## An Agile Approach

With an understanding of the four primary agile value statements, we can turn our attention to what an agile team looks like in practice. Taken collectively, the four value statements lead to software development processes that are highly iterative and incremental and that deliver coded and tested software at the end of each iteration. The following sections cover some of the main ways in which agile teams work, including that they:

* Work as one team
* Work in short iterations
* Deliver something each iteration
* Focus on business priorities
* Inspect and adapt

# Multiple Levels of Planning

A project is at risk if its planning extends well beyond the planner’s horizon and does not include time for the planner to raise her head, look at the new horizon, and make adjustments. A progressive elaboration of the plan is needed. Agile teams achieve this by planning at three distinct horizons. The three horizons are the release, the iteration, and the current day.

Most agile teams are concerned only with the three innermost levels of the planning onion. Release planning considers the user stories or themes that will be developed for a new release of a product or system. The goal of release planning is to determine an appropriate answer to the questions of scope, schedule, and resources for a project. Release planning occurs at the start of a project but is not an isolated effort. A good release plan is updated throughout the project (usually at the start of each iteration) so that it always reflects the current expectations about what will be included in the release.

At the next level is iteration planning, which is conducted at the start of each iteration. Based on the work accomplished in the just-finished iteration, the product owner identifies high-priority work the team should address in the new iteration. Because we are looking at a closer horizon than with release planning, the components of the iteration plan can be smaller. During iteration planning, we talk about the tasks that will be needed to transform a feature request into working and tested software.

Finally, there is daily planning. Most agile teams use some form of daily stand-up meeting to coordinate work and synchronize daily efforts. Although it may seem excessive to consider this planning in the formal sense, teams definitely make, assess, and revise their plans during these meetings. During their daily meetings, teams constrain the planning horizon to be no further away than the next day, when they will meet again. Because of this, they focus on the planning of tasks and on coordinating the individual activities that lead up to the completion of a task.

By planning across these three time horizons—release, iteration, and day—agile teams focus on what is visible and important to the plan they are creating.

#### Conditions of Satisfaction

Every project is initiated with a set of objectives. Your current project may be to create the world’s best word processor. Creating the world’s best word processor, however, will typically be only one objective for this project. There will almost certainly be additional objectives regarding schedule, budget, and quality. These objectives can be thought of as the the customer or product owner’s *conditions of satisfaction*—that is, the criteria that will be used to gauge the success of the project.

At the start of release planning, the team and product owner collaboratively explore the product owner’s conditions of satisfaction. These include the usual items—scope, schedule, budget, and quality—although agile teams typically prefer to treat quality as non-negotiable. The team and product owner look for ways to meet all of the conditions of satisfaction. The product owner may, for example, be equally satisfied with a release in five months that includes one set of user stories as with a release a month later that includes additonal user stories.

Sometimes, however, all of the product owner’s conditions of satisfaction cannot be met. The team can build the world’s best word processor, but they cannot build it by next month. When no feasible solution can be found, the conditions of satisfaction must change. Because of this, release planning and exploration of the product owner’s conditions of satisfaction are highly iterative, as illustrated in [**Figure 3.2**](javascript:moveTo('ch03fig02');).

Once a release plan covering approximately the next three to six months is established, it is used as input into the planning of the first iteration. Just as release planning began with consideration of the product owner’s conditions of satisfaction, so does iteration planning. For an iteration, the product owner’s conditions of satisfaction are typically the features she’d like developed next and some high-level tests about each feature.

Like release planning, iteration planning is iterative. The product owner and the team discuss various ways of best meeting the conditions of satisfaction for the iteration.

Feedback loops are shown in [**Figure 3.2**](javascript:moveTo('ch03fig02');) from the resulting new product increment back into the conditions-of-satisfaction boxes at the start of both release and iteration planning. Based on their experience developing the product increment during the iteration, the team may have gained knowledge or experience that affects planning at one or more of these levels. Similarly, showing the product increment to existing or likely users may generate new knowledge that would cause changes to the plans. An agile team will incorporate these changes into their plans to the extent that they lead to a higher-value product.

# Release Planning

Release planning is the process of creating a very high-level plan that covers a

period longer than an iteration. A typical release will cover perhaps three to six

months and maybe three to twelve or more iterations, depending on how long

the iterations are. A release plan is important for a number of reasons.

First, it helps the product owner and the whole team decide how much must

be developed and how long that will take before they have a releasable product.

The sooner the product can be released (and the better it is when it’s released),

the sooner the organization will begin earning a return on its investment in the

project.

Second, a release plan conveys expectations about what is likely to be developed

and in what timeframe. Many organizations need this information because

it feeds into other strategic planning activities.

Third, a release plan serves as a guidepost toward which the project team

can progress. Without the concept of a release, teams move endlessly from one

iteration to the next. A release plan provides context that allows iterations to

combine into a satisfying whole.

***The Release Plan***

Part of planning a release is determining how much can be accomplished by

what date. In some cases, we start with a date and see how much can be finished

by then. In other cases, we start with a set of user stories and see how long it will

take to develop them. In both cases, once a team has an initial answer, it is assessed

against the organization’s goals for the project: Will the product developed

make the desired amount of money? Will it save enough money? Will the

product capture the target market share? If not, perhaps a longer or shorter

project may achieve an acceptable set of goals.

At a cursory level, determining how much work will fit into a release and

what user stories that will be is a very straightforward process. Multiplying the

planned number of iterations by either the expected or known velocity of the

team gives us the total amount of work that can be performed. We then select

the number of user stories that will fit and are done. Suppose that we wish to

ship a new product in six months. We plan to work in two-week iterations, so

there will be thirteen iterations during the project. We expect the team’s velocity

to be twenty story points or ideal days per iterations. The size of the total project

is then story points or ideal days. The product owner and team

could then discuss all of the stories and prioritize them to deliver the most value

possible while paying attention to not going above 260. The release plan itself is

usually documented as simply a list of the user stories that will be developed during

the project.

During release planning, we do not want to create a plan that indicates

which developers will work on which user stories or tasks, or the sequence in

which work will be performed within an iteration. Creating a plan with that level

of detail during release planning is dangerous and misleading. Decisions about

who works on what and the sequence of activities are best left to the individuals

working on those tasks and are best deferred as long as possible. Additionally, rememberthat items in a release plan are user stories, which are descriptions of

the functionality to be delivered, not individual engineering tasks to be performed.

***Determine the Conditions of Satisfaction***

Before starting to plan a release, it is important to know the criteria by which the

project will be evaluated as a success or a failure. For most projects, the ultimate

scorecard is the amount of money saved or generated. As leading indicators of

whether a project is likely to achieve these financial goals, most projects use the

triumvirate of schedule, scope, and resources. For most projects, this means that

the product owner’s conditions of satisfaction are defined by a combination of

schedule, scope, and resource goals.

***Estimate the User Stories***

Because an estimate represents the cost of developing a user story, it is important

that each has been estimated. Imagine that you’ve decided to replace every

item of clothing in your closet. You arrive at the mall and start shopping. However,

you notice that all the price tags have been removed and that you have no

way of knowing the cost of anything. This is what it feels like to be a product

owner who is not provided any estimates.

It is not necessary to estimate everything that a product owner may ever

want. It is necessary only to have an estimate for each new feature that has some

reasonable possibility of being selected for inclusion in the upcoming release.

Often, a product owner will have a wish list that extends two, three, or more releases

into the future. It is not necessary to have estimates on the more distant

work.

***Select an Iteration Length***

Most agile teams work in iterations of two to four weeks. It’s possible to go

slightly longer, and some teams have experimented with even shorter iterations.

When planning a release, an appropriate iteration length will need to be chosen.

Guidance on doing this is provided in Chapter 15, “Selecting an Iteration

Length.”

***Estimate Velocity***

If the team has experience working together, your best bet is often to use the velocity

the team exhibited most recently. Naturally, if the technology or business

domain has changed dramatically, it may not be appropriate to use a team’s past

velocity. Still, there are techniques you can apply that enable you to make an informedestimate of velocity based on past results. In Chapter 16, “Estimating Velocity,”we will look at such techniques and will also explore options for

estimating velocity.

***Prioritize User Stories***

Most projects have either too little time or too many features. It is often impossible

to do everything that everyone wants in the time allowed. Because of this,

the product owner must prioritize the features she wants developed. A good

product owner will accept ultimate responsibility for prioritizing but will listen

to advice from the development team, especially about sequencing. User stories

are prioritized based on the guidelines given in the previous part of this book.

***Select Stories and a Release Date***

At this point, you have an estimate of the team’s velocity per iteration and have

an assumption about how many iterations there will be. It’s time to see whether

a release can be planned that meets the conditions of satisfaction for the project.

If the project is feature-driven, we can sum the estimates of all needed features

and divide by the expected velocity. This will give us the number of iterations

necessary to complete the desired functionality.

If the project is date-driven, we can determine the number of iterations by

looking at a calendar. Multiplying the number of iterations by the expected velocity

will tell us how many story points or ideal days will fit in the release. We

can count off that many points or ideal days into the prioritized list of user

stories and see how much functionality can be delivered in the desired time.

The next question to be addressed regards how detailed the release plan will

be. Some teams in some environments prefer to create a release plan that shows

what they expect to develop during each iteration. Other teams prefer simply to

determine what they think will be developed during the overall release, leaving

the specifics of each iteration for later. This is something for the team to discuss

and decide during release planning.

***Updating the Release Plan***

At this point, the release plan is done. However, it’s important that the release

plan isn’t filed away somewhere or put up on a shelf, never to be touched again.

The release plan should be revisited and updated with some regular frequency. If

the development team’s velocity has remained fairly constant, and iteration

planning hasn’t introduced any big surprises, you may want to go as long as four

to six weeks without formally updating the release plan. On the other hand,

many projects benefit from establishing a rule that the release plan will be revisited

after each iteration.

# Iteration Planning

A release plan is an excellent high-level view of how a team intends to deliver the

most valuable product they can. However, a release plan provides only the high level view of the product being built. It does not provide the short-term, more

detailed view that teams use to drive the work that occurs within an iteration.

With an iteration plan, a team takes a more focused, detailed look at what will be

necessary to implement completely only those user stories selected for the new

iteration.

An iteration plan is created in an iteration planning meeting. This meeting

should be attended by the product owner, analysts, programmers, testers, database

engineers, user interaction designers, and so on. Anyone involved in taking

a raw idea and turning it into a functioning product should be present.

Tangibly, an iteration plan can be as simple as a spreadsheet or a set of note

cards with one task handwritten on each card. In either case, tasks and stories

should be organized so that it’s possible to tell which tasks go with which stories.

***Tasks Are Not Allocated During Iteration Planning***

Before looking at the things that are done during iteration planning, it’s important

to clarify one thing that is not done. While planning an iteration, tasks are

not allocated to specific individuals. At the start of the iteration, it may appear

obvious who will work on a specific task; however, based on the progress of the

whole team against the entire set of tasks, what is obvious at the start may not be

what happens during the iteration. For example, when planning an iteration we

may assume that our database administrator will complete the “tune the advanced

search query” task because she has the best SQL skills on the team. However,

if she’s unable to get to this task, someone else may step forward and do it.

Individuals do not sign up for tasks until the iteration begins and generally

sign up for only one or two related tasks at a time. New tasks are not begun until

previously selected ones are completed.

***How Iteration and Release Planning Differ***

The release plan looks forward through the release of the product, usually three

to six months out at the start of a new project. In contrast, the iteration plan

looks ahead only the length of one iteration, usually two to four weeks. The user

stories of the release plan are decomposed into tasks on the iteration plan.

Where the user stories of a release plan are estimated in story points or ideal

days, the tasks on the iteration plan are estimated in ideal hours.

Why are the tasks of an iteration plan estimated in hours but the stories of a

release plan are estimated in story points or ideal days? Primarily because it is

possible to do so. The work of an iteration is no more than a few weeks off, and

the team should have a reasonable level of insight into the work, especially after

discussing during the iteration planning meeting. This allows them to credibly

estimate the tasks of an iteration in hours. The user stories that comprise a release

each represent multiple tasks, are more vague, and less understood so they

must be estimated in more abstract units such as story points or ideal days.

The primary purpose of iteration planning is to refine suppositions made in

the more coarse-grained release plan. The release plan is usually intentionally

vague about the specific order in which user stories will be worked on. Additionally,

at the time of iteration planning the team knows more than when the release

plan was last updated. Planning the iteration as it begins allows the team to

make use of their recently acquired knowledge. In this way, agile planning

becomes a two-stage process. The first stage is the release plan, with its rough

edges and general uncertainties. The second stage is the iteration plan. An iteration

plan still has some rough edges and continues to be uncertain. However, because

it is created concurrent with the start of a new iteration, an iteration plan

is more detailed than a release plan.

Creating the iteration plan leads a team into discussions about both product

design and software design. Product design discussions, for example, may be

around topics such as the best combination of stories for optimizing value, interpretation of feedback from showing working software to customers, or the extentto which a desired feature should be implemented (that is, will 20% of the featureand effort deliver 80% of the value?). Software design discussions may, for

example, involve the appropriate architectural tier in which to implement a new

feature, which technologies should be used, whether existing code can be reused,

and so on. As a result of these discussions the team comes to a better understanding

of what should and will be built, and they also create a list of the

tasks needed to achieve their goal for the iteration.

***Velocity-Driven Iteration Planning***

At a broad level, there are two ways of planning an iteration, which I refer to as

*velocity-driven* and *commitment-driven*. Different teams use different approaches,

and each can be successful. Additionally, the two general approaches

can be combined to varying degrees. In this section, we’ll consider velocity driven

iteration planning; in the next, we’ll focus on commitment-driven iteration

planning.

The steps involved in velocity-driven iteration planning, First, the team collaboratively adjusts priorities. They may have learned something in the preceding iteration that alters their priorities. Next, they identify the target velocity for the coming iteration. The team then selects an iteration goal, which is a general description of what they wish to accomplish during the coming iteration. After selecting an iteration goal, the team selects the top-priority user stories that support that goal. As many stories are selected as necessary for the sum of their ideal-day or story-point estimates to equal the target velocity. Finally, each selected story is split into tasks, and each task is estimated.

***Commitment-Driven Iteration Planning***

A commitment-driven approach is an alternative way to plan an iteration. Commitment-driven iteration planning involves many of the same steps as velocity driven iteration planning. However, rather than creating an iteration plan that

uses the yesterday’s weather idea to determine how many story points or ideal

days should be planned into the current iteration, the team is asked to add

stories to the iteration one by one until they can commit to completing no more.

The first steps—adjusting priorities and identifying an iteration goal—are

the same as in the velocity-driven approach. The next step, selecting a story to

add to the iteration, is different. The product owner and team still select the

highest-priority story that supports the iteration goal. However, in commitment-

driven iteration planning, stories are selected and decomposed into tasks,

and the tasks estimated one story at a time. This is different from the velocitydriven

approach, in which a set of stories whose estimates equaled the estimated

velocity were selected.

Stories are selected one at a time because after each story is split into tasks

and the tasks estimated, the team decides whether or not they can commit to delivering that story during the iteration.

Before committing to the work of an iteration, the team needs to look at the

tasks and get a feel for whether they represent an appropriate distribution of

work based on the various skills within the team. Is the Java programmer likely

to be overloaded, while the HTML programmer has nothing to do this iteration?

Are the selected user stories easy to program but time-consuming or difficult to

test, thereby overloading the tester? Do the stories selected each need analysis

and user interaction design before coding can begin?

A team in a situation like this should first try to find ways to better share

work. Can the HTML programmer in this example help the tester? Can someone

other than the user interaction designer do that work? If not, can we leave out of

this iteration some stories that need user interaction design, and can we bring in

some other stories that do not? The key is that everyone on the team is accountable

for contributing whatever is within their capabilities, regardless of whether

it is their specialty.

**Maintenance and the Commitment**

In addition to making progress on a project, many teams are responsible for support

and maintenance of another system. It may be a prior version of the product

they are working on, or it may be an unrelated system. When a team makes a

commitment to complete a set of stories during an iteration, they need to do so

with their maintenance and support load in mind. I am not referring to general

bug fixes that can be prioritized in advance. Those should go through the regular

iteration planning prioritization process. By maintenance and support activities,

I mean those unpredictable but required parts of many teams’ lives—supporting

a production website or database, taking support calls from key customers or

first-tier technical support, and so on.

# Selecting an Iteration Length

The majority of the agile processes, and teams using them, have settled on iteration

lengths of two to four weeks. Some teams use longer iterations but two to

four weeks is an accepted standard for most teams. There’s no one magic iteration

duration that is right for all teams under all circumstances. The right

length for a team on one project may not be the right length for that same team

on a different project.

***Factors in Selecting an Iteration Length***

Your selection of iteration length should be guided by the following factors:

◆The length of the release being worked on

◆The amount of uncertainty

◆The ease of getting feedback

◆How long priorities can remain unchanged

◆Willingness to go without outside feedback

◆The overhead of iterating

◆How soon a feeling of urgency is established

There is no predetermined relative importance to these factors. The importance

of each is entirely dependent upon the context of the project.

***The Overall Length of the Release***

Short projects benefit from short iterations. The length of a project’s iterations

determines

◆How often the software can be shown (in potentially shippable form) to users

and customers. Yes, of course, the software can be shown in miditeration

form to these audiences, but the software usually is of potentially shippable

quality only at the end of an iteration.

◆How often progress can be measured. It’s possible to get a sense of a team’s

rate of progress during an iteration, but only at the end of an iteration can

we truly measure how much work has been truly completed.

◆How often the product owner and team can refine their course, because priorities

and plans are adjusted between iterations.

If a team is working toward a release that is perhaps only three months away,

one-month iterations will give them only two opportunities to gather end-of-iteration

feedback, measure progress, and adjust course. In most cases, this will be

insufficient.

My general rule of thumb is that any project will benefit from having at least

four or five such opportunities. This means that if the overall project duration

will be four or more months, it might be worth considering monthly or fourweek

iterations. If the overall release will, however, be shorter, the project will

benefit from proportionally shorter iterations.

***The Amount of Uncertainty***

Uncertainty comes in multiple forms. There is often uncertainty about exactly

what the customer or users need, what the velocity of the team will be, and about

technical aspects of the project. The more uncertainty of any type there is, the

shorter the iterations should be. When there is a great deal of uncertainty about

the work to be done or the product to be built, short iterations allow more frequent

opportunites for the team to measure its progress through its velocity and

more opportunities to get feedback from stakeholders, customers, and users.

***The Ease of Getting Feedback***

Iteration length should be chosen to maximize the amount, frequency, and timeliness of feedback to the whole team. Depending on the environment, this may

mean longer or shorter iterations. In some organizations, it is extremely easy to

get informal feedback from internal stakeholders or users throughout an iteration

but extemely difficult to get these same individuals to participate in a scheduled

end-of-iteration review meeting. Other organizations have the opposite

problem; it is difficult to get feedback on a day-to-day basis, but stakeholders, users,

and others will attend a scheduled, formal review meeting (especially if food

is provided).

Choose your iteration length to maximize the value of the feedback that can

be received from those inside and outside the organization.

***How Long Priorities Can Remain Unchanged***

Once a development team commits to completing a specific set of features in an

iteration, it is important that they not be redirected from that goal. It is, therefore,

important that the product owner not change priorities during the iteration

and that she help protect the team from others who may attempt to change

priorities. Because of this, the length of time that priorities can go unchanged is

a factor in selecting the iteration length.

***Willingness to Go without Outside Feedback***

Even with a well-intentioned and highly communicative team, it is possible that

the results of an iteration could be found worthless when shown to the broader

organization or external users at the conclusion of the iteration. This may happen

if the developers misunderstand the product owner (and don’t communicate

often enough during the iteration). It could also happen if the product owner

misunderstands the needs of the market or users. The loss is almost never complete

as long as we learn something from it. However, the less often a team receives

outside feedback, the more likely we are to go astray and the greater the

loss will be when that happens.

***The Overhead of Iterating***

There are costs associated with each iteration. For example, each iteration must

be fully regression tested. If this is costly (usually in terms of time), the team

may prefer longer, four-week iterations. Naturally, one of the goals of a successful

agile team is to reduce (or nearly eliminate) the overhead associated with

each iteration. But especially during a team’s early iterations, this cost can be

significant and will influence the decision about the best iteration length.

***How Soon a Feeling of Urgency Is Established***

Colleague NielsMalotaux (2004) points out that “As long as the end date of a

project is far in the future, we don’t feel any pressure and work leisurely. When

the pressure of the finish date becomes tangible, we start working harder.” Even

with four-week iterations the end date is never very far in the future. But it is

sufficiently far away that many teams will feel tangibly less stress during their

first week than during the fourth and final week of an iteration.

The solution to this, of course, is to select an iteration length that evens out

the pressure the team feels. The point is not to put the team under more

pressure (“You *will* deliver today!”). Rather, it is to take the total amount of stress

they’d normally feel and distribute it more evenly across a suitably long

iteration.

***Making a Decision***

One of the main goals in selecting an iteration length is finding one that encourages

everyone to work at a consistent pace throughout the iteration. If the duration

is too long, there is a natural tendency to relax a bit at the start of the

iteration, which leads to panic and longer hours at the end of the iteration.

Strive to find an interation duration that smooths out these variations.

Having experimented with a variety of iteration lengths, my general preference

is two weeks. One-week iterations (or anything shorter) can be very hectic

and stressful. The next deadline is never more than four days away. Extremely

short iterations leave no time for recovery if a team member is out sick or if

***Stick with It to Achieve a Steady Rhythm***

Whatever duration you choose, you are better off choosing a duration

and sticking with it rather than changing it frequently. Teams fall into a

natural rhythm when using an unchanging iteration duration. When I

started doing agile development using an early variation of Scrum

(Takeuchi and Nonaka 1986; DeGrace and Stahl 1990), my teams used to

select the duration of each iteration based on the amount of work we

were bringing into that iteration. A two-week iteration could be followed

by a six-week iteration, which could be followed by a four-week

iteration, and so on. Through experimentation on many projects, I have

since learned that teams are far better off sizing the work to the length

of the iteration (rather than sizing the iteration to the work).

A regular iteration rhythm acts like a heartbeat for the project. Colleague

Simon Baker, an agile coach with think-box ltd., describes it by

saying that “Like a heart beats with a regularity that keeps the body going,

a fixed iteration duration provides a constant which helps establish a

development (and delivery) rhythm. Rhythm in my experience is a significant

factor that helps achieve a sustained pace” (2004).

# Estimating Velocity

One of the challenges of planning a release is estimating the velocity of the team.

You have the following three options:

◆Use historical values.

◆Run an iteration.

◆Make a forecast.

There are occasions when each of these approaches is appropriate. However,

regardless of which approach you are using, if you need to estimate velocity you

should consider expressing the estimate as a range. Suppose you estimate that

velocity for a given team on a given project will be 20 ideal days per iteration.

You have a very limited chance of being correct. Velocity may be 21, or 19, or

maybe even 20.0001. So instead of saying velocity will be 20, give your estimate

as a range, saying perhaps instead that you estimate velocity will be between 15

and 24.

***Use Historical Values***

Historical values are great—if you have them. The problem with historical values

is that they’re of the greatest value when very little has changed between the

old project and team and the new project and team. Any personnel or significant

technology changes will reduce the usefulness of historical measures of velocity.

Before using them, ask yourself questions like these:

◆Is the technology the same?

◆Is the domain the same?

◆Is the team the same?

◆Is the product owner the same?

◆Are the tools the same?

◆Is the working environment the same?

◆Were the estimates made by the same people?

The answer to each question is often yes when the team is moving onto a

new release of a product they just worked on. In that case, using the team’s historical

values is entirely appropriate. Even though velocity in a situation like this

is relatively stable, you should still consider expressing it as a range. You could

create a range by simply adding and subtracting a few points to the average or by

looking at the team’s best and worst iterations over the past two or three

months.

However, if the answer to any of the preceding questions is no, you may want

to think twice about using historical velocities. Or you may want to use historical

velocities but put a larger range around them to reflect the inherent uncertainty

in the estimate.

***Run an Iteration***

An ideal way to forecast velocity is to run an iteration (or two or three) and then

estimate velocity from the *observed velocity* during the one to three iterations.

Because the best way to predict velocity is to observe velocity, this should always

be your default approach.

If a team can run three or more iterations before being giving an estimate of

velocity, they have a couple of additional options for determining a range. First

and easiest, they can simply use the range of observed values. Suppose the team

has completed three iterations and had velocities of 12, 15, and 16. They could

express velocity as likely to be within the range 12 to 16.

***Make a Forecast***

There are times when we don’t have historicals, and it is just not feasible to run

a few iterations to observe velocity. Suppose the estimate is for a project that

won’t start for twelve months. Or suppose the project may start soon, but only

once a client signs a contract for the work. There are two key differences in cases

like this. First, you want to minimize the expenditure on the project so you

won’t actually start running iterations on a project that may not happen or that

is too far in the future. Second, any estimate of velocity on these projects must

reflect a high degree of uncertainty.

In cases like these, we need to forecast velocity. Forecasting velocity is rarely

your first option, but it’s an important option and one you should have in your

bag of tricks. The best way to forecast velocity involves expanding user stories

into their constituent tasks, estimating those tasks (as we do when planning an

iteration), seeing how much work fits into an iteration, and then calculating the

velocity that would be achieved if that work were finished in an iteration. This

involves the following steps:

1. Estimate the number of hours that each person will be available to work on

the project each day.

2. Determine the total number of hours that will be spent on the project during

the iteration.

3. Arbitrarily and somewhat randomly select stories, and expand them into

their constituent tasks. Repeat until you have identified enough tasks to fill

the number of hours in the iteration.

4. Convert the velocity determined in the preceding step into a range.

# Buffering Plans for Uncertainty

***Feature Buffers***

We tell our customers, “We’ll get you all of the functionality in this pile and ideally some of the functionality in that pile.” Creating a feature buffer is simple to do on an agile project. First, the customer selects all of the absolutely mandatory work. The estimates for that work are summed. This represents the minimum that can be released. The customer then selects another 25% to 40% more work, selecting

toward the higher end of the range for projects with more uncertainty or less

tolerance for schedule risk. The estimates for this work are added to the original

estimate, resulting in a total estimate for the project. The project is then planned

as normal for delivery of the *entire* set of functionality; however, some amount of

the work is optional and will be included only if time permits. The optional work

is developed last, only after the mandatory work is complete.

To see how this works, assume that the product owner identifies 100 story

points as mandatory. Each story selected is required to release a product that will

be favourably accepted by the market. The product owner then selects an additional

30% more work, identifying user stories worth an additional 30 story

points. These are added as optional work to the project. The total project is now

expected to be 130 story points.

This feature buffering process is consistent with that used in the agile process,

DSDM (Dynamic Systems Development Method). On DSDM projects, requirements

are sorted into four categories: Must Have, Should Have, Could

Have, and Won’t Have. DSDM refers to this sorting as the MoSCoW rules. No

more than 70% of the planned effort for a project can be targeted at Must Have

requirements. In this way, DSDM projects create a feature buffer equivalent to

30% of the duration of the project.

***Schedule Buffers***

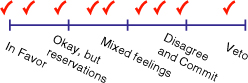
Schedule buffer is adding some time buffer in release plan, this can observe the uncertain events which may happen during the executing of the project.

### Participatory Decision Making

The objective of participatory decision making is to provide the project community with specific practices to frame, analyze, and make the myriad decisions that arise during a project. The lack of adequate decision-making processes in organizations is evident in a couple of quotes from clients I’ve worked with over the years.

“That one decision-gradient diagram [[**Figure 9-9**](javascript:moveTo('ch09fig08');)] was the most important piece of the two-day consulting session,” said a product development VP client recently stated. The gradient kept them from focusing in too early on binary yes-no decisions and led to better discussions.

##### Figure 9-9. Decision Gradients Tell a Story



“It’s difficult to speed up development when management takes weeks to make key decisions,” laments an Irish development manager whose company executives are in Silicon Valley.

“Our project managers are like a herd of deer standing on the highway with a tractor-trailer truck bearing down on them,” says one team member. “They can’t figure out which way to jump, but if they don’t decide soon, we’re going to get run over.”

As I trek around the world of product development and project management I’m continually amazed at how little organizations think about their decision-making processes. Many of them put time and energy into processes such as time recording and virtually ignore decision making. However, in a fast-paced agile project, decision making—like other activities—must be done quickly and effectively. Slow decision making, revisiting decisions again and again, over-analyzing decisions, and poor participation in the decision-making process will doom a project, as poor decisions cascade into a flood of additional decisions.

However, decision making can improve, and it can be participatory, as the GE jet engine plant in Durham, North Carolina, proves. “At GE/Durham, every decision is either an A decision, or a B decision, or a C decision,” writes Charles Fishman ([**1999**](http://my.safaribooksonline.com/9780321659200/bib01#bib01_038)) in an article in *Fast Company*. “An A decision is one that the plant manager makes herself, without consulting anyone. B decisions are also made by the plant manager, but with input from the people affected. C decisions—which make up the most common type—are made by consensus, by the people directly involved, with plenty of discussion.” Using this system, the plant manager only makes 10–12 “A” decisions in a year and spends significant time explaining those to the staff.

The article goes on to address the very crucial issue that arises in discussing self-organizing teams:

*What is the role of a plant manager in a place that manages itself? If the plant needs a manager like Sims to make just 10 decisions a year, what does she do with the bulk of her time?*

*She does the kinds of things that most managers talk about a lot but that they actually spend very little time on. At the operational level, her job is to keep everyone’s attention focused on the goals of the plant: Make perfect engines, quickly, cheaply, safely. Strategically, the plant manager’s job is to make sure that the plant as a whole is making smart decisions about talent, about time, and about opportunities for growth (*[***Fishman 1999***](http://my.safaribooksonline.com/9780321659200/bib01#bib01_038)*).*

These management roles are analogous to the coaching and team development practices discussed earlier in this chapter. Oh, and the GE/Durham plant is a model of effectiveness and efficiency.

Even authors Carl Larson and Frank LaFasto ([**1989**](http://my.safaribooksonline.com/9780321659200/bib01#bib01_069)), who at least recognize the importance of decision making, don’t delve into how to actually improve the process. They do, however, observe, “The third set of leadership principles, and we believe the most important, clearly focus attention on the creation of a supportive decision-making climate.” They also point out that achieving a goal requires change, that change requires decisions be made, and that making decisions involves risk. Without a safe environment in which team members can take risks, effective decision making will be stymied.

At its core, collaboration is about decision making. We can talk, share ideas, and debate issues, but in the final analysis decisions must be made—about design, about features, about tradeoffs, about a host of issues. Collaboration isn’t talking, it’s delivering, and delivering means making decisions. A participatory decision-making process can be useful for larger groups or for two individuals—the process and the issues are the same. Furthermore, although the steps of the decision-making process may proceed less formally between two individuals than they would in a group, the emphasis on sustainable, win-win decisions based on debate and full participation remains key.

|  |
| --- |
| *The biggest complaint from people isn’t that they lack a vote in decisions; it’s that they don’t even get heard on decisions that affect them.* |

One definitional point is critical: Participatory decision making (everyone participates) is different from consensus decision making (everyone votes in favor). The latter is too slow and isn’t appropriate in many project situations where the divergence of ideas and opinions would limit the effectiveness of the decision-making process. The critical element isn’t consensus but sustainability: Will the team consistently implement decisions that are made? Participation leads to sustainability efficiently and effectively. In consensus decision making everyone votes on the decision, and no decision can be implemented without a unanimous vote. In participatory decision making, team members participate in the decision process, and the decision is made by a preponderance of the vote.

No doubt decision making is hard, but it is made harder than necessary by poor practices, and there are practices that can assist teams in making better, implementable decisions. Three elements compose a decision process: decision framing, decision making, and decision retrospection. Framing establishes “who” gets involved in the process, whereas decision making establishes “how” the “whos” go about making a decision. Retrospection provides feedback into the decision-making process. As with other APM practices, decision-making practices must be implemented with the Simplify principle in mind; otherwise the team will end up with just another unwieldy set of procedures and forms.

#### Decision Framing

The often overused term “empowerment” means to delegate decision-making authority to lower levels of organizations by changing who *makes*decisions. Decision framing focuses on who gets *involved* in the decision process. Managers who make decisions without input from subordinates and peers make poor decisions. Engineers who make decisions without input from managers and peers make poor decisions. Who makes the decision is less important than getting the right people involved in the decision process.

However, framing involves more than “who”; it also means considering the values and principles that participants share. Without shared values and principles, teams will have difficulty reaching sustainable decisions. The agile values and principles articulated in earlier chapters, whether adopted verbatim or adapted for a specific organization, are vital to decision making. There is a hierarchy of decision-making criteria—values and principles, product vision, project objectives, and tradeoff matrix, as well as detailed criteria such as design parameters (e.g., usability). Teams that fail to agree on principles—explicitly—will have problems making sustainable decisions as projects progress.

The first task in framing decisions involves identifying types of decisions that need to be made. For example, in an agile project, re-planning occurs at the end of each iteration or wave. Re-planning often involves making tradeoff decisions—schedule versus cost versus stories. Projects should include a decision framework for asking the basic question, “Can we release this product now?”

For each decision type, typical framing questions are

* Who is impacted by the decision?
* Who needs to provide input to the decision?
* Who should be involved in the discussions about the decision?
* Who should make the decision (the product manager, the project leader, the team, the project leader with the team, etc.)?
* What decision criteria should be used?
* How and to whom should the decision results be communicated?
* Who should review the decision?

The answers to these questions will involve several overlapping groups of individuals. For example, a wide group of people may be impacted by the decision, but only selected individuals from those groups may be contacted for input. Everyone who provides input to a decision may not be involved in the discussions about those decisions. Many decisions bore team members, and thus they don’t want to be involved, whereas they do want to be heard on other decisions. Sorting out the various involvements should be the result of careful thinking by the team members and the project and product leaders.

Team members often feel isolated from decision processes, not knowing when, why, or how decisions get made. Making decisions is only part of implementing them. Rapid, effective implementation requires a participatory process that involves the right people, with the relevant information, gathered together at the right time.

Many companies and project leaders spend far more time on development processes than decision making, which brings to mind a race car engine running on increasingly viscous sludge. Both will grind to a halt. Framing is the first step in getting the sludge out of your decision-making process.

#### Decision Making

In many organizations, decision making is viewed as a win-lose proposition. Participants in the process have a preconceived view of the right answer, and their approach is to argue as loudly as possible until the opposition gives up. Collaborative decision making focuses on win-win—or “both/and” rather than “either/or.” Win-win decision making focuses on mutual understanding rather than loud posturing. This shouldn’t imply a lack of heated discussion, but a discussion focused on trying to understand the underlying issues rather than debating preordained positions. Participatory decision making can be contentious but civil, based on mutual trust and respect. It moves teams beyond compromise to reconceiving. Participatory decision making is a process of reconceiving a solution to a problem based on information from all team members. Compromise implies giving up one idea for another (and often results in inferior decisions); reconceiving implies a joining of ideas.

Collaboration is hard. In seemingly interminable meetings, team members often flounder in the “groan zone,” author Sam Kaner’s ([**1996**](http://my.safaribooksonline.com/9780321659200/bib01#bib01_062)) wonderful term for the time period in which meeting participants struggle to understand each other. Although many people have heard of the famous team progression process “forming, storming, norming, performing” (or, more aptly at times, forming, storming, thrashing, crashing), Kaner’s model consists of the divergent zone, the groan zone, and finally the converging zone.

Any decision-making process must be judged against two objectives. First, does the process result in the best choice given the circumstances in which the decision was made? Second, was the decision implemented? As many project leaders have found out the hard way, making and implementing decisions are two different things. How many times have you encountered decisions made within the confines of a conference room that fall completely apart when the participants walk out the door? Anyone can make a decision, but effective managers grasp that implementation requires people to understand and support the decision.

A participatory decision-making process has three components: principles, framework, and practices. The fundamental principles have just been alluded to: viewing the process as a win-win process and treating all participants with respect. All collaborative practices are based on trust and respect, or perhaps more precisely, on building trust and respect.[**[8]**](javascript:moveTo('ch09fn08');) Kaner’s diverge-groan-converge model provides a framework for building these positive relationship qualities. In the diverge-groan-converge framework, the transition from the divergent zone to the convergent zone explains how team members move from having individual opinions to having a unified position. At first, people’s ideas diverge. Even though each person wants to contribute to success and to making a quick decision, each wants to voice his or her own opinion. Everyone has a different perspective or a different experience, which brings needed diversity to the decision process but not much agreement. This groaning period takes time—time for people to speak and hear, time for them to build trust. A little extra time (it’s not really extra, but it seems as if it is) taken on decision making in the early stages of a project will significantly reduce time as the project continues.

[8] The idea of building trust may seem counter to the earlier statement that managers either trust or don’t trust. However, a manager can believe in trusting team members but also understand that the level of trust must be maintained through actions. People are predisposed to trust or not trust, but they still want proof in support of that predisposition.

Convergence occurs as the individual ideas are integrated into a whole solution. Convergence, done correctly, does not necessarily mean that everyone is in complete agreement, but that everyone has participated and will support the final decision. The goal is not merely agreement but “sustainable agreement”—a unified position.

The transition period between divergence and convergence, the groan zone, is the time during which team members groan and complain. In the divergent zone, most group members voice their opinions to make sure the group hears their ideas. Much of this time initially could be considered presentation, during which members are primarily trying to sell their own ideas. Participants then begin to groan because they are trying to understand one another, and understanding requires thought. It is relatively easy to take a position and argue for it. It is much more difficult to attempt to understand why other participants hold their opinions. Participants want to ask questions, they want to be heard, they want to—participate. The groan zone provides a perfect description of what happens in most teams; it is a turbulent zone where innovative, creative results are generated.

One of the best tools for testing how the decision-making process is proceeding, and for arriving at the decision itself, is a decision gradient that replaces the familiar yes-no voting. A decision gradient, as shown in[**Figure 9-9**](javascript:moveTo('ch09fig08');), gives participants more options: in favor; OK, but with reservations; mixed feelings; disagree and commit (to implement the decision); veto. When all participants plot their responses on a line with these gradations, the entire team gets to view its collective opinion. The team can then address issues like trying to understand the person who vetoed the decision or trying to understand why so many people are clustered around “mixed feelings.” Voting—or actually, the discussions about why the voting went one way or another—leads to a deeper understanding of the issues and eventually to another vote. Decision gradients make for better discussion and more effective, sustainable decisions.[**[9]**](javascript:moveTo('ch09fn09');)

[9] See Kaner (1996) for more information on decision gradients.

When some person (manager, technical lead) is designated as the decision maker, a preponderance of agreement among participating team members is helpful but not essential. But when a team as a whole is the decision maker, what actions craft a sustainable decision? In many people’s minds, consensus has come to mean “unanimous,” the connotation used earlier in this section. But consensus has another definition that corresponds to the idea of a preponderance of agreement among participants. Intel is one company known for its attention to decision making. Intel emphasizes decision-making training for employees, and the company focuses on decision framing and making on a regular basis. Intel has an engrained decision-making culture in which the phrase “disagree and commit” is often used. It means that someone might disagree with a decision, but he will commit himself to its implementation.

This non-unanimous type of consensus is built on the following premises:

* Everyone has had an opportunity to have his or her ideas heard and discussed.
* Consensus does not imply unanimous agreement, but it does mean that people understand the decision rationale.
* No one has been silenced due to fear or intimidation.
* The preponderance of the group votes in favor of the decision (or in favor with some reservations).
* No one vetoes the decision (instead, they disagree and commit).

Decisions thus reached are sustainable in ways that lead to team cohesion and positive outcomes. Arbitrary and capricious decisions, those imposed by force of will or organizational power, have the opposite effect.

An additional benefit to a participatory process such as the one just described is that as mutual understanding of the context (including the decision criteria) increases, the time required to make similar decisions decreases rapidly. For example, a defect triage team that develops a shared understanding of the relevant quality factors involved in reaching decisions will speed up its decisions over time. Conversely, teams that do not take additional time in the beginning to fully understand each other’s perspective on some issue (e.g., quality) will constantly argue the same points meeting after meeting, wasting irreplaceable project time.

Different kinds of decisions require different decision criteria. Coin flipping works for what time to go to lunch. The tradeoff matrix steers constraint decisions, just as performance criteria might steer technical decisions. Release decisions might use agile triangle quality criteria. For each kind of decision, one of the discussion topics should be the criteria to be used in making that type of decision. You may even need to go through a decision-making process to arrive at the criteria for making a decision.

#### Decision Retrospection

End-of-iteration, wave, and project retrospectives should include time to review decisions as part of reviewing the team’s performance (discussed in the next chapter). However, if project retrospectives are difficult to do in general, then decision retrospectives are even more so, because finding whom to blame often seems more important than learning. But how do we get better at decisions unless we understand which ones worked out well and which ones didn’t?

Still, few organizations want to examine decisions in any depth, which probably corresponds to the general lack of interest in decision making. Was an error-prone product released? Why? What were the decisions that led up to the release decision? Maybe the decision was actually a “good” one from a market perspective. If so, then the development staff needs to understand the nature of the decision, why it was made, and who was involved in the decision. Maybe the decision was based on market timing information, but the decision makers didn’t listen to the developers, and the actual release was a disaster. If the disaster isn’t analysed, if the decision tradeoff of product stability versus market need analysis isn’t revisited, then nothing will be learned and similar mistakes will be made in the future. On the other hand, a decision may be perceived as incorrect, but further analysis shows that it was actually the correct one given the circumstances. In this case, lack of analysis might keep us from making the same “correct” decision in the future.

Participatory decision making may spell the difference between success and failure on agile projects. Framing decisions, developing a collaborative decision-making process, and conducting decision retrospectives to learn from both success and failure are components of this practice.

#### Leadership and Decision Making

A good project leader has to be a visionary, a teacher, a motivator, a facilitator, and other things, but she must also be a decision maker. The same is true of lead engineers for technical issues. So the question becomes, at what point does a manager’s decision making damage self-organization? First, when the team loses respect for the leader. But what causes loss of respect? The answer: when the manager begins making unilateral or arbitrary decisions. The more unilateral decisions, the less participation from the team, and the less likely the decisions are to be effectively implemented.

Every team and situation are different, so there isn’t a quantitative answer to the question of how many unilateral decisions are too many. However, even though presenting absolute numbers risks misinterpretation, I think the following guidelines may help define appropriate “levels” of management decision making that will continue to foster self-organization. For both project leaders and lead engineers, this rough guide is one unilateral decision every month or two, three to four decisions per month with team involvement, and then delegate the hundreds of other decisions to the team. In practice, few good managers make completely unilateral decisions—they normally talk issues over with at least key members of their team. But occasionally there is a need to get things moving by making a unilateral decision. In that same vein, it is appropriate for project leaders and lead engineers to make certain decisions with team participation, but if they are making more than three or four of these decisions per month, even with team involvement, they are probably too absorbed in the details.

|  |
| --- |
| *Leaders at any level—executive, functional manager, project leader, iteration manager, technical lead—who are making more than three or four unilateral decisions a month are adversely impacting their team’s ability to self-organize.* |

Another issue related to management decision making is the leader’s job of absorbing ambiguity. In fast-moving product development efforts in which key decisions must be made quickly, consensus (unanimous) decision making fails, but even participatory decision making can get mired in discussion and debate. Many product development issues, both technical and administrative, may be fuzzy and ambiguous. In these cases, after participation has evolved to a certain point, managers have to be willing to make final decisions. “Well, the information available to us isn’t crystal clear, but to move forward with the project, we’ll go in this direction.”

Good leaders have earned the credibility to make these decisions. The technical staff respect the leader’s judgment (based on previous actions taken), participate in the analysis and debate process, and willingly accept the decision to move on. The leader has absorbed the ambiguity of the situation, whereas leaving the decision to consensus would have bogged the project down in interminable debate. Good leaders know when to step in and take charge and when to encourage the team to take charge. They also know when to dig into why team decision making isn’t working as it should.

#### Set- and Delay-Based Decision Making

If we want to build adaptive teams and products, not only do we need a participatory decision-making process, but we also need to look at criteria for decision making that encourage experimentation. Point-based engineering dominates current product development. Point-based engineering views design as a series of decisions in which each decision narrows the options for further decisions, and the product progresses in a steady fashion from a gleam in the marketer’s eye to a final product.[**[10]**](javascript:moveTo('ch09fn10');)

[10] Using Real Options is another practice making headway in the agile toolbox. See, for example, “Real Options Underlie Agile Practices,” Chris Matts and Olav Maassen ([**www.infoq.com**](http://www.infoq.com/)).

Toyota upset this apple cart, at least as it pertains to the automotive industry’s design process. Toyota’s approach, set-based concurrent engineering (SBCE), provides a new insight into product design. SBCE operates on two fundamental concepts: postpone design decisions as late as possible and maintain “sets” of design solutions throughout the majority of the design process.

“SBCE assumes that reasoning and communicating about sets of ideas leads to more robust, optimized systems and greater overall efficiency than working with one idea at a time, even though the individual steps may look inefficient,” write Durward K. Sobek, Allen C. Ward, and Jeffrey K. Liker ([**1999**](http://my.safaribooksonline.com/9780321659200/bib01#bib01_098)). Rather than converge on a design “answer,” Toyota’s engineers maintain sets of designs. For a particular car project, they might maintain six alternative solutions that include prototypes and mock-ups for the exhaust system design.

Unlike point requirements, set-based requirements focus on ranges or minimum constraints. So the body design group would impose a criteria “range” on the exhaust system, keeping the tolerances as broad as possible in the beginning and narrowing them over time as the car approaches manufacturing. As the body design and exhaust system designs evolve, engineers are more likely to balance subsystem optimization with overall vehicle optimization. In a point-based approach, each subsystem team has a tendency to quickly create optimized designs for its particular subsystem that are often at odds with overall system design effectiveness.

Toyota’s slow narrowing of options extends even to die making. Rather than specify precise part fit, designers specify wider tolerances. The die makers themselves create the parts, see how they actually fit together, and then send the precise measurements back to the design groups to finalize the detail CAD drawings.

Engineers, whether of automobiles or computers, tend toward point-based solutions—they analyze the problem, review the constraints, and then design “the” solution. But there are always multiple design options, and the larger the product or product family, the more likely that early design decisions will lock the team into suboptimal solutions. Maintaining multiple sets of solutions and delaying final design decisions, even though it may appear to be inefficient, may in fact be faster and more efficient in the long run. As Sobek and his coauthors observe, “Toyota considers a broader range of possible designs and delays certain decisions longer than other automotive companies do, yet has what may be the fastest and most efficient vehicle development cycles in the industry” (**[Sobek 1999](http://my.safaribooksonline.com/9780321659200/bib01" \l "bib01_036)**).