risk or some of the epic content is critical to the product success. Realize that making the plan more detailed will take effort away from working on the product. The increased planning effort might not be worth delaying development.

Velocity

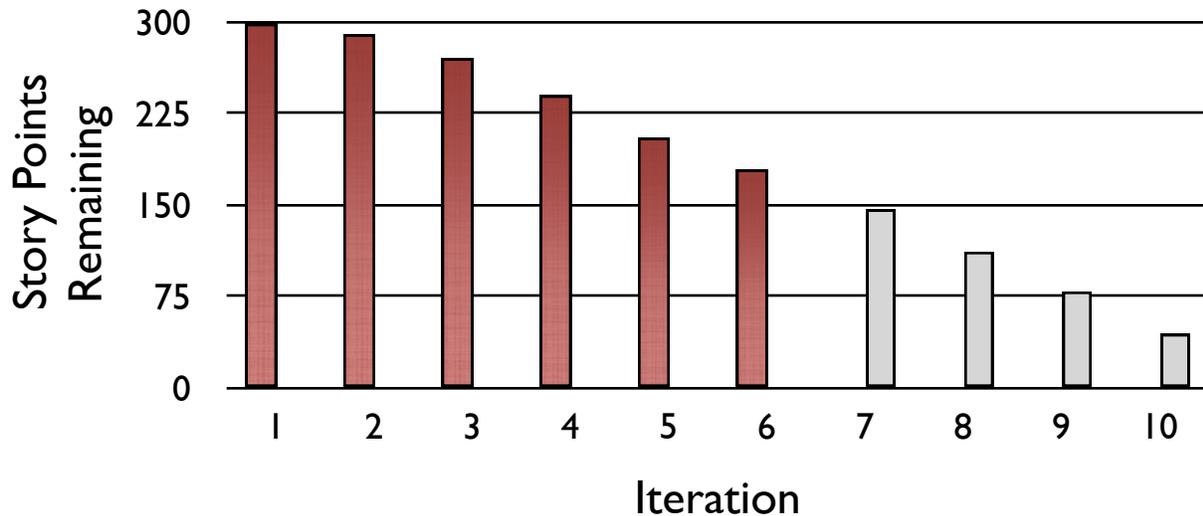
Velocity is the measure of progress made to deliver the product. Initially velocity is estimated; later it is measured. This graph represents a team's velocity as measured over six iterations.



Velocity is measured in story points per iteration. We'll talk more about story points and estimation in the next section. Story points measure the effort needed to complete a story. In the above chart the team initially had a velocity of about 14, and as of iteration 6 the team is consistently getting 30 to 35 points completed each iteration.

Burn-down

If every story in the backlog has a story point estimate, we can create a burn-down chart, like this.



With a reasonably consistent velocity, the burn down chart can be used to see when all the identified stories will be complete. Assuming the goal is to release after the 10th iteration, this chart shows the usual situation that there is more work to do than there is time.

Dates are critical to business, so it is natural to manage a development effort to dates. We always want to put more into the product, and usually there is not enough time. What the burn-down chart provides is an early warning of variation between desired scope and date and the most likely scope and date.

Early warning means you have options

With this early warning, teams have options, unlike the 11th hour “we’re not going to make it” moments, where there are no options but to delay or ship poor quality. Early warning give time to adjust development.

Add people

Fred Brooks said that adding people to a late project makes it later. [BROOKS] This is because the newly added people take time away from those already working to deliver the product. There is an initial productivity loss.

If you add people early enough, it is much more likely that the team’s short term velocity hit is more than made up during the remaining part of the timeline.

Change the date

Some dates can’t be moved. An industry trade show is not going to change its date just because your product is not ready. Although not all dates are immovable. Some dates are arbitrary, others may be unpleasant to change, but can be changed when the evidence of progress and likely outcome is compelling. Fine grained scope control with stories, velocity and burn-down give that kind of evidence.

Adjust the scope

Developing iteratively, using stories to drive the plan, allows a team to do the most critical features first. The most valuable stories can be chosen from the backlog. Priorities can be done using MuSCoW analysis. The trade show can be attended with the most important demonstrable features on the date.

In a traditional up front plan and design approach, valuable time may be spent on features that get cut to meet a date. If the stories drive an incremental delivery plan, along with sound incremental development engineering practices, effort can be used for only the features that are delivered. Upfront work for features that had to be cut in a traditional development effort is not done, and instead is spend on features that are delivered.

Estimation

Estimation is educated guesswork. You are inventing something, so we cannot expect perfect future vision. Beginning a product development effort and fixing the scope, people, and exact delivery date is close to insanity, especially if you keep doing it and get the same high-stress result.

Ask an engineer for an estimate for a new feature. Promise that it will only be used for budgetary and planning purposes. Assure him that he won't be held to it. You'll see arms crossed and an effort to back away. People don't like giving time estimates because even with all the assurances, the number they give too often goes right into the plan and very soon it is viewed as a commitment.

We want the people doing the work to estimate it, and we want estimation to accept that it is educated guesswork.

Problems with traditional estimation efforts

- People reluctantly give time estimates.
- When the estimator is not the person doing the work, the estimate will probably be wrong
- When the estimator is not the person doing the work, the person that has to do the work will not own the estimate.
- Estimates are viewed as commitments

Agile estimation and planning are designed to avoid some of these problems. Mike Cohn, in Agile Estimation and Planning, describes these estimation steps [COHN2]:

- Estimate the relative sizes of each story (story points)
- Estimate the velocity for the team (story points/iteration)
- Measure the actual velocity and feed that back into the plan.

One of the key points here is that estimates are not done in units of time that we are used to like developer-hours or developer-days. Estimates are made in story points, a unit-less number. Typically the stories requiring the least effort to implement, but all about the same effort, are assigned the story point value of 1. The rest of the stories are estimated in integers and are multiples of the stories with the least effort. For example, the stories that are estimated at eight points are estimated to take eight times as long to develop as stories with a one point estimate. Stories are small enough when you can envision that numerous stories can be completed in an iteration.

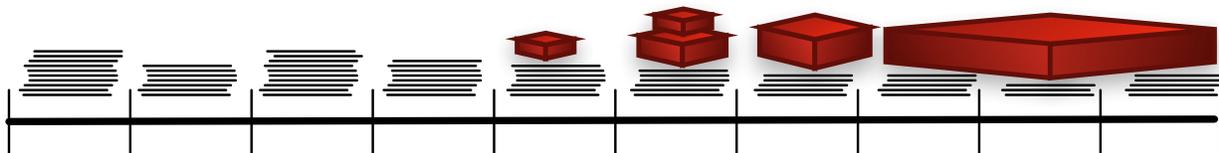
Story points work, because:

- The pressure of time estimates is removed.
- People are good at relative estimates for smaller items
- It does not matter who will do the work, the estimates are not in calendar time so differences in personal velocity do not effect the story point total
- Developers own the estimates
- They can be used, along with velocity to create and evolve a realistic plan
- With the teams run rate as feedback it is a self correcting system

Warnings about velocity misuse

- Don't create incentives for velocity goals
- Don't make stretch goals
- Don't measure individual velocity
- Don't compare the velocity from one team to another

With each story having an estimate, and the team an estimated velocity, a release plan can be formed. The product owner team uses their judgement, and their sense of the relative value of the stories, the date goals of the team, and a cost for each story to lay out a release plan. Developers sometimes suggest pulling some stories forward to reduce risk and manage dependencies (though often dependencies can be managed through stubbing).



Traditional development plans where much effort is placed on nailing down time estimates takes much longer to create than a story point based plan. It appears to be more precise, but it is still guesswork. In an agile development effort, depending upon scope, a plan can be put together very quickly so that work can begin, and the plan revised as we learn by doing. I've coached many teams through initial story creation and estimation where a good first draft plan is completed in a couple days. In one case the product development was also planned by traditional means, and we came up with the same answer. The traditional plan took many weeks of effort to develop. (I admit that the team that had a traditional plan that converged could in part be because of the ground work on the traditional plan, but still the team was surprised when it arrived at a similar estimate through such a different means.)

With a traditional plan, we struggle to be precisely right, but usually end up precisely wrong. In agile we are prefer generally correct to precisely wrong. We can get generally correct more quickly too, and then refine as we go.

We've talked about the estimates long enough. So, how do we get these estimates? Let's see.

Planning Poker

Planning Poker is the most popular estimation technique used by agile teams. Planning poker was invented to solve a deadlocked planning meeting. In a conference room in American Fork, Utah 2002 we had a backlog to estimate. I was the coach. The customer read a story. The two senior engineers discuss the impact of the story on the system. Reluctantly, an estimate is tossed out on the table. They go back and forth for quite a while. Everyone else in the room is drifting off, definitely not engaged. The discussion oscillates from one potential solution to another, avoiding putting a number on the card. When the discussion finally ends, the estimate did not really change over all that discussion, 20 minutes wasted. A few more stories go through the same pattern. With 25 more stories to estimate, how can we get this meeting moving? We took a break, and when we returned I had everyone pick up a note card, and listen to the next story. Next they wrote their estimate on a card and placed it face down on the table. Then all revealed their estimates simultaneously. We converged sometimes but not always. There was discussion when we did not agree. After a while each player had a hand of cards, it looked like a poker game. Planning Poker was born. We got through the stories with time to spare. [GRENNING2]

The problem with that meeting was that only two guys were engaged, and even when they agreed they choose to talk about it, and talk about it... Everyone else nodded and slept. It's the team's estimate and they need to be engaged.

The mechanics are easy. Each developer has a set of planning poker cards with a sparse set of numbers. Mike Cohn, in Agile Estimation and Planning [COHN2] suggests using a modified Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, 40, and so on). In my original paper I suggested a sparse sequence where when the numbers got bigger, there were larger gaps. I wanted to avoid arguing 10 vs. 11. Remember, we're not going to precisely wrong, but generally right. I prefer a set of numbers that are easy to add up for quick estimates (1, 2, 3, 5, 8, 10, 15, 20, 30, 50, and so on).

The product owner reads a story. Developers ask questions about the story is it is not clear, or to determine what is included and what is not included. A good question to ask is "How will it be tested?" The all developers, playing their cards close to their chest, choose a card and place it face down on the table. When all player are down they roll over their card. If they all have the same estimate, the estimate is written on the story and the next story is read. It does not matter if each developer has a different implementation in mind, they agree on the effort, so the estimate is agreed.

When developers do not agree, the outliers discuss their estimates. The low outlier will describe why the story is so easy, the high outlier will describe why the story is so hard. Maybe the product owner will chime in and what is included in the story. The whole team plays again. Usually estimates converge quickly. If they don't pull the card out, to discuss later, or average the estimates, or take the high or low. What ever the team decides is OK.

Planning poker is not always the right tool for the job. It goes much more smoothly when a team has a set of baseline story estimates so that they have a feel for the size of a story point. Planning poker is not the best tool for estimating a large backlog. The Planning Poker Party is the right tool though.

Planning Poker Party

The Planning Poker Party is designed for estimating a large batch of stories. Typically an Iteration Zero backlog has 100-200 stories. In my workshops we can typically create an initial estimate in half a day. With a large batch of stories we go through a sequence of planning steps described in the following subsections. It's best to do this activity with the whole team around a large table. There is a more detailed description my blog [GRENNING3].

High-low Showdown

In high-low showdown we're trying to reduce the number of stories in play for each round of the next game. Lay out five cards on the table marked like this:

- Low
- Medium
- Hard
- More info
- You must be kidding!

The markings represent the relative effort for the stories placed under that heading. Before play begins the product owner would have gone through the backlog with the team, so everyone would be familiar with the stories. The dealer reads a story, some developer (or developers) offers what pile to put it into based on a guess of the relative effort. This should be fast. Don't waste time discussing which pile a specific story lands in. You will get some wrong, but it's OK because in the next activity it will be evident. The real objective of this is to end up with a about $\frac{1}{3}$ of the card in each of the high, medium and low stacks. Of course there will be some on the more info and you must be kidding stacks. Avoid long discussion, just get the cards into a stack that is close enough. We refine estimates in the next activity.

If you have less than 75 stories, you can skip High-low showdown and go right to deal and slide.

Deal and Slide

Start deal and slide with the low effort pile of stories (or the whole pile if you skipped high-low showdown). Spread them out on the table. Depending on the size of your backlog, you might need a big table before we are done. I prefer a pool table.

Developers, in silence, start sliding the story cards around the table, putting the easiest stories to the left and the hardest to the right, forming columns of similar difficulty. Anyone can slide any card any number of times, but if a card won't settle down, a Nervous Nellie, it should be taken out of play. Once the cards that can settle down, do settle down, talk about the Nervous Nelly cards and see if you can settle them. If you can't it is a sign that more info is needed, or someone is kidding.

Repeat the process with the next stack of cards. Don't worry if you find some lows in the mediums, mediums in the highs or vice versa. It will all work out. The high-medium-low sorting was just a way to avoid having 200 cards on the table at the start of the game.

With all the low, medium, and high stories laid out, it is probably not a bad idea to go back and look over the columns and decide if the stories are placed correctly. Next, the columns need headings.

Planning Poker for Groups

In this round, label the headings of the columns with their relative estimates using a planning poker-like approach. The columns to the left should have small numbers and to the right larger numbers. You might find that you join some adjacent columns to avoid splitting hairs over some of the estimates. Remember, generally right, not precisely wrong. If your product owner wants a ballpark on the more info and you must be kidding stories, lay them out and put appropriately large estimates on them as they deserve.

Once the cards are down and columns have estimates, write the estimate on each card.

Developer Guts

In developer guts, developers estimate their velocity. The product owner is supposed to choose the stories for each iteration, but in developer guts I let the developers pretend that they get to choose the content of the first couple iterations. Here's how it goes:

Developers choose a set of stories that they think they can complete in a two week period (assuming a 2 week iteration). Choose another set of stories. Add up the numbers. Are the sums similar? Discuss, repeat and somehow show some guts and choose an estimated velocity.

Once you have completed a few iterations, you will stop playing Developer Guts and use the measured velocity (the actual number of completed points in the last iteration). The practice of using the last velocity as the velocity for the coming iterations is called Yesterdays Weather. [BECK] The name is based on an effective weather forecasting technique, based on the fact that the most likely weather forecast is that it will be the same as yesterday.

Customer Guts

All the stories are estimated; the developers have made a guess at their velocity; now the customer (product owner team) must choose a series of iterations. The customer might accept the first two iterations that the developers put together during developer guts, but it is the customer's call.

The next few iterations should be largely made up of smaller stories, usually ones with estimates in the single digits. These are well understood, or at least focused bits of work. Bigger stories should be broken into smaller ones no later than the start of an iteration. When a big story is not fully delivered, no value is delivered to the customer, and no points are added to the velocity. Smaller stories improve both those situations.

Replanning

Locking into a plan means that all the future things you learn cannot be incorporated into the plan. Agile plans are alive, so plan to replan. Times to replan:

- After a few iterations
- When the velocity widely varies for no good reason
- When new stories are added to the backlog
- When you know the current plan is not right

Get Ready to Start Iterating

Although the product owner can in an emergency change the priorities of the team, it's best not to change an iteration in progress. Valuable work in progress and context are lost. In the non-agile approach, a new field issue, or urgent change would have to sit around for weeks or months to not interrupt the big chunks of work in process. With a two-week iteration, on average an urgent issue will only have to wait one week until it can move to the head of the priority list.

If an iteration had to be changed, estimate the emergency work and remove an equal amount from the iterations. There are other strategies for dealing with regular but reactive maintenance work, like having one developer each iteration be the lead for new emergencies, or have an unspecified support story in each iteration to reserve some bandwidth.

Iteration Planning Meeting

Each iteration there is a planning meeting that wraps up the previous iteration and starts the next. Some teams break these into two meetings. The product owner team should have decided on the stories for iteration before the meeting so the customer team can speak to development with one voice. The meeting should have a regular time slot, and be well attended. Once you get good at these meetings, they should only take a couple hours.

Here is a short list of activities to wrap up each iteration.

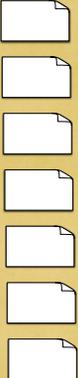
- Demonstrate the completed work
- Record the team velocity
- Update the burn down chart
- Do an iteration retrospective for the prior iteration
 - What worked?, What were the problems? What should we do differently?

Here are the activities to plan the next iteration:

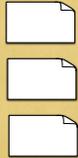
- Product owner presents the stories
- Developers double check estimates
- Some stories might get split
- Developers discuss architectural impact
- Developers choose which stories they will be responsible for
- Update the iteration race track

Iteration Race Track

To monitor the work in progress we put up an iteration race track in a visible area, preferably where we also do the daily standup meeting. A cork board, a white board, or a cubicle wall works fine as a race track. Initially it looks like this:

Selected	Started	Dev Done	Accepted
			

As stories are implemented, the race track will show the state of the work in progress.

Selected	Started	Dev Done	Accepted
			

If this information radiator looked like this at the iteration midpoint, the team should be concerned. Half the time is used, but half the story points are not accepted, or even claimed to be done by the developers.

Team Agreement on Working Practices and Learning Goals

Teams starting an agile development effort should also form and an agreement on the practices and standards they will follow. Agile is about working in self-managing teams and making the process your own.

Many teams draw from the planning practices of Scrum and the engineering practices of Extreme Programming. Some of the choices are mechanical:

- Daily stand up meeting time

- Iteration start day of the week
- Iteration length
- Where the team will meet

Some other practices and decisions are not so easy to make:

- How to apply automated testing to legacy code
- Adopting Test Driven Development
- Setting up a continuous integration server
- Coding standards
- Pair programming, code reviews or both
- How to automate acceptance tests
- Making a shared workspace and or team room
- Working with other teams or remote developers

Other Iteration Zero Activities

In this paper I have focused on the planning aspects of iteration zero. Teams often use iteration zero to get tooling and training in place so that iteration one can focus on building a slice of the product. Some of the activities are:

- Training in Agile Development and Test Driven Development
- Setup up tools for Test Driven Development
- Setup up tools for Story Testing (Acceptance testing)
- Setup up a continuous integration server (Hudson, Cruise Control, for example)
- Setup reporting conventions and tools

Summary

Agile planning is adaptive planning. The plan is centered on what is important to the business and the end user of the product. All plans are guesses and they have to be continually adjusted and evaluated. Making the work visible to non-engineers as stories and demonstrations lets stakeholders see the progress in things they understand. The backlog is made up of stories. The progress is measured in velocity and visualized in the burn-down chart. These are simple but powerful tools.

Development organizations always want more in their systems than time or effort will allow, so it will be natural to have to make tradeoffs to meet date of feature content goals. The difference with an adaptive agile plan is that it provides early warning when the plan execution deviates from the goal. The plan is easy to demonstrate to non-developers. Early warning usually means that there are more options. It is a significant management advantage to have more reliable plans, and plans that give early warning of problems. It is valuable to product owners to decide what feature slices to add first rather than waiting for infrastructure to be built and only seeing visible progress near the end of the development lifecycle where there is little time to react.

A pilot project at a recent client had a track record of typically being three months late, usually only discovering the schedule gap very late in the development cycle. With their first project the stories gave them credible evidence of the scope of the product and its duration. They worked a few iterations to

establish a velocity. The news was not good. They took options to the product owner. We can deliver on time with these features, but these other features cannot be done by the deadline. The scope in jeopardy was laid out in iterations. After a little MuSCoW analysis, the product owner settled on adding two iterations and adjusting the content. The business could adjust to the new plan because there was ample time, and flexibility in the organization.

Bibliography

[MCCONNELL]

[BROOKS] Brooks, Fred, The Mythical Man Month

[COHN1] Cohn, Mike, User Stories

[COHN2] Cohn, Mike, Agile Estimation and Planning

[GRENNING1] Grenning, James, Planning Poker,

<http://www.renaissancesoftware.net/blog/archives/48>

[GRENNING2] Grenning, James, Planning Poker,

<http://www.renaissancesoftware.net/papers/44-planing-poker.html>

[GRENNING3] Grenning, James, Planning Poker Party,

<http://www.renaissancesoftware.net/blog/archives/36>

[MUSCOW] Leslie M. Tierstein, W R Systems, Ltd., MANAGING A DESIGNER/2000
PROJECT, NYOUG Fall '97 Conference