CHAPTER 8

Planning

Today I talked to a friend who wanted to know how I organized software projects. His team has grown rapidly, and their attempts to create and manage detailed plans are spiraling out of control. "It just doesn't scale," he sighed.

The larger your project becomes, the harder it is to plan everything in advance. The more chaotic your environment, the more likely it is that your plans will be thrown off by some unexpected event. Yet in this chaos lies opportunity.

Rather than trying to plan for every eventuality, embrace the possibilities that change brings you. This attitude is very different from facing change with clenched jaws and white knuckles. In this state of mind, we welcome surprise. We marvel at the power we have to identify and take advantage of new opportunities. The open horizon stretches before us. We know in which direction we need to travel, and we have the flexibility in our plan to choose the best way to get there. We'll know it when we find it.

This approach may sound like it's out of control. It would be, except for eight practices that allow you to control the chaos of endless possibility:

- *Vision* reveals where the project is going and why it's going there.
- *Release Planning* provides a roadmap for reaching your destination.
- *The Planning Game* combines the expertise of the whole team to create achievable plans.
- *Risk Management* allows the team to make and meet long-term commitments.
- *Iteration Planning* provides structure to the team's daily activities.
- *Slack* allows the team to reliably deliver results every iteration.
- *Stories* form the line items in the team's plan.
- *Estimating* enables the team to predict how long its work will take.

"PLANNING" MINI-ÉTUDE

The purpose of this étude is to discuss the effect of change on value. If you're new to agile development, you may use it to better understand the purpose of your project and how each story adds value. If you're an experienced agile practitioner, review Chapter 14 and use this étude to help you adapt your plans.

Conduct this étude for a timeboxed half-hour every day for as long as it is useful. Expect to feel rushed by the deadline at first. If the étude becomes stale, discuss how you can change it to make it interesting again.

You will need three different colors of index cards, an empty table or whiteboard, and writing implements.

Step 1. Start by forming pairs. Try for heterogenous pairs.

Step 2. *(Timebox this step to five minutes.)* Within your pair, choose an unimplemented story and discuss why it's valuable to the customer. Write the primary reason on an index card.

Step 3. *(Timebox this step to five minutes.)* Within your pair, brainstorm ideas that would provide even more value, whether by changing the existing story or by creating a new story. Write one idea on an index card of a different color.

Step 4. *(Timebox this step to five minutes.)* Still within your pair, devise an experiment that would prove whether your new idea would work in practice—*without actually trying the idea itself*. What can you do to test the concept behind your idea? Write the test on an index card of a third color.

Step 5. *(Timebox this step to 15 minutes.)* Return to the group. Choose three pairs. Give each pair five minutes to lead a discussion of their findings. Post the resulting cards on the table or whiteboard for as long as you perform this étude.

Discussion questions may include:

- What types of value are there in the cards?
- What would be hard to do? What would be easy?
- How well does your work reflect the potential value of the project?
- Which concrete experiments should you conduct in the next iteration?

Vision

Audience
Product Manager, Customers

We know why our work is important and how we'll be successful.

Vision. If there's a more derided word in the corporate vocabulary, I don't know what it is. This word brings to mind bland corporate-speak: "Our vision is to serve customers while maximizing stakeholder value and upholding the family values of our employees." Bleh. Content-free baloney.

Don't worry—that's not what you're going to do.

Product Vision

Before a project is a project, someone in the company has an idea. Suppose it's someone in the Wizzle-Frobitz company.* "Hey!" he says, sitting bolt upright in bed. "We could frobitz the wizzles so much better if we had some software that sorted the wizzles first!"

Maybe it's not quite that dramatic. The point is, projects start out as ideas focused on results. Sell more hardware by bundling better software. Attract bigger customers by scaling more

* Not a real company.

effectively. Open up a new market by offering a new service. The idea is so compelling that it gets funding, and the project begins.

Somewhere in the transition from idea to project, the compelling part—the *vision* of a better future—often gets lost. Details crowd it out. You have to hire programmers, domain experts, and interaction designers. You must create stories, schedule iterations, and report on progress. Hustle, people, hustle!

That's a shame, because nothing matters more than delivering the vision. The goal of the entire project is to frobitz wizzles better. If the details are perfect (the wizzles are sorted with elegance and precision) but the vision is forgotten (the wizzle sorter doesn't work with the frobitzer), then the software will probably fail. Conversely, if you ship something that helps frobitz wizzles better than anything else, does it really matter *how* you did it?

Where Visions Come From

Sometimes the vision for a project strikes as a single, compelling idea. One person gets a bright idea, evangelizes it, and gets approval to pursue it. This person is a *visionary*.

More often, the vision isn't so clear. There are multiple visionaries, each with their own unique idea of what the project should deliver.

Either way, the project needs a single vision. Someone must unify, communicate, and promote the vision to the team and to stakeholders. That someone is the product manager.

Identifying the Vision

Like the children's game of telephone, every step between the visionaries and the product manager reduces the product manager's ability to accurately maintain and effectively promote the vision.

If you only have one visionary, the best approach is to have that visionary act as product manager. This reduces the possibility of any telephone-game confusion. As long as the vision is both worthwhile and achievable, the visionary's day-to-day involvement as product manager greatly improves the project's chances of delivering an impressive product.

If the visionary isn't available to participate fully, as is often the case, someone else must be the product manager. Ask the visionary to recommend a trusted lieutenant or protégé: someone who has regular interaction with the visionary and understands how he thinks.

Frequently, a project will have multiple visionaries. This is particularly common in custom software development. If this is the case on your project, you need to help the visionaries combine their ideas into a single, cohesive vision.

Before you go too far down that road, however, ask yourself whether you actually have multiple projects. Is each vision significantly different? Can you execute them serially, one vision at a time, as separate projects built by the same team on a single codebase? If you can, that may be your best solution.

If you can't tease apart the visions (do try!), you're in for a tough job. In this case, the role of the product manager must change. Rather than being a visionary himself, the product manager *facilitates* discussion between multiple visionaries. The best person for the job is one who

understands the business well, already knows each of the visionaries, and has political savvy and good facilitation skills.

It might be more accurate to call this sort of product manager a *product facilitator* or *customer coach*, but I'll stick with *product manager* for consistency.

Documenting the Vision

After you've worked with visionaries to create a cohesive vision, document it in a vision statement. It's best to do this collaboratively, as doing so will reveal areas of disagreement and confusion. Without a vision statement, it's all too easy to gloss over disagreements and end up with an unsatisfactory product.

Once created, the vision statement will help you maintain and promote the vision. It will act as a vehicle for discussions about the vision and a touchpoint to remind stakeholders why the project is valuable.

Don't forget that the vision statement should be a *living* document: the product manager should review it on a regular basis and make improvements. However, as a fundamental statement of the project's purpose, it may not change much.

How to Create a Vision Statement

The vision statement documents three things: *what* the project should accomplish, *why* it is valuable, and the project's *success criteria*.

The vision statement can be short. I limit mine to a single page. Remember, the vision statement is a clear and simple way of describing why the project deserves to exist. It's not a roadmap; that's the purpose of release planning.

Ally
Release Planning (p. 207)

In the first section—*what* the project should accomplish—describe the problem or opportunity that the project will address, expressed as an end result. Be specific, but not prescriptive. Leave room for the team to work out the details.

Here is a real vision statement describing "Sasquatch," a product developed by two entrepreneurs who started a new company:

> Sasquatch helps teams collaborate over long distance. It enables the high-quality team dynamics that occur when teams gather around a table and use index cards to brainstorm, prioritize, and reflect.
>
> Sasquatch's focus is *collaboration* and *simplicity*. It is not a project management tool, a tracking tool, or a retrospectives tool. Instead, it is a free-form sandbox that can fulfill any of these purposes. Sasquatch assumes that participants are well-meaning and create their own rules. It does not incorporate mechanisms to enforce particular behaviors among participants.
>
> Sasquatch exudes *quality*. Customers find it a joy to use, although they would be hard-pressed to say why. It is full of small touches that make the experience more enjoyable.
>
> Collaboration, simplicity, and quality take precedence over breadth of features. Sasquatch development focuses on polishing existing features to a high gloss before adding new ones.

In the second section, describe *why* the project is valuable:

> Sasquatch is valuable to our customers because long-distance collaboration is so difficult without it. Even with desktop sharing tools, one person becomes the bottleneck for all discussion. Teams used to the power of gathering around a table chafe at these restrictions. Sasquatch gives everyone an opportunity to participate. It makes long-distance collaboration effective and even enjoyable.
>
> Sasquatch is valuable to us because it gives us an opportunity to create an entrepreneurial product. Sasquatch's success will allow us to form our own product company and make a good living doing work that we love.

In the final section, describe the project's *success criteria*: how you will know that the project has succeeded and when you will decide. Choose concrete, clear, and unambiguous targets:

> We will deploy Sasquatch in multiple stages with increasing measures of success at each stage.
>
> In the first stage, we will demonstrate a proof-of-concept at a major Agile event. We will be successful if we attract a positive response from agile experts.
>
> In the second stage, we will make a Sasquatch beta available for free use. We will be successful if at least 10 teams use it on a regular basis for real-world projects within 6 months.
>
> In the third stage, we will convert Sasquatch to a pay service. We will be successful if it grosses at least $1,000 in the first three months.
>
> In the fourth stage, we will rely on Sasquatch for our income. We will be successful if it meets our minimum monthly income requirements within one year of accepting payment.

Promoting the Vision

After creating the vision statement, post it prominently as part of the team's informative workspace. Use the vision to evangelize the project to stakeholders and to explain the priority (or deprioritization) of specific stories.

Ally

Informative Workspace (p. 86)

Be sure to include the visionaries in product decisions. Invite them to release planning sessions. Make sure they see iteration demos, even if that means a private showing. Involve them in discussions with real customers. Solicit their feedback about progress, ask for their help in improving the plan, and give them opportunities to write stories. They can even be an invaluable resource in company politics, as successful visionaries are often senior and influential.

Including your visionaries may be difficult, but make the effort; distance between the team and its visionaries decreases the team's understanding of the product it's building. While the vision statement is necessary and valuable, a visionary's personal passion and excitement for the product communicates far more clearly. If the team interacts with the visionary frequently, they'll understand the product's purpose better and they'll come up with more ideas for increasing value and decreasing cost.

NOTE

When I'm faced with an inaccessible visionary, I don't assume that the problem is insurmountable. Because the participation of the visionaries is so valuable, I take extra steps to include visionaries in any way I can. I don't take organizational structures for granted, and I push to remove barriers.

If the visionaries cannot meet with the team at all, then the product manager will have to go to them to share the plan, get feedback, and conduct private demos. This is the least effective way of involving the visionaries, and you must decide if the product manager understands the vision well enough to act in their stead. Ask your mentor for help making this decision. If you conclude that the product manager doesn't understand the vision well, talk with your executive sponsor about the risks of continuing, and consider that your team may be better off doing something else until the visionaries are available.

Questions

Discussing the vision has led to contentious arguments. Should we stop talking about our vision?

Even if there are big disagreements about the vision, you should still pursue a unified vision. Otherwise, the final product will be just as fragmented and unsatisfactory as the vision is. You may benefit from engaging the services of a professional facilitator to help mediate the discussions.

Our organization already has a template for vision statements. Can we use it?

Certainly. Be sure you cover *what*, *why*, and *success criteria*. Find a way to fit those topics into the template. Keep your vision statement to a single page if you can.

Our visionary finds it very difficult to communicate a cohesive vision. What can we do when it changes?

Rapidly shifting goals tend to be common with entrepreneurial visionaries. It isn't due to lack of vision or consistency; instead, your visionary sees a variety of opportunities and changes direction to match.

If the vision is constantly changing, this may be a sign that what you think of as the vision is just a temporary strategy in a larger, overarching vision. Take your concerns to the visionary and stakeholders and try to identify that larger vision.

If you succeed in discovering the larger vision, adaptive release planning (see "Adapt Your Plans"" later in this chapter) can help you keep up with your visionary. Adaptive planning's emphasis on learning and on taking advantage of opportunities will fit in perfectly with your visionary's entrepreneurial spirit.

Allies

Release Planning (p. 207)
The Planning Game (p. 221)

Your visionary may continue to shift direction more quickly than you can implement her ideas. Wild and unpredictable shifts make it difficult to develop software effectively. The planning game helps; stick with the normal mechanism of scheduling and implementing stories. Your product manager should act as a buffer in this case, protecting the team from rapid shifts and explaining to your visionary what the team can reasonably accomplish.

Can individual iterations and releases have smaller visions?

Of course! This works particularly well with release planning—it can be a great way to help customers choose the priority of stories as they plan their next release.

I'm less fond of visions for iteration planning, just because iterations are so short and simple that the extra effort usually isn't worth it.

Results

When your project has a clear and compelling vision, prioritizing stories is easy. You can easily see which stories to keep and which to leave out. Programmers contribute to planning discussions by suggesting ways to maximize value while minimizing development cost. Your release plan incorporates small releases that deliver value.

When the visionary promotes the vision well, everyone understands why the project is important to the company. Team members experience higher morale, stakeholders trust the team more, and the organization supports the project.

Contraindications

Always pursue a unified vision for your projects, even at the risk of discovering there isn't one. If the project isn't worth doing, it's better to cancel it now than after spending a lot of money.

Alternatives

Some project communities are so small and tightly knit that everyone knows the vision. However, even these teams can benefit from creating a one-page vision statement. The only alternative to a clear vision is a confusing one—or no vision at all.

Further Reading

Some of the ideas in this section were inspired by the "Mastering Projects" workshop presented by True North pgs, Inc. If you have the opportunity to attend their workshop, take advantage of it.

Release Planning

Audience
Product Manager, Customers

We plan for success.

Imagine you've been freed from the shackles of deadlines. "Maximize our return on investment," your boss says. "We've already talked about the vision for this project. I'm counting on you to work out the details. Create your own plans and set your own release dates—just make sure we get a good return on our investment."

Now what?

One Project at a Time

First, work on only one project at a time. Many teams work on several projects simultaneously, which is a mistake. Task-switching has a substantial cost: "[T]he minimum penalty is 15

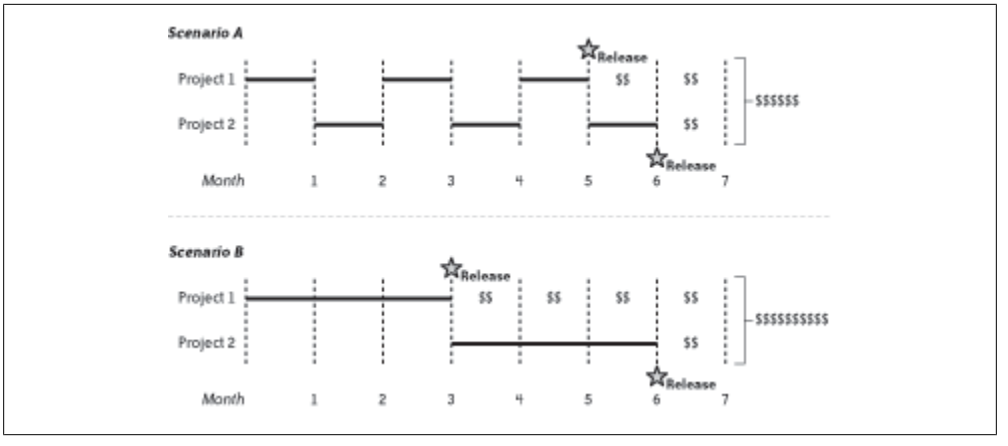

Figure 8-1. Effects of multitasking on value

percent... Fragmented knowledge workers may look busy, but a lot of their busyness is just thrashing" [DeMarco 2002]. Working on one project at a time allows you to release each project as you complete it, which increasees the total value of your work.

Consider a team that has two projects. In this simplified example, each project has equal value; when complete, each project will yield $$ in value every month. Each project takes three months to complete.

> NOTE
> **Although I'm describing value in dollar signs, money isn't the only source of value. Value can be intangible as well.**

In Scenario A (see Figure 8-1), the team works on both projects simultaneously. To avoid task-switching penalties, they switch between projects every month. They finish Project 1 after five months and Project 2 after six. At the end of the seventh month, the team has earned $$$$$$.

In Scenario B, the team works on just one project at a time. They release Project 1 at the end of the third month. It starts making money while they work on Project 2, which they complete after the sixth month, as before. Although the team's productivity didn't change—the projects still took six months—they earned more money from Project 1. By the end of the seventh month, they earned $$$$$$$$$$. That's nearly twice as much value with no additional effort.

Something this easy ought to be criminal. What's really astounding is the number of teams that work on simultaneous projects anyway.

Release Early, Release Often

Releasing early is an even better idea when you're working on a single project. If you group your most valuable features together and release them first, you can achieve startling improvements in value.

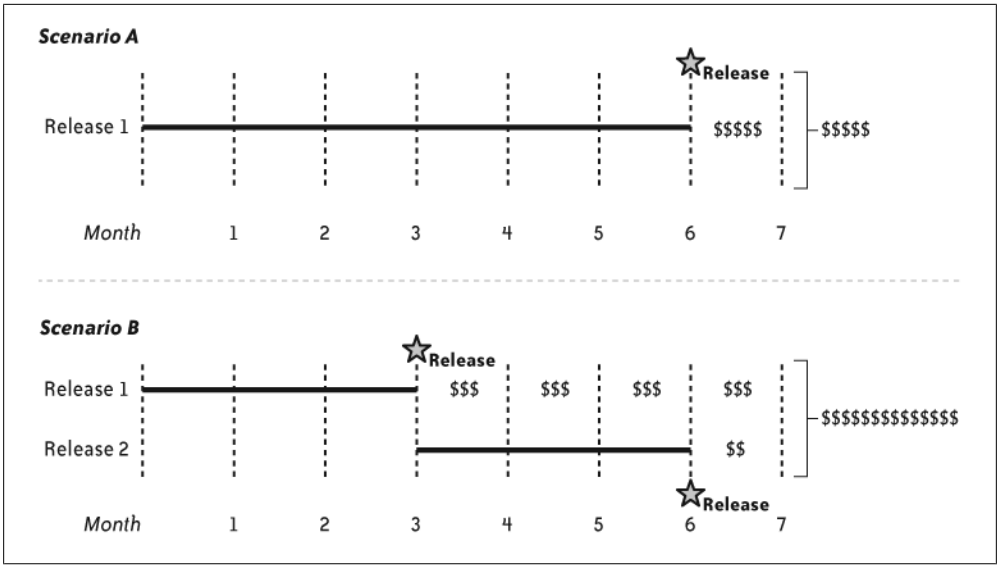

Figure 8-2. Effect of frequent releases on value

Consider another example team. This team has just one project. In Scenario A (see Figure 8-2), they build and release it after six months. The project is worth $$$$$ per month, so at the end of the seventh month, they've earned $$$$$.

In Scenario B, the team groups the most valuable features together, works on them first, and releases them after three months. The first release starts making $$$ per month. They then work on the remaining features and release them at the end of the sixth month. As before, their productivity hasn't changed. All that's changed is their release plan. Yet due to the income from the first release, the team has made $$$$$$$$$$$$$ by the end of the end of the seventh month—nearly *triple* that of Scenario A with its single release.

These scenarios are necessarily simplified. *Software by Numbers* [Denne & Cleland-Huang] has a more sophisticated example that uses real numbers and calculates value over the entire life of the product (see Table 8-1). In their example, the authors convert a five-year project with two end-of-project releases (Scenario A) into five yearly releases ordered by value (Scenario B). As before, the team's productivity remains the same.

Table 8-1. Realistic example of frequent releases

	Scenario A	Scenario B
Total Cost	$4.3 million	$4.712 million
Revenue	$5.6 million	$7.8 million
Investment	$2.76 million	$1.64 million
Payback	$1.288 million	$3.088 million
Net Present Value @ 10%	$194,000	$1.594 million
Internal Rate of Return	12.8%	36.3%

Scenario A is a marginal investment somewhat equivalent to obtaining a 12.8 percent interest rate. It requires an investment of $2.76 million and yields profits of $1.288 million. Considering the risk of software development, the investors can put that money to better use elsewhere. The project should not be funded.

Scenario B—the same project released more often—is an excellent investment somewhat equivalent to obtaining a 36.3 percent interest rate. Although Scenario B costs more because it conducts more releases, those releases allow the project to be self-funding. As a result, it requires a smaller investment of $1.64 million and yields profits of $3.088 million. This project is well worth funding.

Look at these results again. Each of these examples shows dramatic increases in value. Yet *nothing changed* except the order in which the teams released their features!

BENEFITS FOR PROGRAMMERS

Frequent releases are good for the organization. What's it worth to developers? Releases are painful, with flag days and repository freezes and rushes to complete, right?

Slow down. Breathe. Frequent releases can actually make your life easier.

By delivering tested, working, valuable software to your stakeholders regularly, you increase trust. Your stakeholders request a feature and soon see results. There's quick feedback between planning a release and getting the software. You will also get feedback *from* stakeholders more quickly. This allows you to learn and adapt.

There are technical benefits, too. One secret of XP is that doing hard things often and in small doses takes away most of the risk and almost all the pain. If you have the discipline to set up the necessary infrastructure to make a release at any point (with continuous integration and a 10-minute build), doing so takes only slightly more work than checking in your code and running the complete test suite.

Imagine eliminating all the stress of tracking down changes from a dozen branches and trying to merge multiple new features simultaneously to make a demo for a trade show next week that you just found out about on Thursday morning—because you can make a release at any time. Life is much better this way.

How to Release Frequently

Releasing frequently doesn't mean setting aggressive deadlines. In fact, aggressive deadlines *extend* schedules rather than reducing them [McConnell 1996] (p. 220). Instead, release more often by including less in each release. Minimum marketable features [Denne & Cleland-Huang] are an excellent tool for doing so.

A minimum marketable feature, or MMF, is the smallest set of functionality that provides value to your market, whether that market is internal users (as with custom software) or external customers (as with commercial software). MMFs provide value in many ways, such as competitive differentiation, revenue generation, and cost savings.

As you create your release plan, think in terms of stakeholder value. Sometimes it's helpful to think of stories and how they make up a single MMF. Other times, you may think of MMFs

that you can later decompose into stories. Don't forget the *minimum* part of minimum marketable feature—try to make each feature as small as possible.

Once you have minimal features, group them into possible releases. This is a brainstorming exercise, not your final plan, so try a variety of groupings. Think of ways to minimize the number of features needed in each release.

The most difficult part of this exercise is figuring out how to make small releases. It's one thing for a *feature* to be marketable, and another for a whole *release* to be marketable. This is particularly difficult when you're launching a new product. To succeed, focus on what sets your product apart, not the features it needs to match the competition.

An Example

Imagine you're the product manager for a team that's creating a new word processor. The market for word processors is quite mature, so it might seem impossible to create a small first release. There's so much to do just to *match* the competition, let alone to provide something new and compelling. You need basic formatting, spellchecking, grammar checking, tables, images, printing... the list goes on forever.

Approaching a word processor project in this way is daunting to the point where it may seem like a worthless effort. Rather than trying to match the competition, focus on the features that make your word processor unique. Release those features first—they probably have the most value.

Suppose that the competitive differentiation for your word processor is its powerful collaboration capabilities and web-based hosting. The first release might have four features: basic formatting, printing, web-based hosting, and collaboration. You could post this first release as a technical preview to start generating buzz. Later releases could improve on the base features and justify charging a fee: tables, images, and lists in one release, spellchecking and grammar checking in another, and so on.

If this seems foolish, consider Writely, the online word processing application. It doesn't have the breadth of features that Microsoft Word does, and it probably won't for many years. Instead, it focuses on what sets it apart: collaboration, remote document editing, secure online storage, and ease of use.*

According to venture capitalist Peter Rip, the developers released the first alpha of Writely *two weeks* after they decided to create it.† How much is releasing early worth? Ask Google. Ten months later, they bought Writely,‡ even though Writely *still* didn't come close to Microsoft Word's feature set.§ Writely is now known as Google Docs.

---

* *http://www.writely.com/*.

† "Writely is the seed of a Big idea," *http://earlystagevc.typepad.com/earlystagevc/2005/09/writely_is_the_.html*.

‡ "Writely—The Back Story," *http://earlystagevc.typepad.com/earlystagevc/2006/03/sam_steve_and_j.html*.

§ "Only in a bubble is Google's web WP an Office-killer," *http://www.theregister.co.uk/2006/03/10/google_writely_analysis/*.

CUSTOMERS AND FREQUENT RELEASES

"Our customers don't want releases that frequently!"

This may be true. Sometimes your customers won't accept releases as frequently as you deliver them. They may have regulatory requirements that necessitate rigorous testing and certification before they can install new versions.

Sometimes resistance to frequent releases comes from the hidden costs of upgrading. Perhaps upgrades require a difficult or costly installation process. Perhaps your organization has a history of requiring a series of hotfixes before a new release is stable.

Whatever the reason, the key to making frequent releases is to decrease the real or perceived costs of upgrading. Add an upgrade feature that notifies users of new versions and installs them automatically. Provide well-tested upgrade utilities that automatically convert user data.

Hosted applications, such as web applications, provide the ultimate in release flexibility. These allow you to release at any time, possibly without users even noticing. Some XP teams with hosted software and a mature set of existing features actually release *every day*.

Adapt Your Plans

If such significant results are possible from frequent releases, imagine what you could accomplish if you could also increase the value of each release. This is actually pretty easy: after each release, collect stakeholder feedback, cancel work on features that turned out to be unimportant, and put more effort into those features that stakeholders find most valuable.

With XP, you can change your plans more often than once per release. XP allows you to adjust your plan every iteration. Why? To react to unexpected challenges quickly. More importantly, it allows you to take advantage of opportunities. Where do these opportunities come from? You *create* them.

The beginning of every software project is when you know the least about what will make the software valuable. You might know a lot about its value, but you will always know *more* after you talk with stakeholders, show them demos, and conduct actual releases. As you continue, you will discover that some of your initial opinions about value were incorrect. No plan is perfect, but if you change your plan to reflect what you've learned—if you adapt—you create more value.

To increase the value of your software, create opportunities to learn. Think of your plan as a plan for *learning* as much as it is a plan for *implementation*. Focus on what you don't know. What are you uncertain about? What might be a good idea? Which good ideas can you prove in practice? Don't just speculate—create experiments. Include a way of testing each uncertainty.

For example, if you were creating a collaborative online word processor, you might not be sure how extensive your support for importing Microsoft Word documents should be. Some sort of support is necessary, but how much? Supporting all possible Word documents would take a long time to implement and prevent you from adding other, possibly more valuable features. Too little support could damage your credibility and cause you to lose customers.

To test this uncertainty, you could add a rudimentary import feature to your software (clearly marked "experimental"), release it, and have it create a report on the capabilities needed to support the types of documents that real users try to import. The information you gather will help you adapt your plan and increase your product's value.

NOTE

Web users are used to "beta" web applications, so releasing an experimental feature is possible in that context. A project with less forgiving users may require the use of a pre-release program, focus groups, or some other feedback mechanism.

Keep Your Options Open

To take the most advantage of the opportunities you create, build a plan that allows you to release at any time. Don't get me wrong—the point is not to *actually* release all the time, but to *enable* you to release at any time.

Why do this? It allows you to keep your options open. If an important but completely new opportunity comes along, you can release what you have and immediately change directions to take advantage of the opportunity. Similarly, if there's some sort of disaster, such as the project's surprise cancellation, you can release what you have anyway. At any time, you should be able to release a product that has value proportional to the investment you've made.

At any time, you should be able to release a product that has value proportional to the investment you've made.

To release at any time, build your plan so that each story stands alone. Subsequent stories can build on previous stories, but each one should be releasable on its own. For example, one item in your plan might be "Provide login screen," and the next might be "Allow login screen to have client-specific branding." The second item enhances the first, but the first is releasable on its own.

Suppose you're creating a system that gets data from a user, validates the data, and writes it to a database. You might initially create a story for each step: "Get data," "Validate data," and "Write data to database." These are sometimes called *horizontal stripes*. This is an easy way to create stories, but it prevents you from releasing, or even effectively reviewing, the software until you finish all three stories. It gives you less flexibility in planning, too, because the three stories form an all-or-nothing clump in your schedule.

A better approach is to create stories that do all three tasks but provide narrower individual utility. For example, you might create the stories "Process customer data," "Process shipping address," and "Process billing information." These are *vertical stripes* (see Figure 8-3).

Don't worry too much if you have trouble making your stories perfectly releasable. It takes practice. Releasable stories give you more flexibility in planning, but a few story clumps in your plan won't hurt much. With experience, you'll learn to make your plans less lumpy.

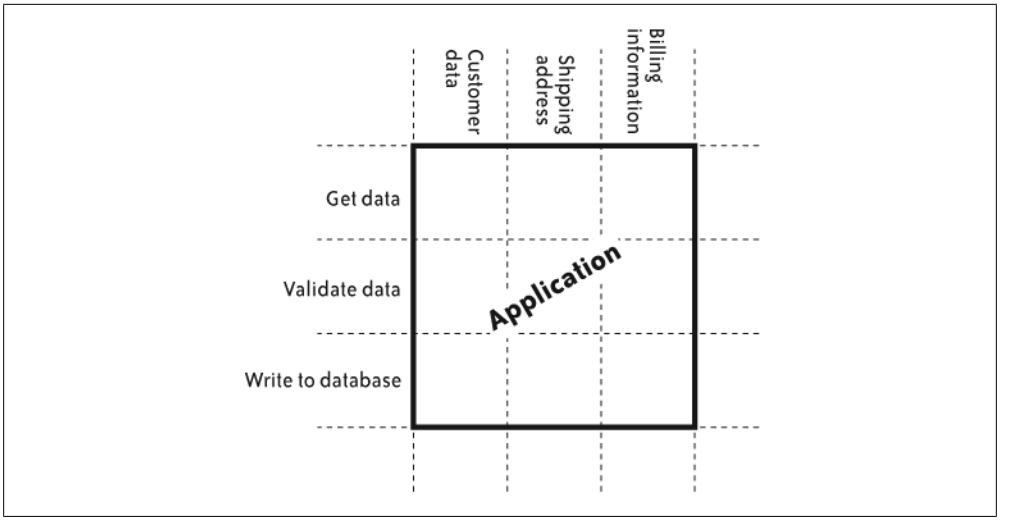

Figure 8-3. Horizontal and vertical stripes

How to Create a Release Plan

There are two basic types of plans: *scopeboxed* plans and *timeboxed* plans. A scopeboxed plan defines the features the team will build in advance, but the release date is uncertain. A timeboxed plan defines the release date in advance, but the specific features that release will include are uncertain.

> NOTE
> **Some people try to fix the release date *and* features. This can only end in tears; given the uncertainty and risk of software development, making this work requires adding a huge amount of padding to your schedule, sacrificing quality, working disastrous amounts of overtime, or all of the above.**

Timeboxed plans are almost always better. They constrain the amount of work you can do and force people to make difficult but important prioritization decisions. This requires the team to identify cheaper, more valuable alternatives to some requests. Without a timebox, your plan will include more low-value features.

To create your timeboxed plan, first choose your release dates. I like to schedule releases at regular intervals, such as once per month and no more than three months apart.

> NOTE
> **Does it seem odd to set the release date before deciding on features or estimates? Don't worry—you'll constrain your plan to fit into the time available.**

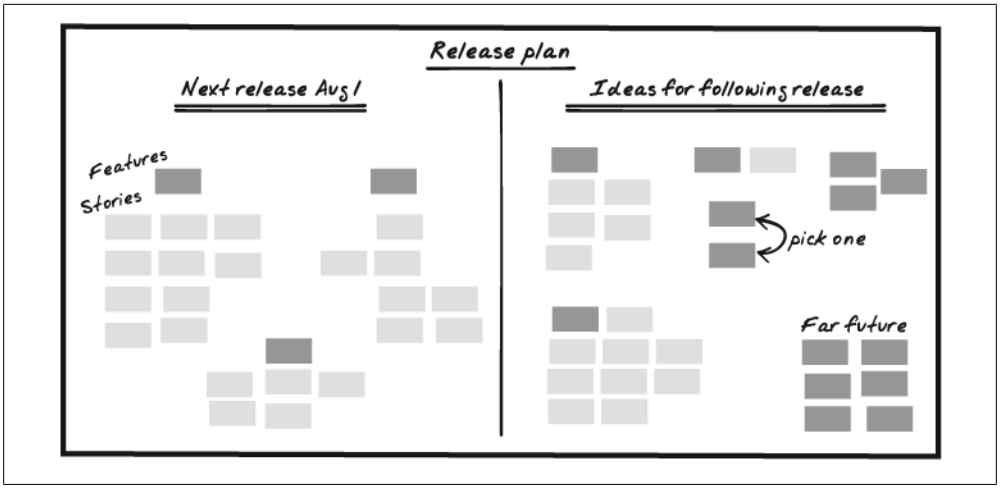

Figure 8-4. A release planning board

Now flesh out your plan by using your project vision to guide you in brainstorming minimum marketable features. Decompose these into specific stories, and work with the programmers to get estimates. Using the estimates as a guide, prioritize the stories so that the highest-value, lowest-cost stories are done first. (For more details, see "The Planning Game"" later in this chapter.)

Allies

Vision (p. 202)
Stories (p. 255)
Estimating (p. 261)

> NOTE
> **To brainstorm features and stories, use the vision to guide you, turn to interaction designers for ideas, and involve stakeholders as appropriate. Classic requirements gathering techniques may also help; see "Further Reading"" at the end of this section for suggestions.**

The end result will be a single list of prioritized stories. Using your velocity, risk factors, and story estimates, you can predict how many stories each release will include (see "Risk Management"" later this chapter). With that information as a guide, discuss options for reducing costs and splitting stories so that each release provides a lot of value.

Ally

Risk Management (p. 226)

This final list of stories is your release plan. Post it prominently (I use a magnetic whiteboard—see Figure 8-4) and refer to it during iteration planning. Every week, consider what you've learned from stakeholders and discuss how you can use that information to improve your plan.

Ally

Iteration Planning (p. 234)

"DONE DONE" AND RELEASE PLANNING

"Done done" applies to release planning as well as to stories. Just as you shouldn't postpone tasks until the end of an iteration, don't postpone stories until the end of a release.

Ally

No Bugs (p. 159)

Every feature should be "done done" before you start on the next feature. This means you need to schedule stories for reports, administration interfaces, security, performance, scalability, UI polish, and installers as appropriate. In particular, schedule bug-fix stories right away unless you've decided that they're not worth fixing in this release.

Planning at the Last Responsible Moment

It takes a lot of time and effort to brainstorm stories, estimate them, and prioritize them. If you're adapting your plan as you go, some of that effort will be wasted. To reduce waste, plan at the last responsible moment. The *last responsible moment* is the last moment at which you can responsibly make a decision (see "XP Concepts"" in Chapter 3). In practice, this means that the further away a particular event is, the less detail your release plan needs to contain.

Another way to look at this is to think in terms of planning horizons. Your *planning horizon* detemines how far you look into the future. Many projects try to determine every requirement for the project up front, thus using a planning horizon that extends to the end of the project.

To plan at the last responsible moment, use a tiered *set* of planning horizons. Use long planning horizons for general plans and short planning horizons for specific, detailed plans, as shown in Figure 8-5.

Your planning horizons depend on your situation and comfort level. The more commitments you need to make to stakeholders, the longer your detailed planning horizons should be. The more uncertain your situation is, or the more likely you are to learn new things that will change your plan, the shorter your planning horizons should be. If you aren't sure which planning horizons to use, ask your mentor for guidance. Here are some good starting points:

- Define the *vision* for the entire project.
- Define the *release date* for the next two releases.
- Define the *minimum marketable features* for the current release, and start to place features that won't fit in this release into the next release.
- Define all the *stories* for the current feature and most of the current release. Place stories that don't fit into the next release.
- *Estimate and prioritize* stories for the current iteration and the following three iterations.
- Determine *detailed requirements and customer tests* for the stories in the current iteration.

Allies

Vision (p. 202)
Stories (p. 255)
Estimating (p. 261)
Customer Tests (p. 280)

ADAPTIVE PLANNING IN ACTION

A few years ago, my wife and I took a two-month trip to Europe. It was easily the biggest, most complicated trip we've taken. We knew we couldn't plan it all in advance, so we used an adaptive approach.

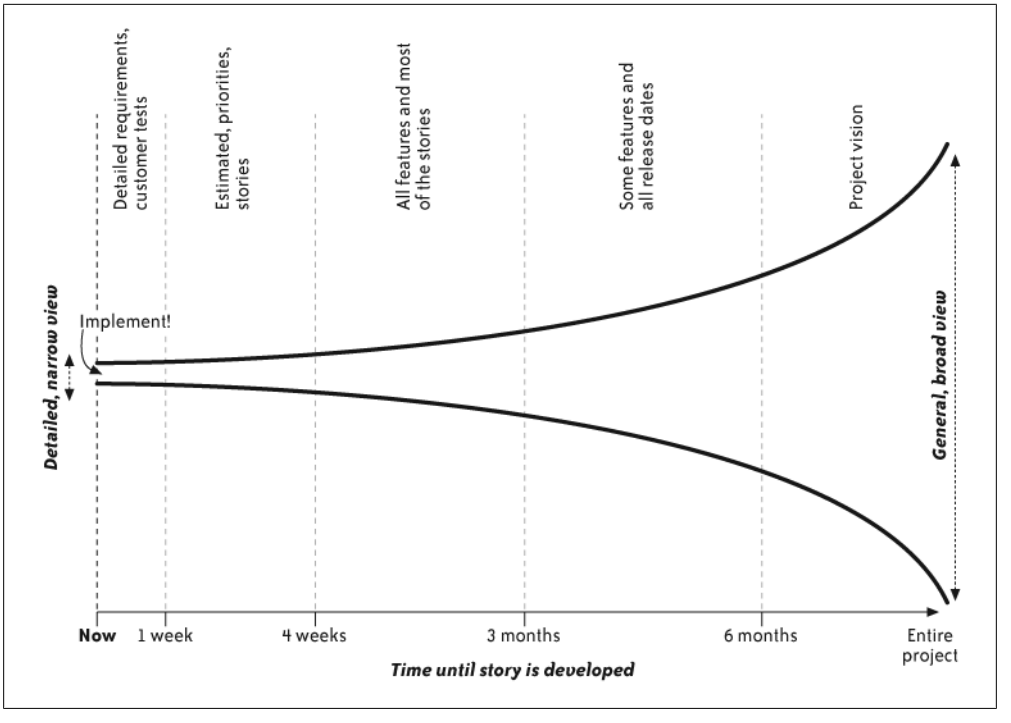

Figure 8-5. Planning horizons

We started with our vision for the trip. No, we didn't write a vision statement (she won't let me be that geeky), but we did agree that our goal for the trip was to visit a wide variety of European cities instead of going to one place and staying there. We discussed the countries we wanted to see, but we didn't make any decisions that would commit us to visiting a particular set.

We identified the last responsible moment for various decisions in our trip. Airline tickets generally get more expensive over time, so we booked our flight to London months in advance and made plans to stay with relatives there at the beginning and end of our trip. Hotels, however, only need a few days' notice. (In fact, too much advance notice means they might lose your reservation.)

We also took steps that would give us more options. We found a good guidebook that covered all of Europe. We purchased a EuroRail pass, which allowed us to use Europe's excellent rail system to travel through the continent. Although we thought we knew which countries we intended to visit, we spent extra money for a pass that would allow us to visit any European country.

With these general decisions made, we left the details for the last responsible moment. While on the trip, a few days before leaving for our next destination, we decided which country and city to visit next. We stopped by the train station, looked up departure times, and made reservations when necessary. We looked up candidate hotels in our guidebook and emailed the three most promising, then went back to enjoying the current city. The next day, we confirmed one of the hotel reservations. On the last day, we lazed about, picked a time to leave from list of train departure times, then boarded the train. We would arrive in the new city four or five hours later, drop off our belongings at the hotel, and go explore… then start the process over by thinking about the next city and country.

This approach not only gave us flexibility, it was *easy* and *relaxing*. Because we only made our reservations a day or two in advance, no hotel ever lost or confused our reservations. If we found that we particularly enjoyed a city, we stayed longer. If we didn't like it, we left early. On previous vacations, we had been slaves to a pre-planned itinerary, fretting over details for the entire trip. On this much longer and more complicated trip, we only had to think about the details for our next few days.

The flexibility also allowed us to experience things that we never would have otherwise. In Italy, we discovered that our intention to go to Turkey would eat up a huge amount of time in travel. We exploited our EuroRail passes and went to northern Europe instead and ended up having some of our most memorable experiences in cities we had never expected to visit.

By creating a rough plan in advance, keeping our options open, and making detailed decisions at the last responsible moment, we had a much better vacation than we would have had otherwise. Similarly, when you use adaptive planning in software development, unpredictable opportunities arise and allow you to increase the value of your software.

Adaptive Planning and Organizational Culture

Does the idea of spending two months travelling in a foreign country without advance hotel reservations seem scary? In practice, it was easy and relaxing, but when I tell the story of our adaptively planned trip to Europe (see the "Adaptive Planning in Action" sidebar), audiences get nervous.

Organizations often have a similar reaction to adaptive planning. An adaptive plan works to achieve a vision. However, just as my wife and I achieved our vision—"have fun visiting a lot of European cities"—but didn't know exactly which cities we would visit, an adaptive team will achieve its vision even though it cannot say exactly what it will deliver.

Ally

Vision (p. 202)

No aspect of agile development challenges organizational culture more than the transition to adaptive planning. It requires changes not only to the development team, but to reporting, evaluation, and executive oversight. The choice of adaptive planning extends to surprisingly diverse parts of the project community, and people often have a shocked or emotional reaction to the idea.

As a result, you may not be able to influence a change to adaptive planning. Unless you have executive support, any change that does occur will probably be slow and gradual. Even *with* executive support, this change is difficult.

Work within your organization's culture.

You can work within your organization's culture to do adaptive planning under the radar. Use adaptive planning, but set your planning horizons to match the organization's expectations. Generally, estimating and prioritizing stories for the remainder of the current release is enough. This works best if you have small, frequent releases.

As your stakeholders and executives gain trust in your ability to deliver, you may be able to shorten your detailed planning horizons and migrate further toward an adaptive plan.

Questions

I thought we were supposed to release every week. Is this different?

You may be confusing iterations with releases. Although the team should release software to internal stakeholders every week as part of the iteration demo, you may not choose to release to end-users or real customers that often.

Weekly releases are a great choice if you have the opportunity. Your ability to do so will depend on your business needs.

If we don't plan the entire project in detail, what should we tell our stakeholders about our plans?

Although you may not plan out all the details of your project in advance, you should have plenty of detail to share with stakeholders. You should always know the overall vision for the project. Depending on your planning horizons, you will probably have a list of the features for the next release as well as a planned date for that release. You will also have specific, estimated stories for near-term work.

If your stakeholders need more information or predictability, you may need longer planning horizons. In any event, be sure to let stakeholders know that this is your *current* plan and that it is subject to change if you find better ways of meeting the project vision.

Planning at the last responsible moment means we can't show exactly what we'll deliver. Doesn't that require too much trust from stakeholders?

Any development effort requires that the organization trust the team to do its job. If stakeholders require a detailed plan in order to trust you, use longer planning horizons that allow you to provide the plan they desire.

If we use short planning horizons, how can we be sure we'll deliver on the project vision?

If you're not sure you can deliver on the project vision, focus your plan on discovering whether you can. You may need to extend your planning horizons or create a small, limited-availability release to test crucial concepts. The details depend on your situation, so if you're not sure what to do, ask your mentor for guidance.

No matter your decision, clearly convey your concern to stakeholders, and let them know how you intend to address the uncertainty.

Results

When you create, maintain, and communicate a good release plan, the team and stakeholders all know where the project is heading. The plan shows how the team will meet the project vision, and team members are confident the plan is achievable. You complete features and release high-quality software regularly and consistently.

If you are adapting your plan well, you consistently seek out opportunities to learn new things about your plan, your product, and your stakeholders. As you learn, you modify your plan to take advantage of new insights. Stakeholders and the team agree that each release is better than originally planned.

Contraindications

Not all of these ideas are appropriate for everyone. I've put the easiest ones first, but even the easy ones have limitations.

Working on one project at a time is an easy, smart way to increase your return on investment. Despite its usefulness, working on one project at a time is anathema to some organizations. Proceed with caution.

Releasing frequently requires that your customers and users be able to accept more frequent releases. This is a no-brainer for most web-based software because users don't have to do anything to get updates. Other software might require painful software rollouts, and some even require substantial testing and certification. That makes frequent releases more difficult.

Adaptive planning requires that your organization define project success in terms of value rather than "delivered on time, on budget, and as specified." This can be a tough idea for some organizations to swallow. You may be able to assuage fears about adaptive plans by committing to a specific release date but leaving the details of the release unspecified.

Keeping your options open—that is, being ready to release, and thus change directions, at any time—requires a sophisticated development approach. Practices such as test-driven development, continuous integration, and incremental design and architecture help.

Allies

Test-Driven Development (p. 287)
Continuous Integration (p. 183)
Incremental Design and Architecture (p. 323)

Tiered planning horizons require a cohesive vision and regular updates to the plan. Use them when you can reliably revisit the plan at least once per iteration. Be sure your team includes a product manager and on-site customers who are responsible for maintaining the plan.

Finally, be cautious of plans without a predefined release date or a release date more than three months in the future. Without the checkpoint and urgency of a near-term release, these plans risk wandering off course.

Alternatives

The classic alternative to adaptive release planning is *predictive release planning*, in which the entire plan is created in advance. This can work in stable environments, but it tends to have trouble reacting to changes.

If you don't use incremental design and architecture, [Denne & Cleland-Huang] provide a sophisticated Incremental Funding Methodology that shows how to prioritize technical infrastructure alongside features. However, XP's use of incremental design neatly sidesteps this need.

Ally

Incremental Design and Architecture (p. 323)

Finally, teams with an established product and a relatively small need for changes and enhancements don't always need a release plan. Rather than thinking in terms of features or releases, these teams work from a small story backlog and release small enhancements every iteration. In some cases, they conduct daily deployment. You could think of this as an adaptive plan with a very short planning horizon.

Further Reading

Software by Numbers [Denne & Cleland-Huang] provides a compelling and detailed case for conducting frequent releases.

Agile Software Development Ecosystems [Highsmith] has an excellent discussion of adaptive planning in Chapter 15.

Lean Software Development [Poppendieck & Poppendieck] discusses postponing decisions and keeping your options open in Chapter 3.

The Planning Game

Audience
Whole Team

Our plans take advantage of both business and technology expertise.

You may know when and what to release, but how do you actually construct your release plan? That's where *the planning game* comes in.

In economics, a *game* is something in which "players select actions and the payoffs depend on the actions of all players."* The study of these games "deals with strategies for maximizing gains and minimizing losses... [and are] widely applied in the solution of various decision making problems."†

That describes the planning game perfectly. It's a structured approach to creating the best possible plan given the information available.

The planning game is most notable for the way it maximizes the amount of information contributed to the plan. It is strikingly effective. Although it has limitations, if you work within them, I know of no better way to plan.

How to Play

XP assumes that customers have the most information about *value*: what best serves the organization. Programmers have the most information about *costs*: what it will take to implement and maintain those features. To be successful, the team needs to maximize value while minimizing costs. A successful plan needs to take into account information from both groups, as every decision to *do* something is also a decision *not* to do something else.

Accordingly, the planning game requires the participation of both customers and programmers. (Testers may assist, but they do not have an explicit role in the planning game.) It's a *cooperative game*; the team as a whole wins or loses, not individual players.

Because programmers have the most information about costs—they're most qualified to say how long it will take to implement a story—they *estimate*.

* Deardorff's Glossary of International Economics, *http://www-personal.umich.edu/~alandear/glossary/g.html*.

† Dictionary definition of "game theory," *http://dictionary.reference.com/search?q=game theory&x=0&y=0*.

Because customers have the most information about value—they're most qualified to say what is important—they *prioritize*.

Neither group creates the plan unilaterally. Instead, both groups come together, each with their areas of expertise, and play the planning game:

1. Anyone creates a story or selects an unplanned story.
2. Programmers estimate the story.
3. Customers place the story into the plan in order of its relative priority.
4. The steps are repeated until all stories have been estimated and placed into the plan.

Allies

Stories (p. 255)
Estimating (p. 261)

NOTE

The planning game doesn't always follow this neat and orderly format. As long as programmers estimate and customers prioritize, the details aren't important. For example, the programmers may estimate a stack of stories all at once for the customers to prioritize later. Typically, most stories are created at the beginning of each release, during initial release planning sessions, as the team brainstorms what to include.

During the planning game, programmers and customers may ask each other questions about estimates and priorities, but each group has final say over its area of expertise.

The result of the planning game is a plan: a single list of stories in priority order. Even if two stories are of equivalent priority, one must come before the other. If you're not sure which to put first, pick one at random.

Overcoming disagreements

Release planning is always a difficult process because there are many more stories to do than there is time available to do them. Also, each stakeholder has his own priorities, and balancing these desires is challenging. Tempers rise and the discussion gets heated—or worse, some people sit back and tune out, only to complain later. This struggle is natural and happens on *any* project, XP or not, that tries to prioritize conflicting needs.

Ally

Stories (p. 255)

My favorite way to plan is to gather the team, along with important stakeholders, around a large conference table. Customers write stories on index cards, programmers estimate them, and customers place them on the table in priority order. One end of the table represents the stories to do first, and the other end represents stories to do last. The plan tends to grow from the ends toward the middle, with the most difficult decisions revolving around stories that are neither critical nor useless.

Use index cards to focus disagreements away from individuals.

Using index cards and spreading them out on a table allows participants to point to stories and move them around. It reduces infighting by demonstrating the amount of work to be done in a visible way. The conversation focuses on the cards and their relative priorities rather than on vague discussions of principles or on "must have/not important" distinctions.

ON DISAPPOINTMENT

The planning game is certain to give you information that makes you unhappy. You may feel tempted to blame the messenger and stop playing the planning game, or stop using XP altogether. That would be a mistake. As David Schmaltz of True North pgs says, every project has a fixed amount of disappointment associated with it. You can either use the planning game to dole out the disappointment in measured doses... or save it all up for the end.

How to Win

When customers and programmers work directly together throughout this process, something amazing happens. I call it *the miracle of collaboration*. It really is a miracle because time appears out of nowhere.

Like all miracles, it's not easy to achieve. When programmers give an estimate, customers often ask a question that causes every programmer's teeth to grind: "Why does it cost so much?"

The instictive reaction to this question is defensive: "It costs so much because software development is hard, damn it! Why are you questioning me!?"

Programmers, there's a better way to react. Reword the customer's question in your head into a simple request for information: "Why is this expensive?" Answer by talking about what's easy and what's difficult.

For example, imagine that a product manager requests a toaster to automatically pop up the toast when it finishes. The programmers reply that the feature is very expensive, and when the product manager asks why, the programmers calmly answer, "Well, popping up the toast is easy; that's just a spring. But detecting when the toast is done—that's new. We'll need an optical sensor and some custom brownness-detecting software."

This gives the product manager an opportunity to ask, "What about all those other toasters out there? How do they know when the toast is done?"

The programmers respond, "They use a timer, but that doesn't really detect when the toast is done. It's just a kludge."

Now the product manager can reply, "That's OK! Our customers don't want a super toaster. They just want a regular toaster. Use a timer like everyone else."

"Oh, OK. Well, that won't be expensive at all."

When you have honest and open dialog between customers and programmers, the miracle of collaboration occurs and extra time appears out of nowhere. Without communication, customers tend not to know what's easy and what's not, and they end up planning stories that are difficult to implement. Similarly, programmers tend not to know what customers think is important, and they end up implementing stories that aren't valuable.

With collaboration, the conflicting tendencies can be reconciled. For example, a customer could ask for something unimportant but difficult, and the programmers could point out the expense and offer easier alternatives. The product manager could then change directions and save time. Time appears out of nowhere. It's the miracle of collaboration.

Questions

Won't programmers pad their estimates or slack off if they have this much control over the plan?

In my experience, programmers are highly educated professionals with high motivation to meet customer expectations. [McConnell 1996] validates this experience: "Software developers like to work. The best way to motivate developers is to provide an environment that makes it easy for them to focus on what they like doing most, which is developing software... [Developers] have high achievement motivation: they will work to the objectives you specify, but you have to tell them what those objectives are" [McConnell 1996] (pp. 255–256).

Although programmer estimates may be higher than you like, it's most likely because they want to set realistic expectations. If the estimates do turn out to be too high, the team will achieve a higher velocity and automatically do more each iteration to compensate.

Won't customers neglect important technical issues if they have this much control over the plan?

Customers want to ship a solid, usable product. They have to balance that desire with the desire to meet crucial market windows. As a result, they may sometimes ask for options that compromise important technical capabilities. They do so because they aren't aware of the nuances of technical trade-offs in the same way that programmers are.

As a programmer, you are most qualified to make decisions on technical issues, just as the customers are most qualified to make decisions on business issues. When the customers ask for an explanation of an estimate, don't describe the technical options. Instead, *interpret* the technology and describe the options in terms of business impact.

Rather than describing the options like this:

> We're thinking about using a Mark 4 Wizzle-Frobitz optical sensor here for optimal release detection. We could use a Mark 1 spring-loaded countdown timer, too. We'd have to write some custom software to use the Mark 4, but it's very sophisticated, cutting-edge stuff and it will allow us to detect the exact degree of brownness of the bread. The Mark 1 is ancient tech without dynamic detection abilities, but it won't take any extra time to implement. Which would you prefer?

Try this instead:

> We have two choices for popping up toast. We can use either an optical sensor or a timer. The optical sensor will allow us to toast the bread to the user's exact preference, but it will increase our estimate by three days. The timer won't take any extra time but the user is more likely to have undercooked or burned toast. Which would you prefer?

If a technical option simply isn't appropriate, don't mention it, or mention your decision in passing as part of the cost of doing business:

> Because this is the first iteration, we need to install a version control system. We've included that cost in the estimate for our first story.

Our product manager doesn't want to prioritize. He says everything is important. What can we do?

Be firm. Yes, everything is important, but something has to come first and something will come last. Someone has to make the tough schedule decisions. That's the product manager's job.

Results

When you play the planning game well, both customers and programmers feel that they have contributed to the plan. Any feelings of pressure and stress are focused on the constraints of the plan and possible options, rather than on individuals and groups. Programmers suggest technical options for reducing scope while maintaining the project vision. Customers ruthlessly prioritize the stories that best serve the vision.

Ally

Vision (p. 202)

Contraindications

The planning game is an easy, effective approach that relies on many of XP's simplifying assumptions, such as:

- Customer-centric stories
- Story dependencies that customers can manage effectively (in practice, this means no technical dependencies and simple business dependencies)
- Customers capable of making wise prioritization decisions
- Programmers capable of making consistent estimates

Allies

Stories (p. 255)
The Whole Team (p. 28)
Estimating (p. 261)

If these conditions are not true on your team, you may not be able to take advantage of the planning game.

The planning game also relies on the programmers' abilities to implement design and architecture incrementally. Without this capability, the team will find itself creating technical stories or strange story dependencies that make planning more difficult.

Ally

Incremental Design and Architecture (p. 323)

Finally, the planning game assumes that the team has a single dominant constraint (for more information about the theory of constraints, see "XP Concepts"" in Chapter 3). It's very rare for a system to exhibit two constraints simultaneously, so this shouldn't be a problem. Similarly, the planning game assumes that the programmers are the constraint. If this isn't true for your team, discuss your options with your mentor.

Alternatives

There are a wide variety of project planning approaches. The most popular seems to be Gantt charts that assume task-based plans and schedule what each individual person will do.

In contrast to that approach, the planning game focuses on what the *team produces*, not on what *individuals do*. The team has the discretion to figure out how to produce each story and organizes itself to finish the work on time.

This focus on results, rather than on tasks, combined with the planning game's ability to balance customer and programmer expertise, makes it the most effective approach to software planning I've experienced. However, if you wish to use another approach to planning, you can

do so. Talk with your mentor about how to make your preferred approach to planning work with the rest of the XP practices.

Risk Management

Audience
Project Manager, Whole Team

We make and meet long term commitments.

The following statement is *nearly* true:

> Our team delivers a predictable amount of work every iteration. Because we had a velocity of 14 story points last week, we'll deliver 14 story points this week, and next week, and the next. By combining our velocity with our release plan, we can commit to a specific release schedule!

Good XP teams do achieve a stable velocity. Unfortunately, velocity only reflects the issues the team *normally* faces. Life always has some additional curve balls to throw. Team members get sick and take vacations; hard drives crash, and although the backups worked, the restore doesn't; stakeholders suddenly realize that the software you've been showing them for the last two months needs some major tweaks before it's ready to use.

Despite these uncertainties, your stakeholders need schedule commitments that they can rely upon. *Risk management* allows you to make and meet these commitments.

A Generic Risk-Management Plan

Every project faces a set of common risks: turnover, new requirements, work disruption, and so forth. These risks act as a multiplier on your estimates, doubling or tripling the amount of time it takes to finish your work.

How much of a multiplier do these risks entail? It depends on your organization. In a perfect world, your organization would have a database of the type shown in Figure 8-6.* It would show the chance of completing before various risk multipliers.

Because most organizations don't have this information available, I've provided some generic risk multipliers instead. (See Table 8-2.) These multipliers show your chances of meeting various schedules. For example, in a "Risky" approach, you have a 10 percent chance of finishing according to your estimated schedule. Doubling your estimates gives you a 50 percent chance of on-time completion, and to be virtually certain of meeting your schedule, you have to quadruple your estimates.

Table 8-2. Generic risk multipliers

	Process approach		
Percent chance	Rigorous[a]	Risky[b]	Description
10%	x1	x1	Almost impossible ("ignore")
50%	x1.4	x2	50-50 chance ("stretch goal")

* Reprinted from [Little].

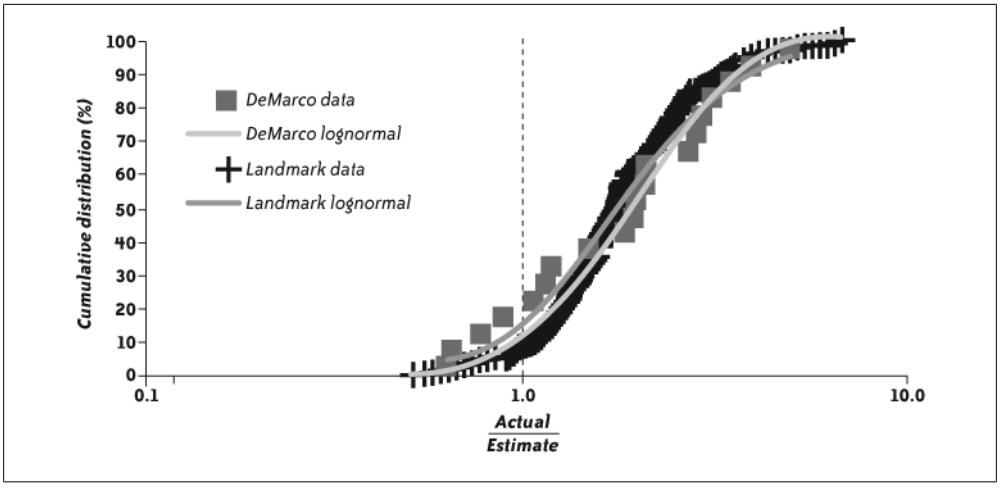

Figure 8-6. An example of historical project data

	Process approach		
Percent chance	Rigorous[a]	Risky[b]	Description
90%	x1.8	x4	Virtually certain ("commit")

[a] These figures are based on DeMarco & Lister's RISKOLOGY simulator, version 4a, available from *http://www.systemsguild.com/riskology.html*. I used the standard settings but turned off productivity variance, as velocity would automatically adjust for that risk.

[b] These figures are based on [Little].

If you use the XP practices—in particular, if you're strict about being "done done" every iteration, your velocity is stable, and you fix all your bugs each iteration—then your risk is lowered. Use the risk multiplier in the "Rigorous" column. On the other hand, if you're *not* strict about being "done done" every iteration, if your velocity is unstable, or if you postpone bugs and other work for future iterations, then use the risk multiplier in the "Risky" column.

Allies

"Done Done" (p. 155)
No Bugs (p. 159)

NOTE

These risk multipliers illustrate an important difference between risky and rigorous appraoches. Both can get lucky and deliver according to their estimates. Risky approaches, however, take a lot longer when things go wrong. They require much more padding in order to make and meet commitments.

Although these numbers come from studies of hundreds of industry projects, those projects didn't use XP. As a result, I've guessed somewhat at how accurately they apply to XP. However, unless your company has a database of prior projects to turn to, they are your best starting point.

Project-Specific Risks

Using the XP practices and applying risk multipliers will help contain the risks that are common to all projects. The generic risk multipliers include the normal risks of a flawed release plan, ordinary requirements growth, and employee turnover. In addition to these risks, you probably face some that are specific to your project. To manage these, create a *risk census*—that is, a list of the risks your project faces that focuses on your project's *unique* risks.

[DeMarco & Lister 2003]suggest starting work on your census by brainstorming catastrophes. Gather the whole team and hand out index cards. Remind team members that during this exercise, negative thinking is not only OK, it's necessary. Ask them to consider ways in which the project could fail. Write several questions on the board:*

1. What about the project keeps you up at night?
2. Imagine it's a year after the project's disastrous failure and you're being interviewed about what went wrong. What happened?
3. Imagine your best dreams for the project, then write down the opposite.
4. How could the project fail without anyone being at fault?
5. How could the project fail if it were the stakeholders' faults? The customers' faults? Testers? Programmers? Management? Your fault? Etc.
6. How could the project succeed but leave one specific stakeholder unsatisfied or angry?

Write your answers on the cards, then read them aloud to inspire further thoughts. Some people may be more comfortable speaking out if a neutral facilitator reads the cards anonymously.

Once you have your list of catastrophes, brainstorm scenarios that could lead to those catastrophes. From those scenarios, imagine possible root causes. These root causes are your risks: the causes of scenarios that will lead to catastrophic results.

Ally

Root-Cause Analysis (p. 91)

For example, if you're creating an online application, one catastrophe might be "extended downtime." A scenario leading to that catastrophe would be "excessively high demand," and root causes include "denial of service attack" and "more popular than expected."

After you've finished brainstorming risks, let the rest of the team return to their iteration while you consider the risks within a smaller group. (Include a cross-section of the team.) For each risk, determine:

- Estimated probability—I prefer "high," "medium," and "low."
- Specific impact to project if it occurs—dollars lost, days delayed, and project cancellation are common possibilities.

You may be able to discard some risks as unimportant immediately. I ignore unlikely risks with low impact and all risks with negligible impact. Your generic risk multipler accounts for those already.

For the remainder, decide whether you will *avoid* the risk by not taking the risky action; *contain* it by reserving extra time or money, as with the risk multiplier; or *mitigate* it by taking

* Based on [DeMarco & Lister 2003] (p. 117)

steps to reduce its impact. You can combine these actions. (You can also *ignore* the risk, but that's irresponsible now that you've identified it as important.)

For the risks you decide to handle, determine transition indicators, mitigation and contingency activities, and your risk exposure:

- *Transition indicators* tell you when the risk will come true. It's human nature to downplay upcoming risks, so choose indicators that are objective rather than subjective. For example, if your risk is "unexpected popularity causes extended downtime," then your transition indicator might be "server utilization trend shows upcoming utilization over 80 percent."
- *Mitigation* activities reduce the impact of the risk. Mitigation happens in advance, regardless of whether the risk comes to pass. Create stories for them and add them to your release plan. To continue the example, possible stories include "support horizontal scalability" and "prepare load balancer."
- *Contingency* activities also reduce the impact of the risk, but they are only necessary if the risk occurs. They often depend on mitigation activities that you perform in advance. For example, "purchase more bandwidth from ISP," "install load balancer," and "purchase and prepare additional frontend servers."
- *Risk exposure* reflects how much time or money you should set aside to contain the risk. To calculate this, first estimate the numerical probability of the risk and then multiply that by the impact. When considering your impact, remember that you will have already paid for mitigation activities, but contingency activities are part of the impact. For example, you might believe that downtime due to popularity is 35 percent likely, and the impact is three days of additional programmer time and $20,000 for bandwidth, colocation fees, and new equipment. Your total risk exposure is $7,000 and one day.

Some risks have a 100 percent chance of occurring. These are no longer risks—they are reality. Update your release plan to deal with them.

Other risks can kill your project if they occur. For example, a corporate reorganization might disband the team. Pass these risks on to your executive sponsor as assumptions or requirements. ("We assume that, in the event of a reorg, the team will remain intact and assigned to this project.") Other than documenting that you did so, and perhaps scheduling some mitigation stories, there's no need to manage them further.

For the remaining risks, update your release plan to address them. You will need stories for mitigation activities, and you may need stories to help you monitor transition indicators. For example, if your risk is "unexpected popularity overloads server capacity," you might schedule the story "prepare additional servers in case of high demand" to mitigate the risk, and "server load trend report" to help you monitor the risk.

You also need to set aside time, and possibly money, for contingency activities. Don't schedule any contingency stories yet—you don't know if you'll need them. Instead, add up your risk exposure and apply dollar exposure to the budget and day exposure to the schedule. Some risks will occur and others won't, but on average, the impact will be equal to your risk exposure.

MONITORING RISKS

One of the hardest things about project-specific risks is remembering to follow up on them. It's human nature to avoid unpleasant possibilities. The best way I've found to monitor risks is to assign

someone to track them. The project manager is a good choice for this role, but you can choose anybody on the team. However, choose someone who's not the team's constraint. It's more important for them to do things that no one else can do.

Every week, review your project-specific risks and check the transition indicators. Consider whether the risks are still applicable, and ask yourself if any new risks have come to light.

NOTE

I write project-specific risks on yellow cards and put them on the release planning board.

How to Make a Release Commitment

Ally

Release Planning (p. 207)

With your risk exposure and risk multipliers, you can predict how many story points you can finish before your release date. Start with your timeboxed release date from your release plan. (Using scopeboxed release planning? See the "Predicting Release Dates"" sidebar.) Figure out how many iterations remain until your release date and subtract your risk exposure. Multiply by your velocity to determine the number of points remaining in your schedule, then divide by each risk multiplier to calculate your chances of finishing various numbers of story points.

```
risk_adjusted_points_remaining = (iterations_remaining - risk_exposure) * velocity / risk_multiplier
```

For example, if you're using a rigorous approach, your release is 12 iterations away, your velocity is 14 points, and your risk exposure is one iteration, you would calculate the range of possibilities as:

```
points remaining = (12 - 1) * 14 = 154 points
10 percent chance: 154 / 1 = 154 points
50 percent chance: 154 / 1.4 = 110 points
90 percent chance: 154 / 1.8 = 86 points
```

In other words, when it's time to release, you're 90 percent likely to have finished 86 more points of work, 50 percent likely to have finished 110 more points, and only 10 percent likely to have finished 154 more points.

You can show this visually with a *burn-up chart*, shown in Figure 8-7.* Every week, note how many story points you have completed, how many total points exist in your next release (completed plus remaining stories), and your range of risk-adjusted points remaining. Plot them on the burn-up chart as shown in the figure.

Use your burn-up chart and release plan to provide stakeholders with a list of features you're committing to deliver on the release date. I commit to delivering features that are 90 percent likely to be finished, and I describe features between 50 and 90 percent likely as *stretch goals*. I don't mention features that we're less than 50 percent likely to complete.

* According to John Brewer (*http://tech.groups.yahoo.com/group/extremeprogramming/message/81856*), the burn-up chart was created by Phil Goodwin as a variant of Scrum's burn-down charts. I've modified it further to include risk-based commitments.

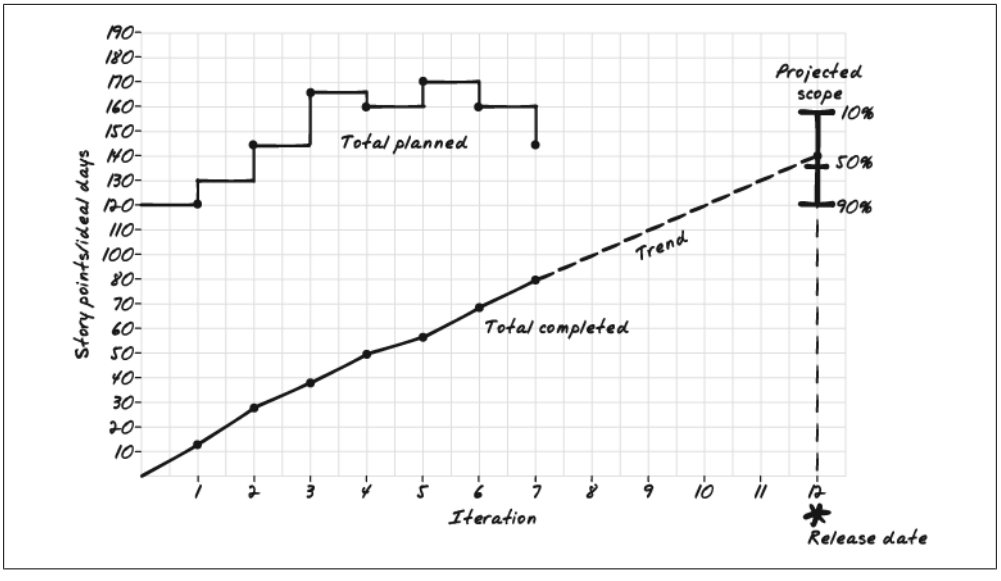

Figure 8-7. A burn-up chart

PREDICTING RELEASE DATES

I recommend using timeboxed plans with a fixed release date—they're less risky. However, you can predict a release date for a given set of stories. To do so, start with the total number of points in the stories you want to deliver. Divide by your velocity to get the number of iterations required to finish those stories, then multiply by each risk multiplier. Finish by adding your risk exposure to each risk-adjusted prediction.

```
risk_adjusted_iterations_remaining = (points_remaining / velocity * risk_multiplier) + risk_exposure
```

Success over Schedule

The majority of this discussion of risk management has focused on managing the risk to your *schedule* commitments. However, your real goal should be to deliver an organizational success—to deliver software that provides substantial value to your organization, as guided by the success criteria in your project vision.

> Success is more than a delivery date.

Taking too long can put that success at risk, so rapid delivery *is* important. Just don't forget about the real goal. As you evaluate your risks, think about the risk to the success of the *project*, not just the risk to the *schedule*. Take advantage of adaptive release planning. Sometimes you're better off taking an extra month to deliver a great result, rather than just a good one.

Ally
Vision (p. 202)

Ally
Release Planning (p. 207)

When Your Commitment Isn't Good Enough

Someday, someone will ask you to commit to a schedule that your predictions show is impossible to achieve. The best way to make the schedule work is to reduce scope or extend the release date. If you can't do so personally, ask your manager or product manager to help.

If you can't change your plan, you may be able to improve your velocity (see "Estimating"" later in this chapter). This is a long shot, so don't put too much hope in it.

It's also possible that your pessimistic predictions are the result of using a risky process. You can improve the quality of your predictions by being more rigorous in your approach. Make sure you're "done done" every iteration and include enough slack to have a stable velocity. Doing so will decrease your velocity, but it will also decrease your risk and allow you to use risk multipliers from the "Rigorous" column, which may lead to a better overall schedule.

Allies

"Done Done" (p. 155)
Slack (p. 247)

Typically, though, your schedule is what it is. If you can't change scope or the date, you can usually change little else. As time passes and your team builds trust with stakeholders, people may become more interested in options for reducing scope or extending the delivery date. In the meantime, don't promise to take on more work than you can deliver. Piling on work will increase technical debt and hurt the schedule. Instead, state the team's limitations clearly and unequivocally.

Ally

Trust (p. 103)

If that isn't satisfactory, ask if there's another project the team can work on that will yield more value. Don't be confrontational, but don't give in, either. As a leader, you have an obligation to the team—and to your organization—to tell the truth.

In some organizations, inflexible demands to "make it work" are questionable attempts to squeeze more productivity out of the team. Sadly, applying pressure to a development team tends to reduce quality without improving productivity. In this sort of organization, as the true nature of the schedule becomes more difficult to ignore, management tends to respond by laying on pressure and "strongly encouraging" overtime.

Look at the other projects in the organization. What happens when they are late? Does management respond rationally, or do they respond by increasing pressure and punishing team members?

If it's the latter, decide now whether you value your job enough to put up with the eventual pressure. If not, start looking for other job opportunities immediately. (Once the pressure and mandatory overtime begin, you may not have enough time or energy for a job hunt.) If your organization is large enough, you may also be able to transfer to another team or division.

As a tool of last resort, *if* you're ready to resign *and* you're responsible for plans and schedules, it's entirely professional to demand respect. One way to do so is to say, "Because you no longer trust my advice with regard to our schedule, I am unable to do the job you hired me to do. My only choice is to resign. Here is my resignation letter."

This sometimes gets the message through when nothing else will work, but it's a big stick to wield, and it could easily cause resentment. Be careful. "Over my dead body!" you say. "Here's your noose," says the organization.

Questions

What should we tell our stakeholders about risks?

It depends how much detail each stakeholder wants. For some, a commitment and stretch goal may be enough. Others may want more detailed information.

Be sure to share your risk census and burn-up chart with your executive sponsor and other executives. Formally transfer responsibility for the project assumptions (those risks that will kill the project if they come true). Your assumptions are the executives' risks.

Your risk multipliers are too high. Can I use a lower multiplier?

Are your replacement multipliers based on objective data? Is that data representative of the project you're about to do? If so, go ahead and use them. If not, be careful. You can lower the multiplier, but changing the *planned* schedule won't change the *actual* result.

We're using a scopeboxed plan, and our stakeholders want a single delivery date rather than a risk-based range. What should we do?

Your project manager or product manager might be able to help you convince the organization to accept a risk-based range of dates. Talk with them about the best way to present your case.

If you must have a single date, pick a single risk multiplier to apply. Which one you choose depends on your organization. A higher risk multiplier improves your chances of success but makes the schedule look worse. A lower risk multiplier makes the schedule look better but reduces your chances of meeting that schedule.

Many organizations have acclimated to slipping delivery dates. Managers in these organizations intuitively apply an informal risk multiplier in their head when they hear a date. In this sort of organization, applying a large risk multiplier might make the schedule seem ridiculously long.

Consider other projects in your organization. Do they usually come in on time? What happens if they don't? Talk with your project manager and product manager about management and stakeholder expectations. These discussions should help you choose the correct risk multiplier for your organization. Remember, though, using a risk-based *range* of dates is a better option, and using a timeboxed schedule is better than using a scopeboxed schedule.

Results

With good risk management, you deliver on your commitments even in the face of disruptions. Stakeholders trust and rely on you, knowing that when they need something challenging yet valuable, they can count on you to deliver it.

Contraindications

Use risk management only for external commitments. Within the team, focus your efforts on achieving the unadjusted release plan as scheduled. Otherwise, your work is likely to expand to meet the deadline. Figure 8-8* shows how a culture of doubling estimates at one company

* Reprinted from [Little].

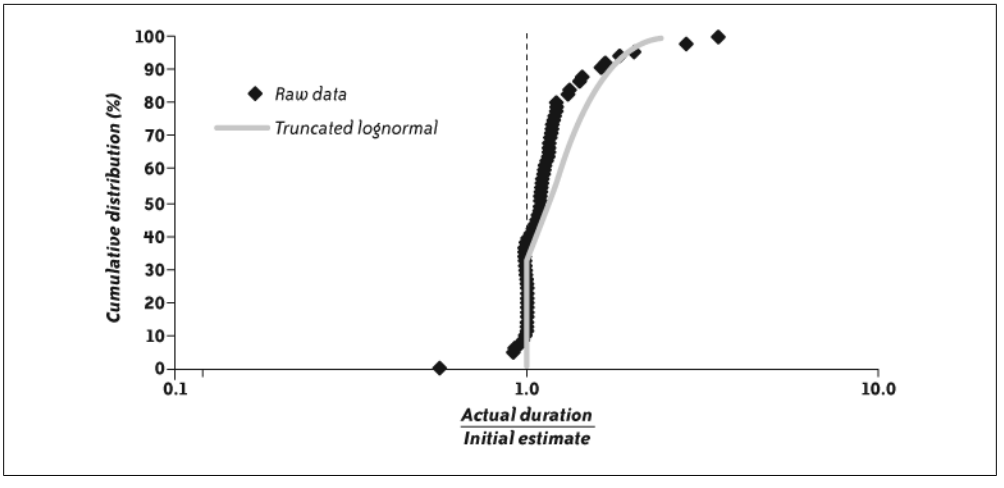

Figure 8-8. Just double the estimate

prevented most projects from finishing early without reducing the percentage of projects that finished late.

Be careful of using risk management in an organization that brags that "failure is not an option." You may face criticism for looking for risks. You may still be able to present your risk-derived schedule as a commitment or stretch goal, but publicizing your risk census may be a risk itself.

Alternatives

Risk management is primarily an organizational decision rather than an individual team decision. If your organization has already institutionalized risk management, they may mandate a different approach. Your only trouble may be integrating it into XP's simultaneous phases; to do so, use this description as a starting point and consider asking your mentor (see "Find a Mentor"" in Chapter 2) for advice specific to your situation.

Some organizations add a risk buffer to individual estimates rather than the overall project schedule. As Figure 8-8 illustrates, this tends to lead to waste.

Further Reading

Waltzing with Bears: Managing Risk on Software Projects [DeMarco & Lister 2003] provides more detail and considers risk management from an organizational perspective. My favorite quote comes from the back cover: "If there's no risk on your next project, don't do it."

Iteration Planning

We stop at predetermined, unchangeable time intervals and compare reality to plan.

Audience
Whole Team

Iterations are the heartbeat of an XP project. When an iteration starts, stories flow in to the team as they select the most valuable stories from the release plan. Over the course of the iteration, the team breathes those stories to life. By the end of the iteration, they've pumped out working, tested software for each story and are ready to begin the cycle again.

Allies

Stories (p. 255)
Release Planning (p. 207)

Iterations are an important safety mechanism. Every week, the team stops, looks at what it's accomplished, and shares those accomplishments with stakeholders. By doing so, the team coordinates its activities and communicates its progress to the rest of the organization. Most importantly, iterations counter a common risk in software projects: the tendency for work to take longer than expected.

The Iteration Timebox

Programming schedules die in inches. At first you're on schedule: "I'll be done once I finish this test." Then you're limping: "I'll be done as soon as I fix this bug." Then gasping: "I'll be done as soon as I research this API flaw... no, really." Before you know it, two days have gone by and your task has taken twice as long as you estimated.

Death by inches sneaks up on a team. Each delay is only a few hours, so it doesn't feel like a delay, but they multiply across the thousands of tasks in a project. The cumulative effects are devastating.

Iterations allow you to avoid this surprise. Iterations are exactly one week long and have a strictly defined completion time. This is a *timebox*: work ends at a particular time regardless of how much you've finished. Although the iteration timebox doesn't *prevent* problems, it *reveals* them, which gives you the opportunity to correct the situation.

In XP, the iteration demo marks the end of the iteration. Schedule the demo at the same time every week. Most teams schedule the demo first thing in the morning, which gives them a bit of extra slack the evening before for dealing with minor problems.

Allies

Iteration Demo (p. 138)
Slack (p. 247)

The Iteration Schedule

Iterations follow a consistent, unchanging schedule:

- Demonstrate previous iteration (up to half an hour)
- Hold retrospective on previous iteration (one hour)
- Plan iteration (half an hour to four hours)
- Commit to delivering stories (five minutes)
- Develop stories (remainder of iteration)
- Prepare release (less than 10 minutes)

Allies

Iteration Demo (p. 138)
Retrospectives (p. 94)
Ten-Minute Build (p. 177)

Many teams start their iterations on Monday morning and end Friday evening, but I prefer iterations that start on Wednesday morning. This allows people to leave early on Friday or take Monday off without missing important events. It

> Choose an iteration start time that works for your team, and stick with it.

also allows the team to conduct releases before the weekend.

How to Plan an Iteration

After the iteration demo and retrospective are complete, iteration planning begins. Start by measuring the velocity of the previous iteration. Take all the stories that are "done done" and add up their original estimates. This number is the amount of story points you can reasonably expect to complete in the upcoming iteration.

Ally
"Done Done" (p. 155)

NOTE

Be sure to use *estimated* time, not actual time, when calculating velocity. This allows the velocity to account for interruptions and other overhead.

With your velocity in hand, you can select the stories to work on this iteration. Ask your customers to select the most important stories from the release plan. Select stories that exactly add up to the team's velocity. You may need to split stories (see "Stories"" later in this chapter) or include one or two less important stories to make the estimates add up perfectly.

Ally
Release Planning (p. 207)

NOTE

Avoid using the iteration planning meeting for extensive release planning. Do the bulk of your release planning during the previous iteration.

Because the iteration planning meeting takes stories from the front of the release plan, you should have already estimated and prioritized those stories. As a result, selecting stories for the iteration plan should only take a few minutes, with perhaps 10 or 15 minutes more to explain:

> Product Manager*: OK, here we are again. What is it this time? Iteration 23? [Writes "Iteration 23" at top of the blank iteration planning whiteboard.]* What was our velocity in the previous iteration?
>
> Programmer*: [Counts up stories.]* Fourteen, same as usual.
>
> Product Manager*: [Writes "Velocity: 14" on the whiteboard.]* OK, let's roll the release planning board in here. *[Helps drag the planning board over, then points at it.]* The other customers and I had some tough decisions to make after our recent release. Our users love what we've been doing, which is good, and they're making all kinds of suggestions, which is also good, but some of their ideas could easily derail us, which is not so good. Anyway, you don't need to worry about that. Bottom line is that we're sticking with the same plan as before, at least for this next iteration. So, let's see, 14 points... *[starts picking stories off of the release planning board]* 3... 5... 6, 7... 10... 12... 14. *[He puts them on the table.]* All right, everybody, that's our iteration. Good luck—I have to go talk with our division VP. Mary's going to stick around to answer any questions you have about these stories. I'll be back this afternoon if you need anything.
>
> Mary: Good luck, Brian. *[The product manager leaves.]* We've talked about these stories before, so I'll just refresh your memory...

After you have chosen the stories for the iteration, everybody but the programmers can leave the meeting, although anybody is welcome to stay if she likes. At least one customer should stick around to answer programmer questions and to keep an ear out for misunderstandings.

> NOTE
> **The team's constraint determines how much the team can do in each iteration (see "XP Concepts" in Chapter 3 for more about the Theory of Constraints). This book assumes programmers are the constraint, so the iteration plan is based entirely on programmer tasks and estimates. Other team members may conduct their own iteration planning sessions if they wish, but it isn't required.**

At this point, the real work of iteration planning begins. Start by breaking down the stories into engineering tasks.

Engineering tasks are concrete tasks for the programmers to complete. Unlike stories, engineering tasks don't need to be customer-centric. Instead, they're programmer-centric. Typical engineering tasks include:

- Update build script
- Implement domain logic
- Add database table and associated ORM objects
- Create new UI form

Brainstorm the tasks you need in order to finish all the iteration's stories. Some tasks will be specific to a single story; others will be useful for multiple stories. Focus only on tasks that are necessary for completing the iteration's stories. Don't worry about all-hands meetings, vacations, or other interruptions.

> NOTE
> **Some nonprogramming stories, such as a story to estimate a stack of other stories, can't be broken into tasks. Reestimate the story in terms of ideal hours (see "Estimating" later in this chapter) and leave it at that.**

Brainstorming tasks is a *design* activity. If everybody has the same ideas about how to develop the software, it should go fairly quickly. If not, it's a great opportunity for discussion before coding begins. You don't need to go into too much detail. Each engineering task should take a pair one to three hours to complete. (This translates into about two to six hours of estimated effort.) Let pairs figure out the details of each task when they get to them.

As each team member has an idea for a task, he should write it down on a card, read it out loud, and put it on the table. Everybody can work at once. You'll be able to discard duplicate or inappropriate tasks later.

As you work, take advantage of your on-site customer's presence to ask about the detailed requirements for each story. What do the customers expect when the story is done?

> Amy, John, Kim, Fred, Belinda, and Joe continue planning. Mary, one of the on-site customers, sits off to the side, working on email.
>
> "OK, here we are again," deadpans John, mimicking their product manager.

Amy snickers. "These stories look pretty straightforward to me. Obviously, we need to implement new domain logic for each story. This warehouse stocking story looks like it will affect the warehouse and SKU classes." She takes an index card and writes, "Update warehouse and SKU classes for warehouse stocking."

"Speaking of products, now that we're going to be storing the weight of each SKU, we need to update the SKU class for that, too," says Belinda. She takes another card and writes, "Update SKU class to include weight." She pauses to consider. "Maybe that's too small to be a task."

Joe speaks up. "That weight story is mostly a UI change at this point. We'll need to update the SKU configuration screen and the administration screen. We can put weight into the SKU class at the same time." He starts writing another task card.

"Wait a second, Joe." Mary looks up from her email. "Did you say that the weight issue was only a UI change?"

"That's right," says Joe.

"That's not entirely true," says Mary. "Although we mostly need to add the ability to enter weight into the system for later use, we do want it to show up on our inventory report."

"Oh, good to know," says Joe. He writes, "Update inventory report with SKU weight" on a task card.

Kim and Fred have been talking off to the side. Now they speak up. "We've made some cards for the database changes we'll need this iteration," Kim says. "Also, I think we need to update our build script to do a better job of updating schemas, so I wrote a task card for that, too."

"Sounds good," says John. "Now, I think these other stories also need domain logic changes..."

After you've finished brainstorming tasks, spread them out on the table and look at the whole picture. Are these tasks enough to finish all the stories? Are there any duplicates or overlaps? Is anybody uncertain about how the plan works with the way the software is currently designed? Discuss and fix any problems.

Finish brainstorming before you start estimating.

Next, estimate the tasks. As with brainstorming, this can occur in parallel, with individual programmers picking up cards, writing estimates, and putting them back. Call out the estimates as you finish them. If you hear somebody call out an estimate you disagree with, stop to discuss it and come to consensus.

Estimate the tasks in *ideal hours*. How long would the task take if you had perfect focus on the task, suffered no interruptions, and could have the help of anybody on the team? Estimate in person-hours as well: a task that takes a pair two hours is a four-hour estimate. If any of the tasks are bigger than six hours of effort, split them into smaller tasks. Combine small tasks that are less than an hour or two.

Finally, stop and take a look at the plan again. Does anybody disagree with any of the estimates? Does everything still fit together?

As a final check, add up the estimates and compare them to the total task estimates from your previous iteration. Using this plan, can you commit to delivering all the stories? Is there enough slack in the plan for dealing with unexpected problems?

Ally
Slack (p. 247)

NOTE

Comparing the total of your task estimates to last week's total will help you get a feel for whether the iteration is achievable. The two numbers don't need to match exactly.

Don't bother comparing your estimate to the actual number of hours you'll be working. There are too many variables outside of your control, such as estimate accuracy, interruptions, and nonproject meetings, for there to be a useful corollation.

You may discover that you aren't comfortable committing to the plan you have. If so, see if there are any tasks you can remove or simplify. Discuss the situation with your on-site customers. Can you replace a difficult part of a story with something easier but equally valuable? If not, split or remove a story.

Ally
Stories (p. 255)

Similarly, if you feel that you can commit to doing more, add a story to the plan.

Continue to adjust the plan until the team is ready to commit to delivering its stories. With experience, you should be able to make plans that don't need adjustment.

The Commitment Ceremony

Commitment is a bold statement. It means that you're making a promise to your team and to stakeholders to deliver all the stories in the iteration plan. It means that you think the plan is achievable and that you take responsibility, as part of the team, for delivering the stories.

Hold a little ceremony at the end of the iteration planning meeting. Gather the whole team together—customers, testers, and programmers—and ask everyone to look at the stories. Remind everybody that the team is about to commit to delivering these stories at the end of the iteration. Ask each person, in turn, if he can commit to doing so. Wait for a verbal "yes."

> Openly discuss problems without pressuring anybody to commit.

It's OK to say "no." If anybody seems uncomfortable saying "yes" out loud, remind them that "no" is a perfectly fine answer. If somebody does say no, discuss the reason as a team and adjust the plan accordingly.

Commitment is important because it helps the team consistently deliver iterations as promised, which is the best way to build trust in the team's ability. Commitment gives people an opportunity to raise concerns before it's too late. As a pleasant side effect, it helps the team feel motivated to work together to solve problems during the iteration.

Ally
Trust (p. 103)

After the Planning Session

After you finish planning the iteration, work begins. Decide how you'll deliver on your commitment. In practice, this usually means that programmers volunteer to work on a task and ask for someone to pair with them. As pairs finish their tasks, they break apart. Individuals pick up new tasks from the board and form new pairs.

Ally

Pair Programming (p. 74)

Other team members each have their duties as well. This book assumes that programmers are the constraint in the system (see "XP Concepts"" in Chapter 3 for more about the Theory of Constraints), so other team members may not have a task planning board like the programmers do. Instead, the customers and testers keep an eye on the programmers' progress and organize their work so it's ready when the programmers need it. This maximizes the productivity of the whole team.

As work continues, revise the iteration plan to reflect the changing situation. (Keep track of your original task and story estimates, though, so you can use them when you plan the next iteration.) Remember that your commitment is to deliver *stories*, not tasks, and ask whether your current plan will succeed in that goal.

> Every team member is responsible for the successful delivery of the iteration's stories.

At the end of the iteration, release your completed software to stakeholders. With a good 10-minute build, this shouldn't require any more than a button press and a few minutes' wait. The following morning, start a new iteration by demonstrating what you completed the night before.

Ally

Ten-Minute Build (p. 177)

Dealing with Long Planning Sessions

Iteration planning should take anywhere from half an hour to four hours. Most of that time should be for discussion of engineering tasks. For established XP teams, assuming they start their iteration demo first thing in the morning, planning typically ends by lunchtime.

New teams often have difficulty finishing planning so quickly. This is normal during the first several iterations. It will take a little while for you to learn your problem space, typical approaches to design problems, and how best to work together.

If iteration planning still takes a long time after the first month or so, look for ways to speed it up. One common problem is spending too much time doing release planning during the iteration planning meeting. Most release planning should happen during the previous iteration, primarily among customers, while the programmers work on stories. Picking stories for the iteration plan should be a simple matter of taking stories from the front of the release plan. It should only take a few minutes because you won't estimate or discuss priorities.

NOTE

The team may have a few new stories as a result of stakeholder comments during the iteration demo. In general, though, the customers should have already prioritized most of the stories.

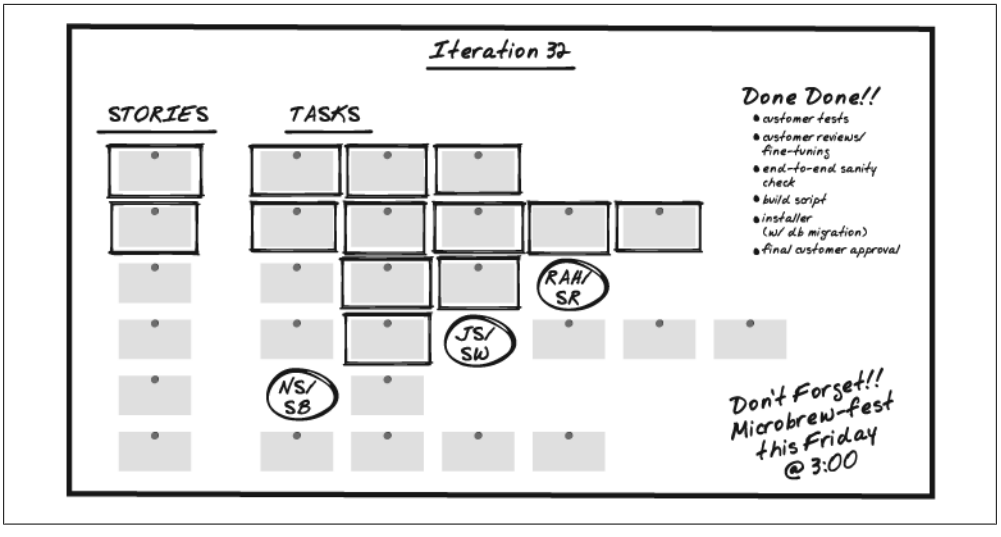

Figure 8-9. An iteration planning board

Long planning sessions also result from spending a lot of time trying to break down the stories into engineering tasks. This may be a result of doing too much design work. Although iteration planning *is* a design activity, it's a very high-level one. Most of the real design work will happen during the iteration as pairs work on specific tasks. If you spend much time discussing possible design details, ask yourself whether you really need to solve these problems in order to come up with good engineering tasks.

If you find that team members don't understand the existing design, or if you have long discussions about how it works, you may lack shared design understanding. Remedy this problem with collective code ownership and more pair programming.

Allies

Collective Code Ownership (p. 191)

Pair Programming (p. 74)

If you find yourselves speculating about possible design choices, your problem may be a result of trying to make your design too general. Remember to keep the design simple. Focus on the requirements that you have today. Trust pairs doing test-driven development to make good decisions on their own.

Design speculation can also occur when you don't understand the requirements well. Take advantage of the on-site customer in your meeting. Ask him to explain the details of each story and why the system needs to behave in a particular way.

Allies

Simple Design (p. 316)

Test-Driven Development (p. 287)

Tracking the Iteration

Like your release plan, your iteration plan should be a prominent part of your informative workspace. Put your stories and tasks on a magnetic whiteboard, as shown in Figure 8-9. When you start work on a task, take it off of the whiteboard and clip it to your workstation. (Mark your initials on the whiteboard so people know where the task went.) As you finish each task, put it back on the board and circle it with a green marker.

Ally

Sit Together (p. 113)

Ally

Informative Workspace (p. 86)

One of the difficulties in iteration planning is identifying that things are going wrong in time to fix them. I take a brief look at our progress every day. Is there a task that's been in progress for more than a day? It might be a problem. If it's halfway through the iteration, are about half the cards marked green? If not, we might not finish everything on time.

After the iteration ends, take your cards down from the board, add a card on top with some vital statistics (iteration number, dates, whatever else you think is relevant), clip them together, and file them in a drawer. Alternatively, you can just throw them away. You're unlikely to come back to the cards, but most teams prefer archiving them just in case.

When Things Go Wrong

Does making a big deal out of commitment mean that you always deliver everything as promised? No, of course not. Commitment is about working around problems and doing what's necessary to deliver the iteration's stories—but sometimes a problem comes up that you can't work around.

When you discover a problem that threatens your iteration commitment, first see if there's any way you can change your plan so that you still meet your commitments. Would using some of your iteration slack help? Is there an engineering task that you can simplify or postpone? Discuss your options as a team and revise your plan.

Ally

Slack (p. 247)

Sometimes the problem will be too big to absorb. In this case, you'll usually need to reduce the scope of the iteration. Typically, this involves splitting or removing a story. As a team, discuss your options and make the appropriate choice.

> NOTE
>
> **Always stay in control of your iteration, even if you have to change your plan to do so. Any iteration that delivers all of the stories in the current plan—even if you changed the plan—is a success.**
>
> **Under no circumstances, however, should you change the iteration deadline.**

After changing the plan, the customers should reestablish trust with stakeholders by explaining what happened, why, and what the team is doing to prevent this sort of problem in the future.

Despite your best efforts, you may have a bad week and end up with nothing at all to demonstrate to your stakeholders. Some teams declare a *lost iteration* when this happens. They roll back their code and use their previous velocity as if the lost iteration never happened. Every team makes mistakes, so this is a fine approach as long as it happens rarely (less than once per quarter). If it happens more often, something is wrong. Ask your mentor for help.

Partially Done Work

At the end of the iteration, every story should be "done done." Partially completed stories should be rare: they reflect a planning problem. That said, they will happen occasionally, particularly when you're new to XP.

Ally

"Done Done" (p. 155)

Some teams think that the best way to handle partially completed stories is to delete all the code for uncompleted stories and deliver only what's completely done. This

sounds harsh, but it's a good idea. "With true timeboxing, the software is either accepted or thrown away at the timebox deadline. That makes it clear that the quality level must be acceptable at all times. The success of timeboxing depends on being able to meet tight schedules by limiting the product's scope, not its quality" [McConnell 1996] (p. 581).

If you follow this practice, you probably won't throw away much code—the iteration is only a week long. Starting fresh may require you to rewrite code, but you'll retain everything you learned when you wrote it the first time. The second attempt will often produce better code and you may even finish more quickly.

> You may delete code, but you won't delete what you've learned.

If you think this is extreme, as long as you know you will definitely work on that story during the next iteration, it's OK to keep the code. However, if you will not *immediately* continue to work on that story, it's best to delete the code. Get it out of your way. It's technical debt and baggage. If you ever need it, you can always get it out of version control.

Ally

Version Control (p. 169)

Emergency Requests

It's inevitable: you're on the third day of the iteration, everything is going well, and a customer comes up to you and says, "Pat, we really need to get this story in." What do you do?

As a programmer, it's very tempting to tell your customer to take a hike—right over the edge of a cliff. Changing direction in the middle of an iteration, after all, means an interruption in concentration, delays, and lost work.

On the other hand, responsiveness to business needs is a core agile value, so suppress that homicidal urge, smile, and provide your customer with options.

You can change the iteration schedule under the condition that you take out as much work as you add. In other words, if you're adding a two-point story to the plan, a two-point story needs to come out of the plan.

In addition, you may only replace stories that you haven't started yet. A two-point story that's half-done isn't a two-point story anymore—but it's not a one-point story, either. It's too difficult to tell how much work you have left on a story until it's "done done," and replacing it will lead to technical debt in the form of half-finished code.

Before making any changes, however, remember that your next planning meeting is less than a week away. Rather than inserting chaos into the team's work, ask yourself how much of an emergency you're really facing. Maybe it can wait until the next planning meeting. After all, it's only a day or three away.

The Batman

Dealing with an emergency request every now and then is fine—it's a great way for the team to be responsive. On the other hand, an emergency in every iteration means that something is wrong.

After the second or third iteration in a row with an emergency request, take a look at what's happening. Perhaps your on-site customers need to be more disciplined about release planning.

Perhaps stakeholders need stronger reminders that requests can wait until the next iteration. Often the requests will die down as your organization adapts to the iteration heartbeat.

In some cases, however, the team has a legitimate need to provide ongoing support for ad hoc requests. If this is true for your team, sacrifice a programmer to be the batman.

> NOTE
> **Some teams have a dedicated phone for emergency support requests; this is, of course, the bat-phone.**

"Batman" is a military term as well as a comic book character: it refers to a soldier assigned to deal with chores so that officers can focus on officering. On an XP team, the *batman* deals with organizational emergencies and support requests so the other programmers can focus on programmering. The batman has no other duties: he doesn't work on stories or the iteration plan.

Rotate a new programmer into the batman role every iteration to prevent burn-out. If the load is particularly high, you may need two or more batmen per iteration.

Depending on your situation, you may be better off using daily iterations rather than a batman. A daily iteration allows you to postpone all emergencies until the next morning, which enables the team to focus better (see the "Daily Iterations"" sidebar). It's appropriate for teams that primarily deal with small ad hoc issues, such as bug fixes and minor enhancements, and don't have a long-term release plan.

DAILY ITERATIONS

If you have a particularly chaotic environment, you probably won't be able to use a lot of the XP practices. However, iterations—particularly daily iterations—can be a great way to bring structure to this environment.

One team I worked with was struggling under a crushing load of support requests. They had resorted to *firefighting*: they responded to whichever request seemed like the biggest emergency.

We instituted a simple daily iteration—nothing else—in which the team prioritized outstanding support requests using the planning game. They deferred any new requests that came in during the day until the next planning game. That was acceptable because the team planned every morning.

Ally

The Planning Game (p. 221)

The results were remarkable. Productivity increased, morale shot up, and the team actually found itself with free time. Daily iterations helped the team tame the chaos and, in so doing, dramatically improved their effectiveness.

If you find yourself struggling with firefighting, daily iterations might help your team, too.

Questions

How should we schedule time for fixing bugs?

You should fix bugs as soon as you find them, preferably as you work on each task. This time is part of the overhead of your iteration. Don't batch them up for fixing later, even if "later" is as soon as the end of the iteration.

Ally
No Bugs (p. 159)

Some bugs will be too big to absorb into your iteration slack. Create story cards for these and schedule them as soon as possible—or decide that they aren't worth fixing at all.

Ally
Slack (p. 247)

If we don't estimate stories during iteration planning, when do we estimate stories?

Estimate new stories as they appear throughout the iteration. If this is too much of an interruption, batch up the stories and estimate them at a particular time every day.

If you have a lot of stories to estimate, as often happens near the beginning of a project, schedule time for estimation with a story.

All the available tasks depend on tasks that other pairs are working on right now. What should I work on?

This happens less frequently than you might think; breaking stories into engineering tasks helps modularize the design. However, it does happen.

It's OK to have two pairs working on the same class. In this case, discuss the issue and come to agreement about the names of the class and the public methods that both pairs will be using. Then pick one pair to write the class. The other pair creates the exact same class but, instead of writing real code, they stub in some hardcoded return values.

When you integrate, replace the fake class with the real one and make sure the tests still pass.

What should the batman do when there are no outstanding support requests?

Whatever she likes, as long as she can easily put it aside when a support request comes in. Don't try to squeeze every last drop of efficiency out of the batman; doing so will likely slow the team down and make the batman's job even more tedious.

Results

When you use iterations well, your team has a consistent, predictable velocity. Stakeholders know what to expect from the team and trust that it will deliver on its commitments. The team discovers mistakes quickly and deals with them while still meeting its commitments. Rarely, the team meets its commitments by replanning, changing its commitments, and communicating these changes to stakeholders.

Contraindications

XP's iterations assume the use of customer-centric stories. To successfully deliver software in such a short timescale, the team must use simultaneous phases as well as simple design and incremental design and architecture. If you don't use these practices, XP-style iterations probably won't work for you.

Allies

Stories (p. 255)
Simple Design (p. 316)
Incremental Design and Architecture (p. 323)

In order to achieve a consistent velocity and deliver on commitments, your iteration must include slack. Never artificially inflate your velocity. Similarly, don't use commitment as a club. Never force team members to commit to a plan they don't agree with.

Ally

Slack (p. 247)

Energized work is also important. Without it, the team will have trouble maintaining equilibrium and a stable velocity.

Finally, there's little value to a strict iteration schedule unless you pay close attention to the feedback cycles of velocity, planning, and frequent releases. A disciplined iteration schedule may improve predictability and estimation, but you must notice and react to changes in order to take advantage of it.

Ally

Energized Work (p. 82)

Alternatives

Ally

Release Early, Release Often (p. 208)

Some methods use longer iterations that release to real customers, not just to internal stakeholders, at the end of each iteration. Other methods use independent phases instead of simultaneous phases and iterations. Either of these approaches can work, but most of XP's practices assume the presence of short iterations. If you don't use XP-style iterations, talk with your mentor about whether XP will work in your situation.

Some established XP teams don't use engineering tasks at all. Instead, they use very small stories that can each be finished in less than a day. This approach works best for established products. Other teams use engineering tasks but don't estimate them. Either approach can work well, but they're advanced techniques. I recommend explicitly creating and estimating engineering tasks to start.

Iteration length

Throughout this book, I've assumed that your team uses one-week iterations. However, iterations may be of any length. Many teams prefer two-week iterations.

I've found that shorter iteration lengths are better for teams new to XP. Teams seem to mature in their understanding of XP based on how many *iterations* they've undertaken rather than how many *weeks* they've experienced. As a result, shorter iterations means more rapid improvement for a team new to XP.

Short iterations allow the team to practice core XP skills more frequently. A short iteration leads to more planning, more internal releases, more iteration demos, and more retrospectives. They reduce the team's ability to use overtime to cover up scheduling flaws, which helps the team learn to estimate and plan well.

One-week iterations also make decisions easier by reducing schedule risk. If a discussion is taking too long, you can say, "This may not be perfect, but we'll review it again next week." This makes planning and retrospectives easier.

On the other hand, one-week iterations put more pressure on the team. This makes energized work more difficult and can limit refactoring. Velocity is less stable in one-week iterations, because even one holiday represents a big loss of time for the iteration.

Allies

Energized Work (p. 82)
Refactoring (p. 306)

I prefer one-week iterations for new teams. For established teams that are comfortable with all the XP practices, I prefer two-week iterations. Two-week iterations are a little less stressful and lead to a more stable velocity.

Three and four-week iterations seem too long to me. They don't provide enough feedback to the team or the larger organization. However, if you think that a longer iteration would be useful for your team, please try it. Be careful: longer iterations are more sensitive to mistakes, because it takes longer to expose and to recover from those mistakes.

Don't use longer iterations if you feel that you need more time to get your work done. Longer iterations won't change the amount of time you have; they only change how often you check your progress. If you have difficulty getting your work done, *shorten* your iteration length (to a minimum of one week) and look at what you're doing to support energized work. Be sure to reduce your workload in each iteration proportionally. Shortening your iteration length will reduce the amount of work you have to do each iteration and will help you identify problems in your process.

> Shorten your iteration length if you're having trouble with XP.

Ally

Energized Work (p. 82)

Some teams base their iterations on a number of business days rather than a calendar. For example, rather than having a seven calendar-day iteration, the team's iterations are five business days long. This is helpful for one-week iterations because it reduces the impact of holidays on the team's velocity. However, I don't recommend business-day iterations because they're harder to schedule with stakeholders. The regular heartbeat of the XP team is an excellent way to generate trust, and business-day iterations don't have the same impact. It's nice to know that Wednesday is *always* the start of a new iteration.

Ally

Trust (p. 103)

Further Reading

Agile Estimating and Planning [Cohn] and *Planning Extreme Programming* [Beck & Fowler] each provide alternative ways of approaching iteration planning.

Slack

Audience

Programmers, Coaches

We deliver on our iteration commitments.

Imagine that the power cable for your workstation is just barely long enough to reach the wall receptacle. You can plug it in if you stretch it taut, but the slightest vibration will cause the plug to pop out of the wall and the power to go off. You'll lose everything you were working on.

Ally

Iterations (p. 42)

I can't afford to have my computer losing power at the slightest provocation. My work's too important for that. In this situation, I would move the computer closer to the outlet so that it could handle some minor bumps. (Then I would tape the cord to the floor so

people couldn't trip over it, install an uninterruptable power supply, and invest in a continuous backup server.)

Your project plans are also too important to be disrupted by the slightest provocation. Like the power cord, they need slack.

How Much Slack?

The amount of slack you need doesn't depend on the number of problems you face. It depends on the *randomness* of problems. If you always experience exactly 20 hours of problems in each iteration, your velocity will automatically compensate. However, if you experience between 20 and 30 hours of problems in each iteration, your velocity will bounce up and down. You need 10 hours of slack to stabilize your velocity and to ensure that you'll meet your commitments.

These numbers are just for illustration. Instead of measuring the number of hours you spend on problems, take advantage of velocity's feedback loop (see "Estimating" later in this chapter for more about velocity). If your velocity bounces around a lot, stop signing up for more stories than your velocity allows. This will cause your velocity to settle at a lower number that incorporates enough slack for your team. On the other hand, if your velocity is rock solid, try reducing slack by committing to a small extra story next iteration.

How to Introduce Slack

One way to introduce slack into your iterations might be to schedule no work on the last day or two of your iteration. This would give you slack, but it would be pretty wasteful. A better approach is to schedule useful, important work that isn't time-critical—work you can set aside in case of an emergency. *Paying down technical debt* fits the bill perfectly.

> NOTE
>
> **Only the constraint needs slack. The rest of the team organizes their work around the constraint's schedule, resulting in slack for the entire team. ("XP Concepts" in Chapter 3 discusses the Theory of Constraints in more detail.)**
>
> **In this book, I've assumed that programmers are your team's constraint. If that isn't true for your team, you will need slack that is appropriate for your constraint. Talk to your mentor (see "Find a Mentor" in Chapter 2) about how to modify this advice for your specific situation.**

Even the best teams inadvertantly accumulate technical debt. Although you should always make your code as clean as you can, some technical debt will slip by unnoticed.

Rather than doing the bare minimum necessary to keep your head above water, be generous in refactoring and cleaning up technical debt in existing code. Every iteration, look for opportunities to make existing code better. Make this part of your everyday work. Every time I find myself scratching my head over a variable or method name, I change it. If I see some code that's no longer in use, I delete it.

Ally

Refactoring (p. 306)

In addition to these small improvements, look for opportunities to make larger changes. Perhaps the code uses primitives rather than introducing a new type, or perhaps a class needs to have some of its responsibilities extracted into a new class.

> NOTE
> **I notice technical debt most when I navigate during pair programming. When a problem slows us down and I find myself feeling irritated, that inspires me to suggest that we fix it.**

Paying down technical debt directly increases team productivity, so I spend a lot of time on it throughout the iteration. I usually spend about eight hours per week paying down technical debt, but other members of my teams spend only a few hours. A good rule of thumb is to spend 10 percent of the iteration on technical debt.

> Perform big refactorings incrementally.

Don't spend all your time on a single problem. Refactor throughout the iteration—an hour encapsulating a structure here, two hours fixing class responsibilities there. Each refactoring should address a specific, relatively small problem. Sometimes you'll fix only part of a larger problem—that's OK as long as it makes the code better. If your team pays down technical debt every week, you'll have the opportunity to see and fix remaining problems in the future.

As you fix technical debt, focus on fixes that make your current work easier. Don't go looking for technical debt that's unrelated to stories you're currently working on. If you consistently relate your improvements to your current work, you'll automatically put the most effort into the most frequently modified and most valuable parts of the system.

Research Time

To keep up with their constantly expanding field, programmers must continually improve their skills. In doing so, they will often learn things that enhance their work on the project.

Dedicated *research time* is an excellent way to encourage learning and add additional slack into your iterations. To introduce it, set aside half a day for each programmer to conduct self-directed research on a topic of his choice. Be completely hands-off. I recommend only two rules: don't spend this time on project stories or tasks, and don't modify any project code.

> NOTE
> **If you're concerned about people goofing off, provide lunch the next day and ask that people share what they've done in informal peer discussion. This is a good way to share knowledge anyway.**

I've introduced this technique to several teams, and it's paid dividends each time. Two weeks after introducing research time at one organization, the product manager told me that research time was the most valuable time the team spent, and suggested that we double it.

Research time works because programmers are typically motivated by a desire to do good work, particularly when they're self-directed. Most programmers have a natural desire to make their

lives easier and to impress their colleagues. As a result, the work done in research time often has a surprisingly high return for the project.

Research time is particularly valuable for XP teams. The continuous drumbeat of iteration deadlines is great for reducing risk and motivating the team to excel, but it can lead to tunnel vision. Dedicated research time gives programmers a chance to widen their ranges, which often leads to insights about how to develop more effectively.

> NOTE
> **I schedule research time for the morning of the penultimate day of the iteration. This is late enough in the iteration that we can use the time as slack if we need to, but not so late to distract programmers with the upcoming deadline. Mornings are better than afternoons because it's harder to start on time when production code is occupying your attention.**

For research time to be effective, you must focus. Half a day can go by very quickly. It's easy to think of research time as a catch-all for postponed meetings. Be strict about avoiding interruptions and enlist the help of the project manager. Ignore your email, turn off your IM, and use your web browser only for specific research.

When you first adopt research time, you might have trouble deciding what to work on. Think about what has puzzled you recently. Would you like to learn more about the details of your UI framework? Is there a programming language you've wanted to try but your organization doesn't use? Has real-time networking always fascinated you?

As you do your research, create spike solutions—small standalone programs—that demonstrate what you've learned. Avoid trying to make software that's generally useful; that will reduce the amount of time available to pursue core ideas. If something turns out to deserve further work, create and schedule a story for it.

Ally

Spike Solutions (p. 333)

When Your Iteration Commitment Is at Risk

Research time and paying down technical debt are important tasks that enhance your programmers' skills and allow them to deliver more quickly. Paying down technical debt, in particular, should be part of every iteration. However, if your iteration commitments are at risk, it's OK to set these two tasks aside *temporarily* in order to meet those commitments.

Use refactoring as a shock absorber

Before starting a big refactoring to pay down technical debt, consider the iteration plan and think about the kinds of problems the team has encountered. If the iteration is going smoothly, go ahead and refactor. If you've encountered problems or you're a little behind schedule, shrug your shoulders and work on meeting your iteration commitment instead. You'll have another opportunity to fix the problem later. By varying the amount of time you spend paying down technical debt, you can ensure that most iterations come in exactly on time.

NOTE

Continue refactoring *new* code as you write it. It's OK to defer cleaning up existing technical debt temporarily, but incurring new technical debt will hurt your productivity.

Incur a little voluntary overtime

Every iteration experiences a few bumps. Varying the time you spend paying down technical debt is your first line of defense. Rarely, though, a problem sneaks past. In this case, if family commitments permit, I voluntarily work a few extra hours. Overtime is unusual in the companies I work with, so an extra hour a day once in a while isn't a big deal. Another programmer or two will often volunteer to do the same.

Ally

Energized Work (p. 82)

NOTE

Be careful not to overuse overtime. Overuse will sap team members' ability to do good work. You can use it to clean up after small problems, but each programmer should only contribute an hour or so per day, and only voluntarily.

Cancel research time

Some problems will be too significant to clean up in overtime. In this case, consider cancelling research time in order to address the problem. First, though, take a second look at whether you need to replan the iteration instead (see "Iteration Planning"" earlier in this chapter). When problems are big, even cancelling research time may not be enough.

Don't Cross the Line

Slack is a wonderful tool. It helps you meet your commitments and gives you time to perform important, nonurgent tasks that improve your productivity.

Be careful, though. Although slack is a wonderful way to handle transitory problems, it can also disguise systemic problems. If you rely on slack to finish your stories in every iteration, that time isn't really slack—it's required.

To return to the power cord example, suppose that your workstation is on a jiggly desk. Every time you type, the desk jiggles a bit and the power cord pops out. You could add just enough slack to the cord to handle your typing, but that wouldn't really be slack; it's necessary to use the computer at all. If anything else happens while you're typing—say, Bob from marketing gives you a friendly slap on the back—the cord will still pop out.

If you work overtime, cancel research time, or don't pay down any technical debt for two or three iterations in a row, you've overcommitted and have no slack. Congratulate yourself for delivering on your commitments anyway. Now add slack by reducing your velocity.

> If you consistently use most of your slack, you've overcommitted.

Making enough time for these nonurgent tasks is difficult. With a deadline rushing head-first toward you, it's difficult to imagine spending any time that doesn't directly contribute to getting stories out the door. New XP teams especially struggle with this. Don't give up. Slack is essential to meeting commitments, and that is the key to successful XP.

Reducing the Need for Slack

In my experience, there are two big sources of randomness on XP teams: customer unavailability and technical debt. Both lead to an unpredictable environment, make estimating difficult, and require you to have more slack in order to meet your commitments.

If programmers have to wait for customer answers as they work, you can reduce the need for slack by making customers more available to answer programmer questions. They may not like that—customers are often surprised by the amount of time XP needs from them—but if you explain that it will help improve velocity, they may be more interested in helping.

On the other hand, if programmers often encounter unexpected *technical* delays, such as surprising design problems, difficulty integrating, or unavailability of a key staging environment, then your need for slack is due to too much technical debt. Fortunately, using your slack to pay down technical debt will automatically reduce the amount of slack you need in the future.

Questions

If our commitment is at risk, shouldn't we stop pair programming or using test-driven development? Meeting our commitment is most important, right?

Allies

Pair Programming (p. 74)
Test-Driven Development (p. 287)

First, pair programming and test-driven development (TDD) should allow you to deliver more quickly, not more slowly. However, they do have a learning curve—particularly TDD—so it's true that avoiding TDD might allow you to meet your commitments in the early stages of your project.

However, you shouldn't use them as slack. Pair programming, test-driven development, and similar practices maintain your capability to deliver high-quality code. If you don't do them, you will immediately incur technical debt and hurt your productivity. You may meet this iteration's commitments, but you'll do so at the expense of the next iteration. If your existing slack options aren't enough, you need to replan your iteration, as discussed in"Iteration Planning"" earlier in this chapter.

In contrast, paying down technical debt, going home on time, and conducting research *enhance* your capability to deliver. Using them as slack once in a while won't hurt.

Research time sounds like professional development. Shouldn't programmers do that on their own time?

In my experience, research time pays dividends within a few months. It's worthwhile even without an explicit policy of encouraging professional development.

However, if research time isn't appropriate for your organization, you can increase the amount of time you spend paying down technical debt instead.

Should we pair program during research time?

You can if you want, but you don't have to. Treat it as you would any other spike.

Ally

Spike Solutions (p. 333)

Shouldn't a project have just one buffer at the end of the project rather than a lot of little buffers?

The book *Critical Chain* [Goldratt 1997] argues for creating a single buffer at the end of a project rather than padding each estimate. It's good advice, and adding slack to each iteration might seem to contradict that.

However, an XP team makes a commitment to deliver every week. In a sense, an XP team conducts a series of one-week projects, each with a commitment to stakeholders and its own small buffer protecting that commitment. This commitment is necessary for stakeholder trust, to take advantage of feedback, and to overcome the inherently high risk of software projects. Without it, the project would drift off course.

XP avoids *Parkinson's Law* ("work expands to fill the time available") and *Student Syndrome* ("work is delayed until its deadline") by performing important but nonurgent tasks in the buffer.

Results

When you incorporate slack into your iterations, you consistently meet your iteration commitments. You rarely need overtime. In addition, by spending so much time paying down technical debt, your code steadily improves, increasing your productivity and making further enhancements easier.

Contraindications

The danger of thinking of these tasks as slack is thinking they aren't important. They're actually *vital*, and a team that doesn't perform these tasks will slow down over time. They're just not time-critical like your iteration commitment is. Don't use slack as an excuse to set aside these tasks indefinitely.

In addition, never incur technical debt in the name of slack. If you can't meet your iteration commitments while performing standard XP tasks, replan the iteration instead. Practices you should never use as slack include test-driven development, refactoring new code, pair programming, and making sure stories are "done done."

Allies

Test-Driven Development (p. 287)
Refactoring (p. 306)
Pair Programming (p. 74)
"Done Done" (p. 155)

Finally, don't use try to use iteration slack to meet release commitments. There isn't enough iteration slack to make a meaningful difference to your release commitments—in fact, removing slack from your iterations could easily add so much chaos to your process that productivity actually goes down. Use risk management to provide slack for your release commitments.

Ally

Risk Management (p. 226)

Alternatives

Slack allows you to be "done done" and meet your iteration commitments. It enables a consistent velocity. It provides an opportunity for extra refactoring. It reduces technical debt and increases team capability. I'm not aware of any alternatives that provide all these benefits.

Rather than including slack at the iteration level, some teams add a safety buffer to every estimate. As [Goldratt 1997] explains, this leads to delays and often doesn't improve the team's ability to meet their commitments.

Many organizations, however, choose not to include any slack in their plans. It's no coincidence that these organizations have trouble meeting commitments. They also waste a lot of time thrashing around as they attempt to achieve the unachievable.

Reading groups

Some teams form a reading group rather than conducting research time. This is an excellent alternative to research time, particularly for teams that are interested in discussing fundamental ideas rather than exploring new technologies.

One way to do so is to take turns reading sections of a book or an article out loud and then discussing them. This approach has the advantage of not requiring advance preparation, which promotes more active participation. One person should act as facilitator and inspire discussion by asking questions about each section.

A reading group can easily spend half a day per week in productive conversation. To take this book as an example, you could probably discuss one or two practice sections per session.

Silver stories

Other teams schedule *silver stories* for slack. These are less-important stories that the team can set aside if they need extra time. I prefer to pay down technical debt instead because teams often neglect this crucial task.

The difficulty with silver stories is that you need to set aside slack tasks at a moment's notice. If you do that to a silver story, you'll leave behind technical debt in the form of half-done code. If you use silver stories, be sure to delete all code related to the story if you have to put it aside.

I've also observed that silver stories hurt programmer morale. If the team needs to use some slack, they don't finish all the stories on the board. Even though the silver stories were bonus stories, it doesn't feel like that in practice. In contrast, *any* time spent paying down technical debt directly improves programmers' quality of life. That's a win.

Further Reading

Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency [DeMarco 2002] provides a compelling case for providing slack throughout the organization.

The Goal and *Critical Chain* [Goldratt 1992] and [Goldratt 1997] are two business novels that make the case for using slack (or "buffers"), instead of padding estimates, to protect commitments and increase throughput.

"Silicon Valley Patterns Study of Domain-Driven Design" (*http://domaindrivendesign.org/discussion/siliconvalleypatterns/index.html*) is an interesting transcript of some meetings of a long-running and highly successful reading group.

Stories

Audience
Whole Team

We plan our work in small, customer-centric pieces.

Stories may be the most misunderstood entity in all of XP. They're not requirements. They're not use cases. They're not even narratives. They're much simpler than that.

Stories are for planning. They're simple one- or two-line descriptions of work the team should produce. Alistair Cockburn calls them "promissory notes for future conversation."* Everything that stakeholders want the team to produce should have a story, for example:

Ally

Release Planning (p. 207)

- "Warehouse inventory report"
- "Full-screen demo option for job fair"
- "TPS report for upcoming investor dog-and-pony show"
- "Customizable corporate branding on user login screen"

This isn't enough detail for the team to implement and release working software, nor is that the intent of stories. A story is a placeholder for a detailed discussion about requirements. Customers are responsible for having the requirements details available when the rest of the team needs them.

Ally

Incremental Requirements (p. 275)

Although stories are short, they still have two important characteristics:

1. Stories represent *customer value* and are written in the customers' terminology. (The best stories are actually written by customers.) They describe an end-result that the customer values, not implementation details.
2. Stories have clear *completion criteria*. Customers can describe an objective test that would allow programmers to tell when they've successfully implemented the story.

The following examples are *not* stories:

- "Automate integration build" does not represent customer value.
- "Deploy to staging server outside the firewall" describes implementation details rather than an end-result, and it doesn't use customer terminology. "Provide demo that customer review board can use" would be better.
- "Improve performance" has no clear completion criteria. Similarly, "Make it fast enough for my boss to be happy" lacks *objective* completion criteria. "Searches complete within one second" is better.

Story Cards

Write stories on index cards.

> NOTE
> **I prefer 3×5-inch cards because they fit into my pocket.**

* *http://c2.com/cgi/wiki?UserStory*.

This isn't the result of some strange Ludditian urge on the part of XP's creators; it's a deliberate choice based on the strengths of the medium. You see, physical cards have one feature that no conglomeration of pixels has: they're tactile. You can pick them up and move them around. This gives them power.

During release planning, customers and stakeholders gather around a big table to select stories for the next release. It's a difficult process of balancing competing desires. Index cards help prevent these disputes by visually showing priorities, making the scope of the work more clear, and directing conflicts toward the plan rather than toward personalities, as discussed in "The Planning Game"" earlier in this chapter.

Ally
Release Planning (p. 207)

Story cards also form an essential part of an informative workspace. After the planning meeting, move the cards to the release planning board—a big, six-foot whiteboard, placed prominently in the team's open workspace (see Figure 8-4). You can post hundreds of cards and still see them all clearly. For each iteration, place the story cards to finish during the iteration on the iteration planning board (another big whiteboard; see Figure 8-9) and move them around to indicate your status and progress. Both of these boards are clearly visible throughout the team room and constantly broadcast information to the team.

Ally
Informative Workspace (p. 86)

Index cards also help you be responsive to stakeholders. When you talk with a stakeholder and she has a suggestion, invite her to write it down on a blank index card. I always carry cards with me for precisely this purpose. Afterward, take the stakeholder and her card to the product manager. They can walk over to the release planning board and discuss the story's place in the overall vision. Again, physical cards focus the discussion on relative priorities rather than on contentious "approved/disapproved" decisions.

> Use index cards to be more responsive to stakeholder suggestions.

If the product manager and stakeholder decide to add the story to the release plan, they can take it to the programmers right away. A brief discussion allows the programmers to estimate the story. Developers write their estimate—and possibly a few notes—on the card, and then the stakeholder and product manager place the card into the release plan.

Ally
Estimating (p. 261)

Physical index cards enable these ways of working in a very easy and natural way that's surprisingly difficult to replicate with computerized tools. Although you *can* use software, index cards just work better: they're easier to set up and manipulate, make it easier to see trends and the big picture, and allow you to change your process with no configuration or programming.

> Index cards are simpler and more effective than computerized tools.

Most people are skeptical about the value of index cards at first, so if you feel that way, you're not alone. The best way to evaluate the value of physical story cards is to try them for a few months, then decide whether to keep them.

Customer-Centricity

Stories need to be *customer-centric*. Write them from the on-site customers' point of view, and make sure they provide something that customers care about. On-site customers are in charge of priorities in the planning game, so if a story has no customer value, your customers won't—and shouldn't—include it in the plan.

Ally
The Planning Game (p. 221)

NOTE
A good way to ensure that your stories are customer-centric is to ask your customers to write the stories themselves.

One practical result of customer-centric stories is that you won't have stories for technical issues. There should be no "Create a build script" story, for example—customers wouldn't know how to prioritize it. Although programmers can tell customers where the technical stories belong in the plan, that disrupts the balance of power over the scope of the project and can leave customers feeling disenfranchised.

Instead, include any technical considerations in the estimate for each story. If a story requires that the programmers create or update a build script, for example, include that cost when estimating for that story.

NOTE
Including technical costs in stories, rather than having dedicated technical stories, requires incremental design and architecture.

Customer-centric stories aren't necessarily always valuable to the end-user, but they should always be valuable to the on-site customers. For example, a story to produce a tradeshow demo doesn't help end-users, but it helps the customers sell the product.

Splitting and Combining Stories

Although stories can start at any size, it is difficult to estimate stories that are too large or too small. Split large stories; combine small ones.

The right size for a story depends on your velocity. You should be able to complete 4 to 10 stories in each iteration. Split and combine stories to reach this goal. For example, a team with a velocity of 10 days per iteration might split stories with estimates of more than 2 days and combine stories that are less than half a day.

> Select story sizes such that you complete 4 to 10 each iteration.

Combining stories is easy. Take several similar stories, staple their cards together, and write your new estimate on the front.

Splitting stories is more difficult because it tempts you away from vertical stripes and releasable stories (discussed in "Release Planning"" earlier in this chapter). It's easiest to just create a new story for each step in the previous story. Unfortunately, this approach leads to story clumps. Instead, consider the *essence* of the story. Peel away all the other issues and write them as new

stories. [Cohn] has an excellent chapter on how to do this in his book, *Agile Estimating and Planning*. He summarizes various options for splitting stories:

- Split large stories along the boundaries of the data supported by the story.
- Split large stories based on the operations that are performed within the story.
- Split large stories into separate CRUD (Create, Read, Update, Delete) operations.
- Consider removing cross-cutting concerns (such as security, logging, error handling, and so on) and creating two versions of the story: one with and one without support for the cross-cutting concern.
- Consider splitting a large story by separating the functional and nonfunctional (performance, stability, scalability, and so forth) aspects into separate stories.
- Separate a large story into smaller stories if the smaller stories have different priorities.

Special Stories

Most stories will add new capabilities to your software, but any action that requires the team's time and is not a part of their normal work needs a story.

Documentation stories

XP teams need very little documentation to do their work (see "Documentation"" in Chapter 7), but you may need the team to produce documentation for other reasons. Create documentation stories just like any other: make them customer-centric and make sure you can identify specific completion criteria. An example of a documentation story is "Produce user manual."

Ally
Documentation (p. 195)

"Nonfunctional" stories

Performance, scalability, and stability—so-called *nonfunctional* requirements—should be scheduled with stories, too. Be sure that these stories have precise completion criteria. See "Performance Optimization"" in Chapter 9 for more.

Ally
Performance Optimization (p. 336)

Bug stories

Ideally, your team will fix bugs as soon as they find them, before declaring a story "done done." Nobody's perfect, though, and you will miss some bugs. Schedule these bugs with story cards, such as "Fix multiple-user editing bug." Schedule them as soon as possible to keep your code clean and reduce your need for bug-tracking software.

Allies
No Bugs (p. 159)
"Done Done" (p. 155)

Bug stories can be difficult to estimate. Often, the biggest timesink in debugging is figuring out what's wrong, and you usually can't estimate how long that will take. Instead, provide a timeboxed estimate: "We'll spend up to a day investigating this bug. If we haven't fixed it by then, we'll schedule another story."

Spike stories

Sometimes programmers won't be able to estimate a story because they don't know enough about the technology required to implement the story. In this case, create a story to research that technology. An example of a research story is "Figure out how to estimate 'Send HTML' story." Programmers will often use a *spike solution* (see "Spike Solutions"" in Chapter 9) to research the technology, so these sorts of stories are typically called *spike stories*.

Ally

Spike Solutions (p. 333)

Word these stories in terms of the goal, not the research that needs to be done. When programmers work on a research story, they only need to do enough work to make their estimate for the real story. They shouldn't try to figure out all the details or solve the entire problem.

Programmers can usually estimate how long it will take to *research* a technology even if they don't know the technology in question. If they can't estimate the research time, timebox the story as you do with bug stories. I find that a day is plenty of time for most spike stories, and half a day is sufficient for most.

Estimating

Other than spike stories, you normally don't need to schedule time for the programmers to estimate stories—you can just ask them for an estimate at any time. It's part of the overhead of the iteration, as are support requests and other unscheduled interruptions. If your programmers feel that estimating is too much of an interruption, try putting new story cards in a pile for the programmers to estimate when it's convenient.

Sometimes you'll have a large number of new stories to estimate. In this case, it might be worth creating a story card for estimating those stories.

Meetings

Like estimating, most meetings are part of the normal overhead of the iteration. If you have an unusual time commitment, such as training or an off-site day, you can reserve time for it with a story.

Architecture, design, refactoring, and technical infrastructure

Don't create stories for technical details. Technical tasks are part of the cost of implementing stories and should be part of the estimates. Use incremental design and architecture to break large technical requirements into small pieces that you can implement incrementally.

Ally

Incremental Design and Architecture (p. 323)

Questions

Are index cards really more effective than computerized tools? What about reporting?

See our discussion of reporting ("Reporting"" in Chapter 6) for more information.

What about backups? Won't the cards get lost?

If you exercise reasonable care, you're unlikely to lose cards. The only time I've seen a team lose their cards was when they created them and then left them in an ignored pile somewhere on their boss' desk for six months.

That said, there's always the possibility of fire or other disaster. If the risk of losing cards is too great for you, consider taking a digital photo of the release planning board every week or so. An eight-megapixel camera has sufficient resolution to capture a six-foot whiteboard and all the detail on the cards.

Why do you recommend sizing stories so that we can complete 4 to 10 stories per iteration?

If you only have a few stories in an iteration, it's harder to see that you're making progress, which increases the risk that you won't see problems in time to correct them. In addition, if something goes wrong and you can't finish a story, it will represent a large percentage of your work. Too many stories, on the other hand, increase your planning and estimating burden. Each story becomes smaller, making it harder to see the big picture.

An average of 6 stories per iteration leads to a 3-month release plan (the maximum I recommend) consisting of 78 stories. That's a nice number. It gives you flexibility in planning without overwhelming you with details.

Our customers understand programming. Can we create programmer-centric technical stories?

It's much more difficult to create customer-centric stories than programmer-centric stories, so it's tempting to find excuses for avoiding them. "Our customers don't mind if we have programmer-centric stories" is one such excuse. Try to avoid it.

Even if your customers really do have the ability to prioritize programmer-centric stories, customer-centric stories lead to better plans. Remember, your goal is to create stories that allow you to release the software at any time. Programmer-centric stories usually don't have that property.

If your customers *are* programmers—if you're writing software *for* programmers, such as a library or framework—then your stories should use a programmer-centric language (see "Ubiquitous Language"" in Chapter 6). Even so, they should reflect your customers' perspective. Create stories about your customers' needs, not your plans to fulfill their needs.

How can we encourage stakeholders to use stories for requesting features?

When a stakeholder asks you for a feature, take out an index card and invite him to write it down so it can be scheduled. For electronic requests, an on-site customer should follow up, either by speaking to the requester in person or creating the story card himself.

If stakeholders refuse to use stories, the product manager can manage this relationship by providing stakeholders what they want to see and translating stakeholders' wishes into stories for the team.

Results

When you use stories well, the on-site customers understand all the work they approve and schedule. You work on small, manageable, and independent pieces and can deliver complete features frequently. The project always represents the best possible value to the customer at any point in time.

Contraindications

Stories are no replacement for requirements. You need another way of getting details, whether through expert customers on-site (the XP way) or a requirements document (the traditional way).

Be very cautious of using customer-centric stories without also using most of the XP development practices (see Chapter 9). Customer-centric stories depend on the ability to implement infrastructure incrementally with incremental design and architecture. Without this ability, you're likely to incur greater technical debt.

Ally

Incremental Design and Architecture (p. 323)

Physical index cards are only appropriate if the team sits together, or at least has a common area in which they congregate. Experienced distributed teams often keep physical index cards at the main site and copy the cards into the electronic system. This is an administrative headache, but for these teams, the benefits of physical cards make the added work worthwhile.

Ally

Sit Together (p. 113)

Some organizations are skittish about using informal planning tools. If important members of your organization require a formal Gantt chart, you may need to provide it. Your project manager can help you make this decision. As with a distributed team, you may find it valuable to use physical cards as your primary source, then duplicate the information into the tool.

Alternatives

For teams that don't use stories, the main distinction between stories and the line items in most plans is that stories are customer-centric. If you can't use customer-centric stories for some reason, customers cannot participate effectively in the planning game. This will eliminate one of its primary benefits: the ability to create better plans by blending information from both customers and programmers. Unfortunately, no alternative practice will help.

Another distinctive feature of stories is the use of index cards. Physical cards offer many benefits over electronic tools, but you can still use electronic tools if necessary. Some teams track their stories using spreadsheets, and others use dedicated agile planning tools. None of these approaches, however, provide the benefits of physical index cards.

Further Reading

Agile Estimating and Planning [Cohn] discusses options for splitting stories in Chapter 12.

Estimating

Audience

Programmers

We provide reliable estimates.

Programmers often consider estimating to be a black art—one of the most difficult things they must do. Many programmers find that they consistently estimate too low. To counter this problem, they pad their estimates (multiplying by three is a common approach), but sometimes even these rough guesses are too low.

Are good estimates possible? Of course! You just need to focus on your strengths.

What Works (and Doesn't) in Estimating

One reason estimating is so difficult is that programmers can rarely predict how they will spend their time. A task that requires eight hours of uninterrupted concentration can take two or three days if the programmer must deal with constant interruptions. It can take even longer if the programmer works on another task at the same time.

> Estimate in ideal time.

Part of the secret to making good estimates is to predict the effort, not the calendar time, that a project will take. Make your estimates in terms of *ideal engineering days* (often called *story points*), which are the number of days a task would take if you focused on it entirely and experienced no interruptions.

Using ideal time alone won't lead to accurate estimates. I've asked some teams I've worked with to measure exactly how long each task takes them (one team gave me 18 months of data), and even though we estimated in ideal time, the estimates were never accurate.

Still, they were *consistent*. For example, one team always estimated their stories at about 60 percent of the time they actually needed. This may not sound very promising. How useful can inaccurate estimates be, especially if they don't correlate to calendar time? Velocity holds the key.

Velocity

Although estimates are almost never accurate, they are *consistently* inaccurate. While the estimate accuracy of *individual* estimates is all over the map—one estimate might be half the actual time, another might be 20 percent more than the actual time—the estimates *are* consistent in aggregate. Additionally, although each iteration experiences a different set of interruptions, the amount of time required for the interruptions also tends to be consistent from iteration to iteration.

As a result, you can reliably convert estimates to calendar time if you aggregate all the stories in an iteration. A single scaling factor is all you need.

This is where velocity comes in. Your *velocity* is the number of story points you can complete in an iteration. It's a simple yet surprisingly sophisticated tool. It uses a feedback loop, which means every iteration's velocity reflects what the team actually achieved in the previous iteration.

> NOTE
> **To predict your next iteration's velocity, add the *estimates* of the stories that were "done done" in the previous iteration.**

This feedback leads to a magical effect. When the team underestimates their workload, they are unable to finish all their stories by the iteration deadline. This causes their velocity to go down, which in turn reduces the team's workload, allowing them to finish everything on time the following week.

Similarly, if the team overestimates their workload, they find themselves able to finish more stories by the iteration deadline. This causes their velocity to go up, which increases the team's workload to match their capacity.

Velocity is an extremely effective way of balancing the team's workload. In a mature XP team, velocity is stable enough to predict schedules with a high degree of accuracy (see "Risk Management"" earlier in this chapter).

Velocity and the Iteration Timebox

Velocity relies upon a strict iteration timebox. To make velocity work, *never* count stories that aren't "done done" at the end of the iteration. *Never* allow the iteration deadline to slip, not even by a few hours.

You may be tempted to cheat a bit and work longer hours, or to slip the iteration deadline, in order to finish your stories and make your velocity a little higher. Don't do that! Artificially raising velocity sabotages the equilibrium of the feedback cycle. If you continue to do it, your velocity will gyrate out of control, which will likely reduce your capacity for energized work in the process. This will further damage your equilibrium and your ability to meet your commitments.

Ally
Energized Work (p. 82)

NOTE

One project manager wanted to add a few days to the beginning of an iteration so his team could "hit the ground running" and have a higher velocity to show to stakeholders. By doing so, he set the team up for failure: they couldn't keep that pace in the following iteration. Remember that velocity is for predicting schedules, not judging productivity. See "Reporting"" in Chapter 6 for ideas about what to report to stakeholders.

Velocity tends to be unstable at the beginning of a project. Give it three or four iterations to stabilize. After that point, you should achieve the same velocity every iteration, unless there's a holiday during the iteration. Use your iteration slack to ensure that you consistently meet your commitments every iteration. I look for deeper problems if the team's velocity changes more than one or twice per quarter.

Ally
Slack (p. 247)

TIPS FOR ACCURATE ESTIMATES

You can have accurate estimates if you:

1. Estimate in terms of ideal engineering days (story points), not calendar time
2. Use velocity to determine how many story points the team can finish in an iteration
3. Use iteration slack to smooth over surprises and deliver on time every iteration
4. Use risk management to adjust for risks to the overall release plan

How to Make Consistent Estimates

Ally

Risk Management (p. 226)

There's a secret to estimating: experts automatically make consistent estimates.* All you have to do is use a consistent estimating technique. When you estimate, pick a single, optimistic value. How long will the story take if you experience no interruptions, can pair with anyone else on the team, and everything goes well? There's no need to pad your estimates or provide a probabilistic range with this approach. Velocity automatically applies the appropriate amount of padding for short-term estimates and risk management adds padding for long-term estimates.

> To make a good estimate, go with your gut.

There are two corollaries to this secret. First, if you're an expert but you don't trust your ability to make estimates, relax. You automatically make good estimates. Just imagine the work you're going to do and pick the first number that comes into your head. It won't be right, but it *will* be consistent with your other estimates. That's sufficient.

Second, if you're not an expert, the way to make good estimates is to become an expert. This isn't as hard as it sounds. An expert is just a beginner with lots of experience. To become an expert, make a lot of estimates with relatively short timescales and pay attention to the results. In other words, follow the XP practices.

All the programmers should participate in estimating. At least one customer should be present to answer questions. Once the programmers have all the information they need, one programmer will suggest an estimate. Allow this to happen naturally. The person who is most comfortable will speak first. Often this is the person who has the most expertise.

NOTE

I assume that programmers are your constraint in this book (see "Theory of Constraints"" in Chapter 3), which means they should be the only ones estimating stories. The system constraint determines how quickly you can deliver your final product, so only the constraint's estimates are necessary. If programmers aren't the constraint on your team, talk with your mentor about how to adjust XP for your situation.

If the suggested estimate doesn't sound right, or if you don't understand where it came from, ask for details. Alternatively, if you're a programmer, provide your own estimate and explain your reasoning. The ensuing discussion will clarify the estimate. When all the programmers confirm an estimate, write it on the card.

NOTE

If you don't know a part of the system well, practice making estimates for that part of the system in your head, then compare your estimate with those of the experts. When your estimate is different, ask why.

* Unless you have a lot of technical debt. If you do, add more iteration slack (see "Slack"" earlier in this chapter) to compensate for the inconsistency.

At first, different team members will have differing ideas of how long something should take. This will lead to inconsistent estimates. If the team makes estimates as a group, programmers will automatically synchronize their estimates within the first several iterations.

Comparing your estimates to the actual time required for each story or task may also give you the feedback you need to become more consistent. To do so, track your time as described in "Time usage"" in Chapter 6.

How to Estimate Stories

Estimate stories in story points. To begin, think of a story point as an ideal day.

> NOTE
> **It's OK to estimate in half-points, but quarter-points shouldn't be necessary.**

When you start estimating a story, imagine the engineering tasks you will need to implement it. Ask your on-site customers about their expectations, and focus on those things that would affect your estimate. If you encounter something that seems expensive, provide the customers with less costly alternatives.

As you gain experience with the project, you will begin to make estimates intuitively rather than mentally breaking them down into engineering tasks. Often, you'll think in terms of similar stories rather than ideal days. For example, you might think that a particular story is a typical report, and typical reports are one point. Or you might think that a story is a complicated report, but not twice as difficult as a regular report, and estimate it at one and a half points.

Sometimes you will need more information to make an accurate estimate. In this case, make a note on the card. If you need more information from your customers, write "??" (for "unanswered questions") in place of the estimate. If you need to research a piece of technology further, write "Spike" in place of the estimate and create a spike solution story (see "Spike Solutions"" in Chapter 9).

> *The team has gathered to estimate stories. Mary, one of the on-site customers, starts the discussion. Amy, Joe, and Kim are all programmers.*
>
> Mary: Here's our next story. *[She reads the story card aloud, then puts it on the table.]* "Report on parts inventory in warehouse."
>
> Amy: We've done so many reports by now that a new one shouldn't be too much trouble. They're typically one point each. We already track parts inventory, so there's no new data for us to manage. Is there anything unusual about this report?
>
> Mary: I don't think so. We put together a mock-up. *[She pulls out a printout and hands it to Amy.]*
>
> Amy: This looks pretty straightforward. *[She puts the paper on the table. The other programmers take a look.]*
>
> Joe: Mary, what's this age column you have here?
>
> Mary: That's the number of business days since the part entered the warehouse.
>
> Joe: You need business days, not calendar days?

Mary: That's right.

Joe: What about holidays?

Mary: We only want to count days that we're actually in operation. No weekends, holidays, or scheduled shutdowns.

Kim: Joe, I see what you're getting at. Mary, that's going to increase our estimate because we don't currently track scheduled shutdowns. We would need a new UI, or a data feed, in order to know that information. It could add to the complexity of the admin screens, and you and Brian have said that ease of admin is important to you. Do you really need age to be that accurate?

Mary: Hmm. Well, the exact number isn't that important, but if we're going to provide a piece of information, I would prefer it to be accurate. What about holidays—can you do that?

Kim: Can we assume that the holidays will be the same every year?

Mary: Not necessarily, but they won't change very often.

Kim: OK, then we can put them in the config file for now rather than creating a UI for them. That would make this story cheaper.

Mary: You know, I'm going to need to look into this some more. This field isn't that important and I don't think it's going to be worth the added administration burden. Rather than business days, let's just make it the number of calendar weeks. I'll make a separate story for dealing with business days. *[She takes a card and writes, "Report part inventory age in business days, not calendar weeks. No holidays, weekends, or scheduled shutdowns."]*

Kim: Sounds good. That should be pretty easy then, because we already track when the part went into the warehouse. What about the UI?

Mary: All we need to do is add it to the list of reports on the reporting screen.

Kim: I think I'm ready to estimate. *[Looks at other programmers.]* This looks like a pretty standard report to me. It's another specialization for our reporting layer with a few minor logic changes. I'd say it's one point.

Joe: That's what I was thinking, too. Anybody else?

[The other programmers nod.]

Joe: One point. *[He writes "1" on the story card.]* Mary, I don't think we can estimate the business day story you just created until you know what kind of UI you want for it.

Mary: That's fair. *[Writes "??" on the business day card.]* Our next story...

How to Estimate Iteration Tasks

During iteration planning, programmers create engineering tasks that allow them to deliver the stories planned for the iteration. Each engineering task is a concrete, technical task such as "update build script" or "implement domain logic."

Ally

Iteration Planning (p. 234)

Estimate engineering tasks in ideal hours rather than ideal days. When you think about the estimate, be sure to include everything necessary for the task to be "done done"—testing, customer review, and so forth. See "Iteration Planning"" earlier in this chapter for more information.

Ally
"Done Done" (p. 155)

When Estimating Is Difficult

If the programmers understand the requirements and are experts in the required technology, they should be able to estimate a story in less than a minute. If the programmers need to discuss the technology, or if they need to ask questions of the customers, then estimating may take longer. I look for ways to bring discussions to a close if an estimate takes longer than five minutes, and I look for deeper problems if every story involves detailed discussion.

One common cause of slow estimating is inadequate customer preparation. To make their estimates, programmers often ask questions the customers haven't considered. In some cases, customers will disagree on the answer and need to work it out.

A *customer huddle*—in which the customers briefly discuss the issue, come to a decision, and return—is one way to handle this. Another way is to write "??" in place of the estimate and move on to the next story. The customers then work out the details at their own pace, and the programmers estimate the story later.

Expect the customers to be unprepared for programmer questions during the first several iterations. Over time, they will learn to anticipate most of the questions programmers will ask.

Programmer inexperience can also cause slow estimating. If the programmers don't understand the problem domain well, they will need to ask a lot of questions before they can make an estimate. As with inadequate customer preparation, this problem will go away in time. If the programmers don't understand the *technology*, however, immediately create a spike story (see "Spike stories"" earlier in this chapter) and move on.

Ally
Spike Solutions (p. 333)

Some programmers try to figure out all the details of the requirements before making an estimate. However, only those issues that would change the estimate by a half-point or more are relevant. It takes practice to figure out which details are important and which you can safely ignore.

This sort of overattention to detail sometimes occurs when a programmer is reluctant to make estimates. A programmer who worries that someone will use her estimate against her in the future will spend too much time trying to make her estimate perfect rather than settling on her first impression.

Programmer reluctance may be a sign of organizational difficulties or excessive schedule pressure, or it may stem from past experiences that have nothing to do with the current project. In the latter case, the programmers usually come to trust the team over time.

Ally
Trust (p. 103)

To help address these issues during the estimation phase, ask leading questions. For example:

- Do we need a customer huddle on this issue?
- Should we mark this "??" and come back to it later?
- Should we make a spike for this story?
- Do we have enough information to estimate this story?

- Will that information change your estimate?
- We've spent five minutes on this story. Should we come back to it later?

Explaining Estimates

It's almost a law of physics: customers and stakeholders are invariably disappointed with the amount of features their teams can provide. Sometimes they express that disappointment out loud. The best way to deal with this is to ignore the tone and treat the comments as straightforward requests for information.

In fact, a certain amount of back-and-forth is healthy: it helps the team focus on the high-value, low-cost elements of the customers' ideas. (See "How to Win"" earlier in this chapter.)

One common customer response is, "Why does that cost so much?" Resist the urge to defend yourself and your sacred honor, pause to collect your thoughts, then list the issues you considered when coming up with the estimate. Suggest options for reducing the cost of the story by reducing scope.

Your explanation will usually satisfy your customers. In some cases, they'll ask for more information. Again, treat these questions as simple requests. If there's something you don't know, admit it, and explain why you made your estimate anyway. If a question reveals something that you haven't considered, change your estimate.

Be careful, though: the questions may cause you to doubt your estimate. Your initial, gut-feel estimate is most likely correct. Only change your estimate if you learn something genuinely new. Don't change it just because you feel pressured. As the programmers who will be implementing the stories, you are the most qualified to make the estimates. Be polite, but firm:

> Politely but firmly refuse to change your estimates when pressured.

> I'm sorry you don't like these estimates. We believe our estimates are correct, but if they're too pessimistic, our velocity will automatically increase to compensate. We have a professional obligation to you and this organization to give you the best estimates we know how, even if they are disappointing, and that's what we're doing.

If a customer reacts with disbelief or browbeats you, he may not realize how disrespectful he's being. Sometimes making him aware of his behavior can help:

> From your body language and the snort you just made, I'm getting the impression that you don't respect or trust our professionalism. Is that what you intended?

Customers and stakeholders may also be confused by the idea of story points. I start by providing a simplified explanation:

> A story point is an estimation technique that automatically compensates for overhead and estimate error. Our velocity is 10, which means we can accomplish 10 points per week.

If the questioner pushes for more information, I explain all the details of ideal days and velocity. That often inspires concern: "If our velocity is 10 ideal days and we have 6 programmers, shouldn't our velocity be 30?"

You can try to explain that ideal days aren't the same as developer days, but that has never worked for me. Now I just offer to provide detailed information:

> Generally speaking, our velocity is 10 because of estimate error and overhead. If you like, we'll perform a detailed audit of our work next week and tell you exactly where the time is going. Would that be useful?

("Reporting"" in Chapter 6 discusses time usage reports in detail.)

These questions often dissipate as customers and stakeholders gain trust in the team's ability to deliver. If they don't, or if the problems are particularly bad, enlist the help of your project manager to defuse the situation. "Team Strategy #1: Customer-Programmer Empathy"" in Chapter 6 has further suggestions.

How to Improve Your Velocity

Your velocity can suffer for many reasons. The following options might allow you to improve your velocity.

Pay down technical debt

The most common technical problem is excessive technical debt (see "Technical Debt""). This has a greater impact on team productivity than any other factor does. Make code quality a priority and your velocity will improve dramatically. However, this isn't a quick fix. Teams with excessive technical debt often have months—or even years—of cleanup ahead of them. Rather than stopping work to pay down technical debt, fix it incrementally. Iteration slack is the best way to do so, although you may not see a noticeable improvement for several months.

Ally
Slack (p. 247)

Improve customer involvement

If your customers aren't available to answer questions when programmers need them, programmers have to either wait or make guesses about the answers. Both of these hurt velocity. To improve your velocity, make sure that a customer is always available to answer programmer questions.

Ally
Sit Together (p. 113)

Support energized work

Tired, burned-out programmers make costly mistakes and don't put forth their full effort. If your organization has been putting pressure on the team, or if programmers have worked a lot of extra hours, shield the programmers from organizational pressure and consider instituting a no-overtime policy.

Ally
Energized Work (p. 82)

Offload programmer duties

If programmers are the constraint for your team—as this book assumes—then hand any work that other people can do *to* other people. Find ways to excuse programmers from unnecessary meetings, shield them from interruptions, and have somebody else take care of organizational bureaucracy such as time sheets and expense reports. You could even hire an administrative assistant for the team to handle all non-project-related matters.

Provide needed resources

Most programming teams have all the resources they need. However, if your programmers complain about slow computers, insufficient RAM, or unavailability of key materials, get those resources for them. It's always surprising when a company nickle-and-dimes its software teams. Does it make sense to save $5,000 in equipment costs if it costs your team half an hour per programmer every day? A team of six programmers will recoup that cost within a month. And what about the opportunity costs of delivering fewer features?

Add programmers (carefully)

Velocity is related to the number of programmers on your team, but unless your project is woefully understaffed and experienced personnel are readily available, adding people won't make an immediate difference. As [Brooks] famously said, "Adding people to a late project only makes it later." Expect new employees to take a month or two to be productive. Pair programming, collective code ownership, and a shared workspace will help reduce that time, though adding junior programmers to the team can actually *decrease* productivity.

Allies

Pair Programming (p. 74)
Collective Code Ownership (p. 191)
Sit Together (p. 113)

Likewise, adding to large teams can cause communication challenges that decrease productivity. Six programmers is my preferred size for an XP team, and I readily add good programmers to reach that number. Past 6, I am very cautious about adding programmers, and I avoid team sizes greater than 10 programmers.

Questions

How do we modify our velocity if we add or remove programmers?

If you add or remove only one person, try leaving your velocity unchanged and see what happens. Another option is to adjust your velocity proportionally to the change. Either way, your velocity will adjust to the correct number after another iteration.

How can we have a stable velocity? Team members take vacations, get sick, and so forth.

Your iteration slack should handle minor variations in people's availability. If a large percentage of the team is away, as during a holiday, your velocity may go down for an iteration. This is normal. Your velocity should return to normal in the next iteration.

Ally

Slack (p. 247)

If you have a small number of programmers—four or fewer—you may find that even one day of absence is enough to affect your velocity. In this case, you may wish to use two-week iterations. See "Iteration Planning"" earlier in this chapter for a discussion of the trade-offs.

What should we use as our velocity at the beginning of the project?

For your first iteration, just make your best guess. Set your velocity for the next iteration based on what you actually complete. Expect your velocity to take three or four iterations to stabilize.

Your organization may want you to make release commitments before your velocity has stabilized. The best approach is to say, "I don't have a schedule estimate yet, but we're working on it. I can promise you an estimate in three or four weeks. We need the time to calibrate the developers' estimates to what they actually produce."

Isn't it a waste of time for all the programmers to estimate stories together?

It does take a lot of programmer-hours for all the programmers to estimate together, but this isn't wasted time. Estimating sessions are not just for estimation—they're also a crucial first step in communicating and clarifying requirements. Programmers ask questions and clarify details, which often leads to ideas the customers haven't considered. Sometimes this collaboration reduces the overall cost of the project. (See "The Planning Game"" earlier in this chapter.)

All the programmers need to be present to ensure they understand what they will be building. Having the programmers together also increases estimate accuracy.

What if we don't ask the right questions of the customer and miss something?

Sometimes you will miss something important. If Joe hadn't asked Mary about the age column, the team would have missed a major problem and their estimate would have been wrong. These mistakes happen. Information obvious to customers isn't always obvious to programmers.

Although you cannot *prevent* these mistakes, you can reduce them. If all the programmers estimate together, they're more likely to ask the right questions. Mistakes will also decrease as the team becomes more experienced in making estimates. Customers will learn what details to provide, and programmers will learn which questions to ask.

In the meantime, don't worry unless you encounter these surprises often. Address unexpected details when they come up. (See "Iteration Planning"" earlier in this chapter.)

If unexpected details frequently surprise you, and the problem doesn't improve with experience, ask your mentor for help.

When should we reestimate our stories?

Story estimates don't need to be accurate, just self-consistent. As a result, you only need to reestimate a story when your understanding of it changes in a way that affects its difficulty or scope.

To make our estimates, we made some assumptions about the design. What if the design changes?

XP uses incremental design and architecture, so the whole design gradually improves over time. As a result, your estimates will usually remain consistent with each other.

Allies

Refactoring (p. 306)
Incremental Design and Architecture (p. 323)

How do we deal with technical dependencies in our stories?

With proper incremental design and architecture, technical dependencies should be rare, although they can happen. I typically make a note in the estimate field: "Six (four if story Foo done first)."

If you find yourself making more than a few of these notes, something is wrong with your approach to incremental design. Ask your mentor for help.

What if we want to get a rough estimate of project length—without doing release planning—before the project begins?

[DeMarco 2002] has argued that organizations set project deadlines based on the value of the project. In other words, the project is only worth doing if you can complete it before the

deadline. (Some organizations play games with deadlines in order to compensate for expected overruns, but assume the deadline is accurate in this discussion.)

If this is true—and it coincides with my experience—then the estimate for the project isn't as important as whether you can finish it before the deadline.

To judge whether a project is worth pursuing, gather the project visionary, a seasoned project manager, and a senior programmer or two (preferably ones that would be on the project). Ask the visionary to describe the project's goals and deadline, then ask the project manager and programmers if they think it's possible. If it is, then you should gather the team and perform some real estimating, release planning, and risk management by conducting the first three or four iterations of the project.

Allies

Release Planning (p. 207)
Risk Management (p. 226)

This approach takes about four weeks and yields a release date that you can commit to. [McConnell 2005] provides additional options that are faster but less reliable.

Results

When you estimate well, your velocity is consistent and predictable with each iteration. You make commitments and meet them reliably. Estimation is fast and easy, and you can estimate most stories in a minute or two.

Contraindications

This approach to estimating assumes that programmers are the constraint (see "XP Concepts"" in Chapter 3 for more about the Theory of Constraints). It also depends on fixed-length iterations , small stories, and tasks. If these conditions aren't present, you need to use a different approach to estimating.

This approach also requires trust: developers need to believe they can give accurate estimates without being attacked, and customers and stakeholders need to believe the developers are providing honest estimates. That trust may not be present at first, but if it doesn't develop, you'll run into trouble.

Ally

Trust (p. 103)

Regardless of your approach to estimating, never use missed estimates to attack developers. This is a quick and easy way to destroy trust.

Alternatives

There are many approaches to estimating. The one I've described has the benefit of being both accurate and simple. However, its dependency on release planning for long-term estimates makes it labor-intensive for initial project estimates. See [McConnell 2005] for other options.

Further Reading

Agile Estimating and Planning [Cohn] describes a variety of approaches to agile estimation.

Software Estimation: Demystifying the Black Art [McConnell 2005] provides a comprehensive look at traditional approaches to estimation.