Part 1

Getting Started

Chapter 1

Groundwork

Writing is imprecise. User stories shift some of the focus to conversations about requirements

Physics. Calculus. My daughters. I'll admit that these often confuse me and I've grown used to that. But I shouldn't be confused by a lunch menu. Today I was.

Entrée comes with choice of soup or salad and bread.

That should not have been a difficult sentence to understand but it was. Which of these did it mean I could choose?

Soup or (Salad and Bread)
(Soup or Salad) and Bread

We often act as though written words are precise, yet they aren't. Contrast the words written on that menu with the waitress' spoken words: "Would you like soup or salad?" Even better, she removed all ambiguity by placing a basket of bread on the table before she took my order.

Just as bad is that words can take on multiple meanings. As an extreme example, consider these two sentences:

Buffalo buffalo buffalo.
Buffalo buffalo Buffalo buffalo.

Wow. What can those sentences possibly mean? Buffalo can mean either the large furry animal (also known as a bison), or a city in New York, or it can mean "intimidate" as in "The developers were buffaloed into promising an earlier delivery date." So, the first sentences means that bison intimidate other bison. The second sentence means that bison intimidate bison from the city of Buffalo.

Unless we're writing software for bison this is an admittedly unlikely example; but, is it really much worse than this typical requirements statement:

- ✧ The system should prominently display a warning message whenever the user enters invalid data.

Does *should* mean the requirement can be ignored if we want? I *should* eat three servings of vegetables a day; I don't. What does *prominently display* mean? What's prominent to whoever wrote this may not be prominent to whoever codes and tests it.

As another example, I recently came across this requirement that was referring to a user's ability to name a folder in a data management system:

- ✧ The user can enter a name. It can be 127 characters.

From this statement it is not clear if the user must enter a name for the folder. Perhaps a default name is provided for the folder. The second sentence is almost completely meaningless. Can the folder name be other lengths or must it always be 127 characters?

So much of these types of confusion—whether from the imprecision of written words or from words with multiple meanings—goes away if we shift the focus from writing requirements down to talking about them.

The Sacred Written Word

Humans used to have such a marvelous oral tradition; myths and history were passed orally from one generation to the next. Until an Athenian ruler started writing down Homer's *The Iliad* so that it would not be forgotten, stories like Homer's were told, not read. Our memories must have been a lot better back then and must have started to fade sometime in the 1970s because by then we could no longer remember even short statements like "The system shall prompt the user for a login name and password." So, we started writing them down.

And that's where we started to go wrong.

It seems so easy to think that if everything is written down and agreed to then there can be no disagreements, developers will know exactly what to build, testers will know exactly how to test it, and, most importantly, customers will get exactly what they wanted. Well, no, that's wrong: Customers will get exactly what they *wrote down*, which may not be exactly what they *wanted*.

What Is a Story?

A user story is a relatively small piece of functionality that will be valuable to the software's users. User stories are traditionally hand-written on note cards, because of their low-tech elegance. (For arguments for and against this practice, see Chapter 6, "Writing Stories.") So, a user story may appear as shown in User Story 1.1, which is a card from a hypothetical online travel reservation system. Other sample stories for the travel reservation system might include:

- A user can cancel a reservation.
- Users can see photos of the hotels.
- Users can restrict searches so they only see hotels with available rooms.

Because user stories represent functionality that will be valued by users the following examples do not make good user stories:

USER STORY 1.1 An initial user story written on a note card.

A user can make a hotel reservation.

- The software will be written in C++.
- The program will connect to the database through a connection pool.

The first example is not a good user story because users are typically unaware of the programming language that was used (unless perhaps this is a programming library or open source project). The second is not a valid user story because users generally do not care about the technical details of how the application connects to the database.

Where Are The Details?

It's one thing to say "A user can make a hotel reservation." It's another thing to code and test a program against this as a system's only requirement. Where are the details? What about all the unanswered questions like:

- Does the user have to enter a credit card to confirm her reservation? If so, what credit cards are accepted and is the charge applied immediately?
- In what ways can the user search for a hotel? Can she search by city? By a quality rating? By price range? By type of room?
- What information is shown for each hotel?

- ✧ Can users make special requests such as for a crib?

Many of these details can be expressed as additional stories. In fact, User Story 1.1 is not a very good story because it's too big. Chapter 6, "Writing Stories," fully addresses the question of story size but as a starting point it's good to have stories that can be coded and tested in between half a day and perhaps a week by one or a pair of programmers. Liberally interpreted, "make a hotel reservation" could imply the majority of a travel reservation system so it will certainly take most programmers more than a week.

When a story is too large it is sometimes referred to as an *epic*. Epics can be split into two or more stories of smaller size. For example, the epic User Story 1.1 could be split into these stories:

- ✧ A user can search for a hotel. Search fields include city, price range, and availability.
- ✧ A user can view detailed information about a hotel.
- ✧ A room can be reserved with a credit card.

However, we do not continue splitting stories until we have a story that covers every last detail. For example the story "a hotel room can be reserved with a credit card" is a very reasonable and realistic story. We do not need to further divide it into:

- ✧ A hotel room can be reserved with a Visa card.
- ✧ A hotel room can be reserved with a MasterCard.
- ✧ A hotel room can be reserved with an American Express card.

Similarly, the user story does not need to be augmented in typical requirements documentation style like this:

4.6) A room can be reserved with a credit card.
  4.6.1) The system shall accept Visa, MasterCard and American Express cards.
  4.6.2) The system shall charge the credit the indicated rate for all nights of the stay before the reservation is confirmed.

4.6.3) The system shall give the user a unique confirmation number.

Rather than writing all these details as stories, the better approach is for the development team and the customer to discuss these details. That is, have a conversation about the details at the point when the details become important. There's nothing wrong with making a few annotations on a story card based on a discussion, as shown in User Story 1.2. However, the conversation is the key, not the note on the story card. Neither the developers nor the customer can point to the card three months later and say, "But, see I said so right there." Stories are not contractual obligations.

USER STORY 1.2 A story card with a note.

> *A room can be reserved with a credit card.*
>
> *Marco says Visa and MasterCard only.*

"How Long Does it Have to Be?"

I was the kid in high school literature classes who always asked, "How long does it have to be?" whenever we were assigned to write a paper. The teachers never liked the question but I still think it was a fair one because it told me what their expectations were. It is just as important to understand the expectations of a project's users. Those expectations are best captured in the form of the acceptance tests.

If you're using paper note cards you can turn the card over and write the acceptance tests on the back of the card, as shown in User Story 1.3. Or, if you're using an electronic system there's probably a place you can enter the acceptance test descriptions.

USER STORY 1.3 The back side of a story card shows acceptance tests for the story.

Try it with a valid Visa and then a valid MasterCard.

Enter card numbers that are missing a digit, have an extra digit and have transposed two digits.

Try it with a card with a valid number but that has been cancelled.

Try it with a card expiration date in the past.

The test descriptions are meant to be short and incomplete. Tests can be added or removed at any time. The goal is to convey additional information about the story so that the developers will know when they are done. Just as my teacher's expectations were useful to me in knowing when I was done writing about *Moby Dick,* it is useful for the developers to know the customer's expectations so they know when they are done.

Card, Conversation and Confirmation

Ron Jeffries has come up with the wonderful alliteration that stories have three critical aspects: Card, Conversation, and Confirmation.[1] This means that a user story is represented by the card it is written on, the conversations between the customer and developers that define and refine the meaning of the story, and finally by the acceptance tests that confirm the story has been correctly programmed.

The Card may be the most visible manifestation of a user story but it is not necessarily the most important. Rachel Davies has said that cards "represent customer requirements rather than document them."[2] This is the perfect way to think about user stories: While the

1. Jeffries, R. "Essential XP: Card, Conversation, and Confirmation." *XP Magazine*, August 30, 2001. www.xprogramming.com.

card may contain the text of the story, the details are to be found in the Conversation and the Confirmation.

Summary

- ✧ Written words are imprecise and may be misleading.
- ✧ Writing things down is no guarantee that customers will get what they want; at best they'll get what they wrote down.
- ✧ User stories are relatively small pieces of functionality that will be valued by users. They can typically be developed in one-half to five days of programming.
- ✧ User stories comprise Card, Conversation, and Confirmation. The text of the story is written on a Card; the details that underlie a story are discovered during Conversations between developers and the customer; one or more acceptance tests is written for each story as Confirmation that the story has been correctly developed.

Questions

1.1 Which of the following are not good stories? Why?

a The user can run the system on Windows XP and Linux.

b All graphing and charting will be done using a third-party library.

c The user can undo up to fifty commands.

d The software will be released by June 30.

e The software will be written in Java.

f The user can select her state from a drop-down list of states.

2. Davies, R. "The Power of Stories," presented at XP2001.

g The system will use Log4J to log all error messages to a file.

h The user will be prompted to save her work if she hasn't saved it for 15 minutes.

i The user can select an "Export to XML" feature.

j The user can export data to XML.

1.2 What is wrong with the requirements statement, "All multi-page reports should be numbered"?

1.3 What advantages do requirements conversations have over requirements documents?

1.4 Why would you want to write tests on the back of a story card?

Chapter 2

What Stories Are Not

User stories are not just a variation on things we've tried in the past.

To help us better understand what user stories are, it's important to look at what they are not. This chapter explains how user stories differ from three other common approaches to requirements: use cases, IEEE 830 software requirements specifications, and interaction design scenarios.

User Stories Aren't IEEE 830

The Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) has published a set of guidelines on how to write software requirements specifications.[1] This document, known as IEEE Standard 830, was last revised in 1998. The IEEE recommendations cover such topics as how to organize the requirements

1. IEEE Computer Society, *IEEE Recommended Practice for Software Requirements Specifications.* New York, NY 1998.

specification document, the role of prototyping, and the characteristics of good requirements. The most distinguishing characteristic of an IEEE 830-style software requirements specification is the use of the phrase "The system shall..." which is the IEEE's recommended way to write functional requirements. A typical fragment of an IEEE 830 specification looks similar to the following:

- 4.6) The system shall allow a room to be reserved with a credit card.
  - 4.6.1) The system shall accept Visa, MasterCard and American Express cards.
  - 4.6.2) The system shall charge the credit card the indicated rate for all nights of the stay before the reservation is confirmed.
  - 4.6.3) The system shall give the user a unique confirmation number.

Documenting a system's requirements to this level is tedious, error-prone, and very time-consuming. Additionally, a requirements document written in this way is, quite frankly, boring to read. Just because something is boring to read is not sufficient reason to abandon it as a technique. However, if you're dealing with 300 pages of requirements like this (and that would only be a medium-sized system) you have to assume that it is not going to be thoroughly read by everyone who needs to read it. Readers will either skim or skip sections out of boredom. Additionally, a document written at this level will frequently make it impossible for a reader to grasp the big picture.

There is a tremendous appeal to the idea that we can think, think, think about a planned system and then write all the requirements as "The system shall…" That sounds so much better than "if possible, the system will..." or even "if we have time, we'll try to…" that better characterizes most projects.

Unfortunately, it is effectively impossible to write all of a sytem's requirements this way. There is a powerful and important feedback loop that occurs when users first see the software being built for them. When users see the software they will come up with new ideas and change their minds about old ideas. When users ask for changes to the software contemplated in the requirements specification it is commonly referred to as a “change of scope.” This type of thinking is incorrect for two reasons. First, it implies that the software was at some point sufficiently well-known for its scope to have been considered fully defined. It doesn’t matter how much effort we put into upfront thinking about requirements, we absolutely know that users will have different (and better) opinions once they see the software. Second, it emphasizes the thinking that software is complete when it fulfills a list of requirements. We need to instead think that software is complete when it fulfills its intended users’ goals. If the scope of the user’s goals change then perhaps we can speak of a “change of scope” but the term is usually applied even when it is only the details of a specific software soluton that have changed.

IEEE 830-style requirements have sent many projects astray because they focus attention on a checklist of requirements rather than on the user’s goals. Lists of requirements do not give the reader the same overall understanding of a product that stories do. It is very difficult to read a list of requirements without automatically considering solutions in your head as you read. Carroll, for example, suggests that designers “may produce a solution for only the first few requirements they encounter.”[2] For example, consider the following requirements:[3]

3.4) The product shall have a gasoline-powered engine.
3.5) The product shall have four wheels.
    3.5.1) The product shall have a rubber tire mounted to each wheel.
3.6) The product shall have a steering wheel.
3.7) The product shall have a steel body.

2. Carroll, J.M. 2000. *Making Use: Scenario-Based Design in Human-Computer Interaction*. Cambridge, MA, The MIT Press.

3. Adapted from Cooper, A. 1999. *The Inmates are Running the Asylum*. Indianapolis, IN, SAMS.

By this point I suppose images of an automobile are floating around in your head. Of course, an automobile satisfies all of the requirements listed above. The one in your head may be a bright red convertible while I might be envisioning a blue pickup. Presumably the differences between your convertible and my pickup are covered in additional requirements statements.

But suppose that instead of writing an IEEE 830-style requirements specification, the customer for this product had written these two user stories:

- ✧ The product makes it easy and fast for the user to mow her lawn.
- ✧ The user is comfortable while using the product.

These two stories tell us that what the customer really wants is a riding lawn mower, not an automobile. Some details may be left out of the user stories but when the stories are dicussed with the customer, the conversation will almost certainly cover the type of engine, number and type of wheels, and how the lawn mower is steered. By focusing on the user's goals for the new product, rather than a list of attributes of the new product, we are able to design a better solution to the user's needs.

User Stories Are Not Use Cases

First introduced by Ivar Jacobsen[4], use cases are today most commonly associated with the Unified Process. A use case is a generalized description of an interaction between the system and an actor, where an actor is either a user or another system. Use cases may be written in unstructured text or to conform with a structured template. The templates proposed by Alistair Cockburn[5] are among the most commonly used. A sample is shown in Use Case 2.1, which is

4. Jacobson, I. 1992. *Object-Oriented Software Engineering*. Upper Saddle River, NJ, Addison-Wesley.

USE CASE 2.1 A sample use case showing

Use Case Title: Accept reservation for a room

Primary Actor: Purchaser

Level: Actor-Goal

Stakeholders and Interests:

Purchaser—to get a good rate on a hotel room

Hotel Company—to maximize RevPAR (Revenue Per Available Room)

Precondition: A room is available for the nights desired

Minimal Guarantees: None

Success Guarantees: Room is reserved; purchaser's credit card charged.

Trigger: Purchaser finds an available room she likes and chooses to reserve it.

Main Success Scenario:

1. Purchaser submits credit card number, date, and authentication information.
2. System validates credit card.
3. System charges credit card full amount for all nights of stay.
4. Purchaser is given a unique confirmation number.

Extensions:

2a: The card is not of a type accepted by the system:

2a1: The system notifies the user to use a different card.

2b: The card is expired:

2b1: The system notifies the user to use a different card.

3a: The card has insufficient available credit to book the room:

3a1: The system charges as much as it can to the current credit card.

3a2: The user is told about the problem and asked to enter a second credit card for the remaining charge. The use case continues at Step 2.

2c: The card is expired:

2c1: The system notifies the user to use a different card.

5. Cockburn, A. 2001. *Writing Effective Use Cases.* Upper Saddle River, NJ, Addison-Wesley.

equivalent to the user story “A hotel room can be reserved with a credit card.”

Since this is not a book on use cases we won’t fully cover all the details of Use Case 2.1; however, it is worth reviewing the meaning of the Main Success Scenario and Extensions sections. The Main Success Scenario is a description of the primary successful path through the use case. In this case, success is achieved after completing the four steps shown. The Extensions section defines alternative paths through the use case. Often, extensions are used for error handling; but, extensions are also used to describe successful but secondary paths, such as in extension 3a of Use Case 2.1. Each path through a use case is referred to as a *scenario*. So, just as the Main Success Scenario represents the sequence of steps one through four, an alternate scenario is represented by the sequence 1, 2, 2a, 2a1, 2, 3, 4.

One of the most obvious differences between stories and use cases is their scope. A use case is almost always much larger than a story. Looking at the user story “A room can be reserved with a credit card” we see it is similar to the main success scenario of Use Case 2.1. This leads to the observation that a user story is similar to a single scenario of the use case. Each story is not necessarily equivalent to a main success scenario; for example, we could write the story “When a user tries to use an expired credit card the system prompts her to enter a different credit card,” which is equivalent to Extension 2b of Use Case 2.1.

User stories and use cases also differ in the level of completeness. James Grenning has noted that “user stories plus acceptance tests are basically the same thing as a use case.”[6] In Chapter 1, “Groundwork,” we saw that appropriate acceptance test cases for this story might be:

- Try it with a valid Visa and then a valid MasterCard.

6. Grenning, J. on extremeprogramming@yahoogroups.com discusson list, February 23, 2003.

- Enter card numbers that are missing a digit, have an extra digit and have transposed two digits.
- Try it with a card with a valid number but that has been cancelled.
- Try it with a card expiration date in the past.

Looking at these acceptance tests we can see the correlation between them and the extensions of Use Case 2.1.

Another important difference between use cases and stories is their longevity. Use Cases are usually permanent artifacts that continue to exist as long as the product is in under active development or maintenance. Stories, on the other hand, are not intended to outlive the iteration in which they are added to the software.

An additional difference is that use cases are more prone to including details of the user interface. This causes definite problems, especially early on in a new project when user interface design should not be made more difficult by preconceptions. I recently came across the use case shown in Use Case 2.2, which describes the steps for composing and sending an email message.

USE CASE 2.2 A use case to compose and send an email message.

Use Case Title: Compose and send email message
Main Success Scenario:

1. User selects the "New Message" menu item.
2. System presents the user with the "Compose New Message" dialog.
3. User edits email body, subject field and recipient lines.
4. User clicks the Send button.
5. System sends the message.

This use case has user interface assumptions throughout. It assumes that there is a "New Message" menu item, that there is a dialog for composing new messages, that there are subject and recipient input fields on that dialog, and that there is a Send button. Many of these assumptions may seem good and safe but they may rule out a user interface where I click on a recipient's name to initiate the message instead of typing it in. And speaking of typing, this use case has precluded voice recognition.

Admittedly, there are far more email clients that work with typed messages than with voice recognition but the point is that a use case is not the proper place to specify the user interface like this. Think about the user story that would replace Use Case 2.2: "A user can compose and send email messages." No hidden user interface assumptions there. With stories, the user interface will come up during the conversation with the customer.

To get around the problem of user interface assumptions in use cases, Constantine and Lockwood have suggested the use of *essential use cases*.[7] An essential use case is a use case that has been stripped of hidden assumptions about technology and implementation details. For example, Use Case 2.3 shows an essential use case for composing and sending an email message. What is interesting about essential use cases is that is that the user intentions could be directly interpreted as user stories.

USE CASE 2.3 An essential use case.

User Intention	**System Responsibility**
Compose email message	
	Collect email content and recipient(s)
Send email message	
	Send the message

7. Constantine, L. L. and Lockwood, L. A. D. 1999. *Software for Use: A practical guide to the models and methods of usage-centered design*. Reading, MA, Addison-Wesley.

Finally, use cases and stories are written for different purposes.[8] Use cases are written in a format acceptable to both customers and developers so that each may read and agree to them. Their purpose is to document an agreement between the customer and the development team. Stories, on the other hand, are written to facilitate release and iteration planning and to serve as placeholders for conversations that uncover the users' detailed needs.

User Stories Aren't Scenarios

In addition to referring to a single path through a use case the word *scenario* is also used by human-computer interaction designers. In this context a scenario is a detailed description of a user's interaction with a computer. The scenarios of interaction design are not the same as a scenario of a use case. In fact, an interaction design scenario is typically larger, or more encompassing, than even a use case. For example, consider this scenario:

> Maria has been thinking about taking a vacation. For the past three years she's spent a week at the same hotel in Cancún but she's ready for a change. Her friend Michelle recommends Vancouver. Maria goes to our website to research hotels. She logs in with her username and password. She wants to find a three- or four-star hotel that is near downtown. Because she was a competitive swimmer in college and still enjoys swimming she wants a hotel with a lap pool. She searches for and finds a few hotels that match. There are more matches than will fit on the screen so Maria resizes her browser to see if more hotels will display at one time. She selects the Royal Canadian hotel. She selects an option on the site to email information about the hotel to someone else. Maria enters Jessica's email address since they will be traveling together. The next morning Maria gets an email from Jessica saying the hotel looks great and that she should go ahead and reserve a room. After review-

8. Davies, R. "The Power of Stories," presented at XP2001.

> ing photos of the rooms, Maria decides they don't need a suite. She selects a standard room with two queen beds and enters her credit card information and reserves a room for three nights next month.

Carroll[9] says that scenarios include the following characteristic elements:

- a setting
- actors
- goals or objectives
- actions and events

The setting is the location where the story takes place. In the story about Maria the story presumably takes place on her home computer; but, since that is not stated the location of the story could be her office during the workday.

Each scenario includes at least one actor. It is possible for a scenario to have multiple actors. For example, in our scenario both Maria and Jessica are actors. Maria may be referred to as the *primary actor* because the scenario mostly describes her interactions with the system. However, because Jessica receives an email from the system and then uses the website to look at a proposed hotel she is considered a *secondary actor*. Unlike use cases, actors in interaction design scenarios are always people and not other systems.

Each actor in a scenario is pursuing one or more goals. As with actors, there can be primary and secondary goals. For example, Maria's primary goal is to reserve an appropriate room at a desirable hotel. While working to achieve that goal she pursues secondary goals such as viewing detailed information about a hotel or sharing information with a friend.

Carroll refers to the actions and events as the plot of a scenario. They are the steps an actor takes to achieve her goal or a system's

9. Carroll, J.M. 2000. *Making Use: Scenario-Based Design in Human-Computer Interaction*. Cambridge, MA, The MIT Press.

response. Searching for a hotel with a pool is an action Maria performs. The response to that action is the event of the system displaying a list of matching hotels.

The primary differences between user stories and scenarios are scope and detail. The example scenario contains many possible stories, such as

- A user can send information about a hotel to a friend via email.
- Users log onto the site.
- Users can search for hotels by excluding hotels that don't have certain amenities.

Even with the additional detail provided by a scenario there are still some questions left unanswered. For example:

- Maria logged onto the site with a username and password. Are all users required to log onto the site? Or does logging on enable some of the features Maria used (perhaps the feature to send an email)?
- When Jessica receives the email does the email contain information about the hotel or does it just link to a page on the site with that information?

Summary

- User stories are different from IEEE 830 software requirements specifications, use cases, and interaction design scenarios.
- No matter how much thinking, thinking and thinking we do we can never fully specify a non-trivial system upfront.
- There is a valuable feedback loop that occurs between defining requirements and users having early and frequent access to the software.

- It is more important to think about users' goals than to list the attributes of a solution.
- User stories are similar to a use case scenario. But use cases still tend to be larger than a single story and are more prone to having embedded assumptions about the user interface.
- Additionally, user stories differ from use cases in their completeness and longevity. Use cases are much more complete than are user stories. Use cases are designed to be permanent artifacts of the development process; user stories are more transient and not intended to outlive the iteration in which they are developed.
- User stories and use cases are written for different purposes. Use cases are written so that developers and customers can discuss them and agree to them. User stories are written to facilitate release planning and to serve as reminders to fill in requirements details with conversations.
- Interaction design scenarios are much more detailed than user stories.
- A typical interaction design scenario is much larger than a user story. One scenario may comprise multiple use cases, which, in turn, may comprise many scenarios.

Questions

2.1 What are the key differences between user stories and use cases?

2.2 What are the key differences between user stories and IEEE 830 requirements statements.

2.3 What are the key differences between user stories and interaction design scenarios?

2.4 For a non-trivial project, why is it impossible to write all the requirements at the start of the project?

2.5 What is the advantage to thinking about users' goals rather than on listing the attributes of the software to be built?

Chapter 3

Why User Stories?

Stories offer many advantages over use cases, scenarios, and IEEE 830 software requirements specifications. Here we find out why we should work with user stories rather than with these alternative approaches.

With all of the available methods for considering requirements why should we choose user stories? This chapter looks at the following advantages of user stories over alternative approaches:

- User stories force a shift to verbal communication
- User stories are comprehensible by everyone
- User stories are the right size
- User stories work for iterative development
- User stories are perfect for time-constrained projects
- User stories encourage participatory design

Verbal Communication

One great reason to use stories is because they shift the focus away from written communication and to verbal communication. Chapter 1, "Groundwork," already showed us how written text can be imprecise and words can have multiple meanings. Naturally, these problems exist as well with verbal communication; but, when customers, developers, and users talk there is not the false appearance of precision and accuracy that there is with written words. No one signs off on a conversation and no one points to it and says, "Right there, three months ago on a Tuesday you said passwords could not contain numbers."

Rachel Davies has said that stories "represent customer requirements rather than document them."[1] Our goal with user stories is not to document every last detail about a desired feature; rather, it is to write down a few short placeholding sentences that will remind developers and customers to hold future conversations. Many of my conversations occur through email and I couldn't possibly do my job without it. I send and receive hundreds of emails every day. But, when I need to talk to someone about something complicated I invariably pick up the phone or walk to the person's office or workspace.

A recent conference on traditional requirements engineering included a half day tutorial on writing "perfect requirements" and promised to teach techniques for writing better sentences to achieve perfect requirements. Writing *perfect* requirements seems like such a lofty and unattainable goal. And even if each sentence in a requirements document is perfect when we type it, we still have the problem of users changing their minds as they learn more about the software being developed. A far more valuable goal is to augment *adequate stories* with *perfect conversations.*

1. Davies, R. "The Power of Stories," presented at XP2001.

User Stories Are Comprehensible

One of the advantages that use cases and scenarios brought us over IEEE 830-style software requirements specifications is that they were understandable by both users and developers. IEEE 830-style documents often contain too much technical jargon to be readable by users and too much domain-specific jargon to be readable by developers.

However, the brevity of stories offers an advantage over scenarios. Constantine and Lockwood[2] have observed that the emphasis scenarios place on realism and detail can cause scenarios to obscure broader issues. This makes it more difficult when working with scenarios to understand the basic nature of the interactions.

User Stories Are the Right Size

Additionally, user stories are the right size—not too big, not too small, but just right. At some point in the career of most developers it has been necessary to ask a customer or user to prioritize IEEE 830-style requirements. The usual result is that 90% of the requirements are mandatory, 5% are very desirable but can be deferred briefly, and another 5% may be deferred a bit longer. This is because it's hard to prioritize and work with thousands of sentences all starting with "The system shall…" For example, consider the following example:

4.6) The system shall allow a room to be reserved with a credit card.
   4.6.1) The system shall accept Visa, MasterCard and American Express cards.
      4.6.1.1) The system shall verify that the card has not expired.

2. Constantine, L. L. and Lockwood, L. A. D. 1999. *Software for Use: A practical guide to the models and methods of usage-centered design,* p. 101. Reading, MA, Addison-Wesley.

        4.6.2) The system shall charge the credit card the indicated rate for all nights of the stay before the reservation is confirmed.

    4.7) The system shall give the user a unique confirmation number.

Each level of nesting within an IEEE 830 requirements specification indicates a relationship between the requirements statements. In the example above, it is unrealistic to think that a customer could prioritize 4.6.1.1 separately from 4.6.1. If items cannot be prioritized or developed separately, perhaps they shouldn't be written as separate items. If they are only written separately so that each may be discretely tested, it would be better to just write the tests directly.

When you consider the thousands or tens of thousands of statements in a software requirements specification (and the relationships between them) for a typical product, it is easy to see the inherent difficulty in prioritizing them.

Use cases and interaction design scenarios suffer from the opposite problem—they're just too big. Just as it is hard to prioritize when there are thousands of small "The system shall…" statements it is also hard to priorize when there are a few dozen use cases or scenarios. Many projects have tried to correct this by writing many smaller use cases with the result that they swing too far in that direction.

Stories, on the other hand, are of a manageable size such that they may be conveniently used for planning releases and by developers for programming and testing.

User Stories Work for Iterative Development

User stories also have the tremendous advantage that they are compatible with iterative development. I do not need to write all of my stories before I begin coding the first. I can write some stories, code and test those stories, and then repeat as often as necessary. When writing stories I can write them at whatever level of detail is appropriate. If I'm just starting to think about a project I may write epic stories like “the user can compose and send email.” That may be just right for very early planning. Later I’ll split that story into perhaps a dozen other stories:

- A user can compose an email message.
- A user can include graphics in email messages.
- A user can send email messages.
- A user can schedule an email to be sent at a specific time.

Scenarios and IEEE 830 documents do not lend themselves to this type of progressive levels of detail. By the way they are written, IEEE 830 documents imply that if there is no statement saying “The system shall…” then it is assumed that the system shall not. This makes it impossible to know if a requirement is missing or simply has not been written yet.

The power of scenarios is in their detail. So, the idea of starting a scenario without detail and then progressively adding detail as it is needed by the developers makes no sense and would strip scenarios of their usefulness entirely.

Use cases can be written at varying levels of detail and Cockburn[3] has suggested excellent ways of doing so. However, rather than writing use cases with free-form text, most organizations define a standard template. The organization then mandates that all use cases conform to the template. This becomes a problem many people feel compelled to fill in each space on a form. In practice, few organizations have the discipline to write some use cases at a summary level and some at the detail level.

3. Cockburn, A. 2000. *Writing Effective Use Cases.* Upper Saddle River, NJ, Addison-Wesley.

Perfect for Time-Constrained Projects

Stories also have the advantage that they are perfect for time-constrained projects. On a project that is assigned an aggressive deadline many teams will find it difficult to stick with a heavier weight requirements definition process, even if they believe doing so will save them time over the long term. On many time-constrained projects there is a tremendous feeling that the team must rush to start coding. In these cases the team is better off with a process that lets them document requirements quickly and also defer detailed decisions.

User stories fit the bill here because they are less formal and less verbose than other forms of requirements documents. This means that user stories can typically be written much more quickly. Similarly, user stories defer as long as possible the collection of detailed requirements. An initial placeholder is written but the details are only discussed when a developer is ready to program the story. This makes stories ideal for use on time constrained projects.

User Stories Encourage Participatory Design

In *participatory design* the users of a system become a part of the team designing their software. They do not become a part of the team through management edict ("Thou shalt form a cross-functional team and include the users"); rather, the users become part of the team because they are so engaged by the requirements and design techniques in use.

Standing in contrast to participatory design is *empirical design*, in which the designers of new software make decisions by studying the prospective users and the situations in which the software will be used. Empirical design relies heavily on interview and observation but users do not become true participants in the design of the software.

User stories and scenarios, unlike IEEE 830 statements and use cases, are totally comprehensible to users. Use cases, and especially IEEE 830 software requirements specifications, are not as comprehensible to users. The greater accessibility of stories and scenarios encourages users to become participants in the design of the software. Further, as users learn how to characterize their needs in stories that are directly useful to developers, developers more actively engage the users. This virtuous cycle benefits everyone involved in developing or using the software.

Summary

- User stories force a shift to verbal communication. Unlike other requirements techniques that rely entirely on written documents, user stories place significant value on conversations between developers and users.
- User stories are comprehensible by both developers and users. IEEE 830 software requirements specifications and use-cases tend to be filled with too much technical or business jargon.
- User stories, which are typically smaller than use cases and scenarios but larger than IEEE 830 statements, are the "right size." Planning, as well as programming and testing, can be completed with stories without further decomposition or aggregation.
- User stories work well with iterative development because it is easy to start with an epic story and later split it into multiple smaller user stories.
- User stories are perfect for time-constrained projects. Individual user stories may be written very quickly and it is also extremely easy to write stories of different sizes. Areas of less importance, or that won't be developed initially, may easily be left as epics while other stories are written with more detail.
- User stories encourage participatory, rather than empirical, design, in which users become active and valued participants in the design of the software.

Developer Responsibilities

- ✧ You are responsible for understanding why you have chosen any technique you choose. If you decide to write user stories, you need to know why.
- ✧ You are responsible for knowing the advantages of other requirements tecniques or for knowing when it may be appropriate to apply one. For example, if you are working with a customer and cannot come to an understanding about a feature, perhaps discussing an interaction design scenario or developing a use case may help.

Customer Responsibilities

- ✧ One of the biggest advantages of user stories over other requirements approaches is that they encourage participatory design. You are responsible for becoming an active participant in the design of the software.

Questions

3.1 What are the six good reasons for using user stories to express requirements?

3.2 What is the key difference between participatory and empirical design?

Chapter 4

Who's The User?

There's rarely an ideal user available when you need one. In those cases it is possible to have other people fill in and represent your users.

If we're going to build a system with *user* stories then we certainly need a *user*. Unfortunately, it is sometimes harder to find a user than it should be. We might be doing a shrinkwrap product with users across the country but be unable to bring one (or more) of them onsite with us to write the stories. Or we might be writing software that will be used within our company but someone tells us we cannot talk to the users.

On many projects there is no single user available to help write or identify stories. These projects will need a *user proxy*. Selection of an appropriate user proxy can be critical to the success of the project. The background and motives of possible proxy users must be considered. A user proxy with a marketing background will approach the stories differently than will a user proxy who is a domain expert. It is important to be aware of these differences. In this chapter we will consider various user proxy types who may sometimes fill in for real users.

The Users' Manager

When doing development on a project for interal use, the organization may be reluctant to give you full and unlimited access to one or more users but may be willing to give you access to the users' manager. Consider this a bait-and-switch unless the manager is also a true user of the software. Even then, it is almost certain that the manager has different usage patterns of the software than does a typical user. For example, on one call center application the team was initially given access to shift supervisors. While shift supervisors did use the software, many of the features they wanted in a new version were focused around managing call queues and transferring calls between agents. These features were of very minimal importance to the users they supervised, for whom the software was mostly intended. If the developers had not pushed for direct access to more typical users, the supervisors' less frequently used features would have been overemphasized in the product.

Sometimes the users' manager intercedes and wants to play the user role on the project because of her ego. She may acknowledge that she's not a typical user but will insist that she knows more about what her users need than they do. Naturally, though, in this type of situation you will need to be careful not to offend the user's manager. But, you do need to find a way at least partially around her and to the end users for the project to succeed. Some ideas for this are given later in this chapter in the section "What to Do When Working With a User Proxy" on page 42.

A Development Manager

A development manager is one of the worst possible choices to act as a proxy user, unless perhaps you are writing software targeted at development managers. While the development manager may

Five Minutes Does Not Equal One Minute

The "user" on this internal project was a Vice President who never used the software and had a level of managers between her and the end users. In prioritizing stories for the next iteration she wanted the developers to focus on improving the speed of the database queries. The team noted the story and its high priority but they were puzzled. They knew application performance was critical and had built a monitoring mechanism into the software: each time a database query was executed its parameters, the time it took to execute, and the name of the user were stored in the database. This information was monitored at least once a day and there had been no indications of a performance problem. Yet their "user" had told them that some queries were taking "up to five minutes."

After the meeting with the Vice President, the team looked into the query execution history. Here's what they found: A couple of users had in fact executed queries that took one minute to complete. That was certainly longer than desired but considering what they were searching for, the size of the database, and the infrequency of that type of search it was within the expected performance of the system. But the users had reported the one-minute query to their manager. The manager then reported it to the Vice President; but, to make sure the problem got the Vice President's attention, the manager said queries were taking two minutes. Then the Vice President reported it to the developers and to make sure she got their attention increased the problem to "up to five minutes."

Users' managers can be sources of misinformation. Whenever possible corroborate their statements by talking to real users.

have nothing but the most honorable intentions, it is far too likely that she will also have some conflicting goals. For example, the development manager may prioritize stories differently than would a real user because doing so allows her to accelerate the introduction of an exciting, new technology. Additionally the development manager may have unaligned corporate goals: perhaps her annual bonus is tied to a completion date on the project, which could cause her to consider the project complete before a real user would.

Finally, most development managers simply do not have hands-on experience as users of the software they are building and are not domain experts. If your prospective user is a development manager who does have domain expertise, then consider her a domain expert and read the discussion in the "Domain Experts" section of this chapter before deciding if you have an adequate user proxy.

The Marketing Group

Larry Constantine[1] points out that marketing groups understand markets rather than users. This can lead a marketing group to focus more on the quantity of features in the product than on the quality of those features. In many cases a marketing group may provide useful high-level guidance about relative priorities but often does not have the insight to provide specific details about the stories.

In one company the marketing group was serving as a customer proxy for a new product that would replace the company's current paper-based product. The company had a very successful history selling printed books containing rules that hospitals and insurance companies had agreed upon: If the hospital followed the rules they would be reimbursed by the insurance company. For example, an appendectomy was only called for if (among other things) the patient's white blood count was above a certain threshold.

The marketing group had no interest in talking to the users of the printed books to find out what they might want the software to do. Instead they reasoned that they knew exactly what their users would want and development could proceed under the guidance of the marketing group. The marketing group chose a book metaphor for the software. Rather than take advantage of the inherent flexibility

1. Constantine, L. L. and Lockwood, L. A. D. 1999. *Software for Use: A practical guide to the models and methods of usage-centered design*. Reading, MA, Addison-Wesley.

of software they settled for an “automated book.” The users were, quite obviously, disappointed with the software. Unfortunately, the company could have known this very early on if they had used real users rather than a marketing department as a user proxy.

Salespersons

The danger in using a salesperson as a user proxy is that it does not lead to a comprehensive view of the product to be built. The most important story to a salesperson will usually be the story whose absence cost her the last sale. If she lost a sale because the product does not have an undo feature you can bet that the undo story card will be instantly sorted right to the top of the pile. Depending on the importance of a specific lost sale it may be desirable to write a new story or two; however, a product development company that puts too much emphasis on each lost sale may lose track of whatever strategic, long-term vision is held for the product.

Salespeople are, however, a great conduit to users and you should use them in this way. Ask them to introduce you to customers either on the phone or while along on a sales visit. Even better, attend an industry trade show and work in your company’s exhibit.

Domain Experts

Domain experts, sometimes called subject matter experts, are critical resources because of how well they understand the domain the software will be targeted at. Naturally some domains are harder to understand than others. I used to write a lot of software for attorneys and paralegals and while the software was sometimes complex, I could usually understand what they were asking for. Much later, I was involved with writing software for statistical geneticists. This domain was filled with words like phenotype, centimorgan, and haplotype. These were words I had never heard before, which made the domain much harder to grasp. This made all of the developers much

Talk to your users

In 1995 this team was challenged with creating what would be one of the web's first general health information sites. Because there were not yet any competitors the team could not look to them for story ideas. The user proxy for the project was a director with a marketing background. Because of his marketing background he understood the importance of talking with prospective users to find out what they would want in a health information site. However, because of the pressure to deliver the site quickly he plunged forward and built the site using only his gut feel to guide him.

As you can guess, the project didn't fulfill its users' needs. About a month after the site launched I walked into the marketing director's office. He pointed to his monitor and said, "Look at that. Just look at that." On his screen was a porn site. After a moment I asked why we were looking at it. I don't think he'd even noticed the porn on the site. His eyes were fixed on a hit counter and he said, "Just look, they've had 100,000 hits since yesterday. We've had 200."

If you want your software to be used, you have to talk to those who will use it.

more reliant on a domain expert to help us understand what we were developing.

While domain experts are great resources, their usefulness is really dependent upon whether they are current or former users of the software type you are building. For example, when building a payroll system you will undoubtedly want to have a Certified Public Accountant (CPA) available as a domain expert. However, since the users will probably be payroll clerks and not CPAs you will probably get better stories from the payroll clerks.

Another potential problem with using a domain expert as your user proxy is that you may end up with software aimed only at users with similar levels of domain expertise. Domain experts can be inclined to point the project toward a solution that is suitable for them but is too complex or is just plain wrong for the targeted user audience.

Former Users

A former user can be great as a proxy if her experience is very recent. However, as with other user proxies you should carefully consider whether the former user's goals and incentives are fully aligned with those of the real users.

Customers

Customers are those who make the buying decision; they are not necessarily users of the software. It is important to consider the desires of your customers because they, not your users, are the ones who write the check to buy the software. (Unless, of course, your users and customers are the same people.)

Corporate desktop productivity software is a perfect example of the distinction between customer and user. The corporate IT staff may decide which word processor is used by all the employees of the company. In this case, the IT staff is the customer but all employees of the company are users (including the IT staff, who are both customer and user). The features in a product like this must be sufficient that the users do not scream too loudly; but, the features must also be those that appeal to the customer making the buying decision.

For example, security features are typically unimportant to most users of desktop productivity software. Security is vitally important, however, to the IT staff (the customers) who make the buying decision.

This project team had designed a database-intensive application. Data would be loaded into the system from other systems the customers had already had. The developers needed to specify a file format that would be used to exchange this data. In this case the customer was the CIO of the organization; the users of this functionality were the IT staff in his organization who would write the extract program to move data from their current systems into the format specified for the new system. When asked about his preferences for the file format, the customer (the CIO) decided that XML would be an ideal technology since it was relatively new at the time and was certainly sexier than a non-standard comma-separated value (CSV) file. When the software was delivered, the users (the IT staff) completely disagreed—they would have preferred the much simpler to generate CSV file over the XML file. If the development team had got the stories directly from the users they could have known this and would not have wasted time on the XML format.

Trainers and Technical Support

Trainers and technical support personnel may seem like logical choices to fill in as a user proxy. They spend their days talking to real users so they must certainly know what users want. Unfortunately, if you use a trainer as your user proxy you will end up with a system that is easy to train. Similarly, if you use someone from technical support you will end up with a system that is easily supported. For example, someone from technical support may put low priority on the advanced features that she anticipates will lead to increased support work. While training ease and and supportability are good goals, they are most likely not what a true user would prioritize.

What to Do When Working With a User Proxy

Although not ideal, it is still possible to write great software with a user proxy rather than a real user. There are a number of techniques you can apply to increase your chances of success in these cases.

When a real user is available but access to her is restricted, one of the best techniques is to request permission to start a *user task force*. A user task force consists of any number of real users, from a handful to a couple of dozen. You pitch the task force on the premise that it exists to bounce ideas off but that the designated user proxy will be your final decision-maker. In most cases the final decision-maker will go along with this, especially as it gives her a safety net to protect against bad decisions.

Once you have a user task force in place and staffed with real users you use it to guide as much of the decision-making as possible. You can do this by having a series of meetings to discuss parts of the application and then have the group identify, write, and prioritize stories.

One project team developing a system for internal users had great success by taking the direction of the user proxy, using prototypes to show a user task force, and then acting on feedback generated during user task force meetings. This particular project ran with month-long iterations. The first few days to a week of each iteration was spent prototyping and holding one or more user task force meetings. In this way the user proxy (the users' manager in this case) had fine control over the strategic direction of the project but the implementation details were moved from her to the user task force.

When There Really Is No User Available

When there really is no user available and you must resort to a user proxy, one valuable technique is use two user proxies instead of one. This helps reduce the likelihood of building a system that meets exactly one person's needs. When using more than one user proxy be sure to use different types of user proxy. For example, combine a domain expert with someone from marketing rather than using two domain experts. You can do this by either having two designated user proxies or by having one designated user proxy but encouraging her to rely on other, informal user proxies.

If you are developing software that will compete with other commercial products you can use the competing products as a source for your stories. What features in the competing products get mentioned in reviews? What features are discussed in online newsgroups? Are the features discussed because they are overly complex to use?

I remember arguing a few years ago with a use case advocate about what type of document best expresses the requirements of a system. He argued in favor of a well thought-out use case model. I argued in favor of the user's guide. I have never seen a project that concluded with a perfectly accurate and current use case model. I have seen many that conclude with an accurate and current user's guide. If you are writing new software that will compete with existing products, you can learn a great deal by studying the competing products.

Another technique you can use when working with a user proxy rather than real users is to release the product as soon as possible. Even if the release is called preliminary or an early beta, getting it into the hands of users early will help identify inconsistencies between the thinking of your user proxy and your real users. Even better, once the software is in the hands of one or more early adopters you have now opened up a communication path to those users and can use that to talk with them about upcoming features.

Can You Do It Yourself?

When you cannot find, or get access to, a real user, avoid falling into the trap of thinking you know your users' minds and do not need or can ignore your user proxy. While each type of user proxy has some type of shortcoming that makes her less desirable than a real user, most developers come with even more shortcomings for pretending to be a real user. In general, developers do not have marketing backgrounds that allow them to understand the relative value of features, they do not have the same amount of customer contact as salespeople, they are not domain experts, and so on.

Summary

- In this chapter we learned about different types of user proxies and why no user proxy is as ideal as a true user when it comes to writing *user* stories.

- The users' manager may not be an appropriate user proxy unless she is also a user.
- Development managers make tempting user proxies because they are already involved in the day-to-day detail of the project. However, the development is rarely an intended user of the software being built and is therefore a poor choice as a user proxy.
- In product companies the customer frequently comes from the marketing group. Someone from the marketing group is often a good choice as a user proxy but must overcome the temptation to focus on the quantity rather than quality of features.
- Salespeople can make good customers when they have contact with a broad variety of customers who are also users. Salespeople must avoid the temptation to focus on whatever story could have won the last lost sale. In all cases, salespeople make excellent conduits to users.
- Domain experts can make excellent user proxies but must avoid the temptation to write stories for a product that only someone with their expertise can use.
- Customers, those who make the purchasing decision, can make great user proxies if in close communication with the users for whom they are purchasing the software. Obviously, a customer who is also a user is a fantastic combination.
- In order to be good user proxies, trainers and technical support personnel must avoid the temptation to focus too narrowly on the aspects of the product they see every day.
- This chapter also looked briefly at some techniques for working with user proxies, including the use of user task forces, using multiple user proxies, competitive analysis, and releasing early to get user feedback.

Developer Responsibilities

- ✧ You have the responsibility to help your organization select the appropriate user for the project.
- ✧ You are responsible for understanding how different types of user proxies will think about the system being built and how their backgrounds might influence your interactions.

Customer Responsibilities

- ✧ If you will not be a user of the software, you are responsible for knowing which categories of user proxy describe you.
- ✧ You are responsible for understanding what biases you may bring to the project and for knowing how to overcome them, whether by relying on others or some other means.

Chapter 5

User Roles and Personas

Almost all software has more than one user. Identifying broad types of users can help you identify stories for all of your users.

On many projects, stories are written as though there is only one type of user. All stories are written from the perspective of that user type. This simplification is a fallacy and can lead a team to miss stories for users who do not fit the general mold of the system's primary user type. The disciplines of usage-centered design[1] and interaction design[2] teach us the benefits of identifying user roles and personas prior to writing stories. In this chapter we will look at user roles, role modeling, user role maps, and personas and show how taking these initial steps leads to better stories and better software.

1. Constantine, L. L. and Lockwood, L. A. D. 1999. *Software for Use: A practical guide to the models and methods of usage-centered design*. Reading, MA, Addison-Wesley.
2. Cooper, A. 1999. *The Inmates are Running the Asylum*. Indianapolis, IN, SAMS.

User Roles[3]

Suppose we are building a web-based travel reservation system. Users will be able to book hotels, flights, rental cars, cruises, and so on. Who do you think the user is for a system like this? Is the user Jim, a frequent flier who travels every week but always to the same place? Or is it Mary, another frequent flier but one who never knows where she'll travel until the day before? Perhaps the user is Mary's assistant, Howard, who books most of Mary's reservations. Or maybe we should target Laura, who just wants a simple way to schedule her family's vacation this year? But wait, maybe the user is Dominic, a marketing vice president for a hotel chain, who wants to use the site to monitor reservations across his entire chain.

Clearly we cannot write stories from a single perspective and have those stories reflect the experiences, backgrounds, and goals of each of these users. Jim may use the site weekly to book his routine trips but he may not be a power user and know the advanced features of the system as well as Howard. Howard probably knows all the best ways to search the system to find the best flights for his boss, Mary. Laura may use the system once a year so she needs it to be simple; however, if she's planning a family vacation she may want to do more research on the hotels she selects than would perhaps Jim or Howard who are traveling for business rather than pleasure.

Each user comes to your software with a different background and with different goals. However, while each user is unique it is still possible to aggregate individual users and think of them in terms of *user roles.* A user role is a collection of defining attributes that characterize a population of users and their intended interactions with the system. So, in the example above we identified the following user roles:

3. Much of the discussion of user roles in this chapter is based on the work of Larry Constantine and Lucy Lockwood. Further information on user role modeling is available in Constantine, L. L. and Lockwood, L. A. D. 1999. *Software for Use: A practical guide to the models and methods of usage-centered design*. Reading, MA, Addison-Wesley.

- Frequent Flier (Jim and Mary)
- Repeat Traveler (Jim)
- Scheduler (Howard)
- Infrequent Vacation Planner (Laura)
- Insider (Dominic)

Naturally, there will be some overlap between different user roles. As we see from the travel reservation system, almost every user role will book trips on the system. A couple of the roles will use the system to do more serious research about a destination. You want to continue defining roles until the overlap between them becomes such that introducing new user roles is not increasing your understanding of the users.

Identifying Roles

To identify user roles the customer and as many of the developers as possible meet in a room with either a large table or a wall they can tape cards to. It's always ideal to include the full team for the user role modeling that initiates a project but it's not necessary. As long as a reasonable representation of the developers are present along with the customer you can have a successful session.

Each participant grabs a stack of note cards from a pile placed in the middle of the table. (Even if you plan to store the user roles electronically you should start by writing them on cards.) Start with everyone writing role names on cards and then placing them on a table or taping or pinning them to a wall.

When a new role card is placed the author says the name of the new role and nothing more. Much as in a brainstorming session, there is no dicsussion of the cards or evaluation of the roles. Rather, each person writes as many cards as he or she can think of. There are no turns, you don't go around the table asking for new roles. Each participant just writes a card whenever she thinks of a new role.

Cards are placed to show approximate relationships. For example, in a travel reservation system the Frequent Flier card will be placed to overlap the Business Traveler card because those roles share some attributes. The Vacation Traveler card would not overlap the Business Traveler card. If two cards represent identical roles (either with the same or different names) they are placed to overlap completely.

While brainstorming roles the room will be filled with sounds of pens scratching on cards and will be punctuated by someone occasionally placing a new card and reading the name of the role. Continue until progress stalls and participants are having a hard time thinking up new roles. At that point you may not have identified all of the roles but you're close enough. Rarely does this last longer than fifteen minutes.

After the group has identified as many roles as they can think of try to consolidate and condense the roles. Start with cards that are entirely overlapping. The authors of overlapping cards describe what they meant by those role names. After a brief discusson the group decides if the roles are equivalent. If equivalent the roles can either be consolidate into a single role (perhaps taking its name from the two initial roles) or one of the initial role cards can be ripped up.

Role Modeling

Once we've written down a few user roles it is possible to model those roles by defining attributes of the role. A role attribute is a fact or bit of useful information about the users who fulfill the role. Any information about the user roles that distinguishes one role from another may be used as a role attribute. Here are some attributes worth considering when preparing any role model:

- The frequency with which the user will use the software.
- The user's level of expertise with the domain.
- The user's general level of proficiency with computers and software.

- The user's level of proficiency with the software being developed.
- The user's general goal for using the software. Some users are after convenience, others favor a rich experience, and so on.

Beyond these standard attributes you should consider the software being built and see if there are any attributes that might be useful in describing its users. For instance, in the travel reservation system above we considered whether the user will be using the software for himself or on behalf of someone else.

While identifying and considering user roles you may find it useful to take notes on 4" x 6" note cards. You can then hang the cards in a common area used by the team so they be used as reminders. A sample user role card is shown in Figure 5.1.

User Role: Infrequent Vacation Planner

Not particularly computer-savvy but quite adept at using the Web. Will use the software infrequently but intensely (perhaps 5 hours to research and plan a trip). Values richness of experience (lots of content) over speed. But, software must be easy to learn and easily recalled months later.

FIGURE 5.1 A sample user role card.

User Role Maps

Even with only a handful of user roles, the interactions and relationships between them can become too much to remember. In those cases it is very useful to draw a user role map, as shown in Figure 5.2.

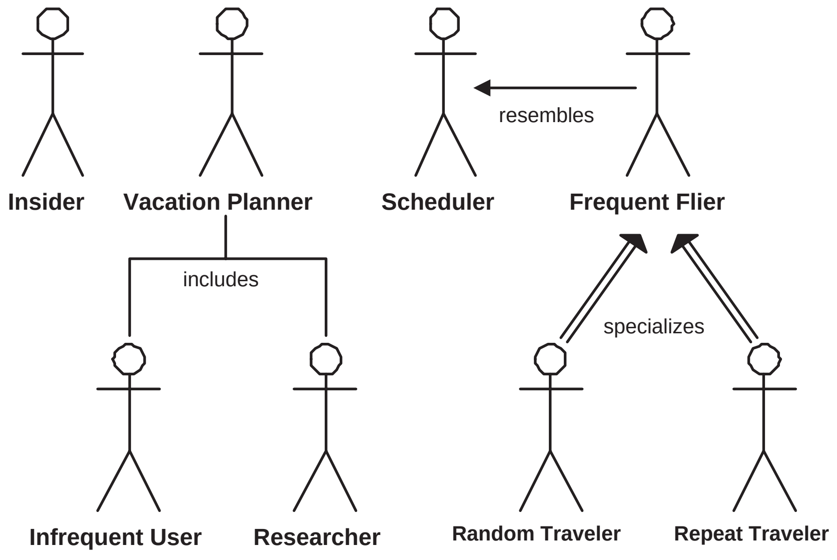


FIGURE 5.2 A user role map for the travel reservation system.

A user role map uses stick figures to represent each role that will use the software. Lines are drawn between the role icons to indicate specific types of relationship. A solid line indicates an inclusionary relationship. Figure 5.2 shows that the Vacation Planner role includes both the Infrequent User and Researcher roles. This indicates that a Vacation Planner shares attributes with the two roles it includes.

The dashed line between Scheduler and Frequent Flier indicates a resemblance relationship. A Scheduler (someone who schedules travel for someone else, just as Howard did in the example at the start of this chapter) resembles a Frequent Flier in his use of the software.

The final type of relationship shown in Figure 5.2 is the specialization relationship between Frequent Flier, Random Traveler, and Repeat Traveler. A Repeat Traveler is someone like Jim from our example who travels repeatedly to the same place. A Random Traveler may perhaps be a traveling salesperson who is on the road most weeks but without any set pattern to her travel. The usage patterns of a Repeat Traveler will differ from a Random Traveler because the Repeat Traveler will value software features that let him quickly book a new trip based on an old trip. However, a Random Traveler and a Repeat Traveler will share many of the stories so it is useful to think of them as specialized roles based on a general Frequent Flier role.

While it can be very useful to create a user role map you should not obsess about drawing all the relationships perfectly. It is possible to draw completely different sets of relationships and still have accurate user role maps. The practice of talking about users sufficiently to identify user roles and relationships is what will benefit you. Drawing the perfect user role map and identifying each role perfectly should not even be your goals.

Personas

Identifying user roles is a great leap forward but for some of the more important user roles it is usually worth going one step further and creating a *persona* for the role. A persona is an imaginary representation of a user role. In this chapter's example, Jim is a persona representing a Repeat Traveler. Creating a persona requires more than just adding a name to a user role. A persona should be described sufficiently that everyone on the team feels like they know the persona. For example, Jim may be described as follows:

> Jim lives in four bedroom house in a nice suburb north of Chicago. However, he works as a vice president of marketing in Sacramento, California. Three weeks out of every four he flies from Chicago to Sacramento on Monday morning and then flies home on Friday. The company lets him work every fourth week out of his home. Jim schedules his own flights, usually a month or more in advance. He's partial to United Airlines but is always on the lookout for bargain fares so that the company will allow him to continue to live in Chicago. Jim quickly learns most software

but becomes very impatient when he finds a bug or when a website is slow.

This persona description gives us a good introduction to Jim. But nothing speaks as loudly as a picture, so you should also find a picture of Jim and include that with the persona definition. You can get photographs all over the web or you can cut one from a magazine. A solid persona definition combined with a photograph will give everyone on the team a thorough introduction to the persona.

Most persona definitions are too long to fit on a 4" x 6" note card so I suggest you write them on a piece of paper and hang them in the team's common space. You do not need to write persona definitions for every user role. You should, however, write them for the primary roles—that is, the users it is important to satisfy with the software.

A story becomes much more expressive when put in terms of a user role or persona. After you have identified personas and user roles your goal should be to banish the words "the user" from the team's vocabulary. Rather than writing stories like "Users can save itineraries and use them to book subsequent trips" you should write "A Repeat Traveler can save an itinerary and use it to book a subsequent trip." Or even better, "Jim can save an itinerary and use it to book a subsequent trip."

Extreme Characters

Djajadiningrat et al. has proposed the use of extreme characters when considering the design of a new system.[4] They describe an example of designing a Personal Digital Assistant (PDA) handheld computer. Instead of designing solely for a typical sharp-dressed, BMW-driving management consultant the system designers should consider users with exaggerated personalities. Speficially, the authors

4. Djajadiningrat, J.P., Gaver, W.W. and Frens, J.W., 2000. "Interaction Relabelling and Extreme Characters: Methods for exploring aesthetic interactions" in *Proceedings of DIS 2000, Designing Interactive Systems*, pp. 66–71. New York, NY, ACM.

suggest designing the PDA for a drug dealer, the Pope, and a twenty-year old woman who is juggling multiple boyfriends.

It is very possible that considering extreme characters will lead you to stories you would be likely to miss otherwise. For example, it is easy to imagine that the drug dealer and a woman with several boyfriends may each want to maintain multiple separate schedules in case the PDA is seen by the police or a boyfriend. The Pope probably has less need for secrecy but may want a larger font size.

So, while extreme characters may lead to new stories it is hard to know whether those stories will be ones that should be included in the product. It is probably not worth much investment in time but you might want to experiment with extreme characters.

Summary

- ✧ Most project teams consider only a single type of user. This leads to software that ignores the needs of at least some user types.
- ✧ To avoid writing all stories from the perspective of a single user, identify the different user roles who will interact with the software.
- ✧ By defining relevant attributes for each user role you can better see the differences between roles.
- ✧ A user role map is a useful technique for discovering and understanding the relationships between user roles.
- ✧ Some user roles benefit from being described by personas. A persona is an imaginary representation of a user role. The persona is given a name, a face, and enough relevant details to make seem real to the project members.
- ✧ For some applications, extreme characters may be helpful in looking for stories that would otherwise be missed.

Developer Responsibilities

- ✧ You are responsible for participating in the process of identifying user roles and personas.
- ✧ You are responsible for understanding each of the user roles or personas and how they differ.
- ✧ While developing the software, you are responsible for thinking about how different user roles may prefer the software to behave.
- ✧ You are responsible for making sure that identifying and describing user roles does not go beyond its role as a tool in the process.

Customer Responsibilities

- ✧ You are responsible for looking broadly across the space of possible users and identifying appropriate user roles.
- ✧ You are responsible for participating in the process of identifying user roles and personas.
- ✧ You are responsible for ensuring that the software does not focus inappropriately on a subset of users.
- ✧ When writing stories you will be responsible for ensuring that each story can be associated with at least one user role or persona.

Questions

5.1 Take a look at the eBay website. What user roles can you identify?

5.2 Draw the user role map for the roles you identified in the previous question.

5.3 Write persona descriptions for three or four the most important user roles.

Chapter 6

Writing Stories

While stories do not follow a prescribed syntax the best stories share some common attributes.

In this chapter we turn our attention to writing the stories. We first consider whether stories should be hand-written on paper note cards or whether they should be stored electronically in some software system. Next we look at the attributes of a good story, which are:

- ✧ Stories are promises to converse, not detailed specifications
- ✧ Stories have value to users or customers
- ✧ Stories are independent from one another
- ✧ Stories are testable
- ✧ Stories are appropriately sized

Format

One of the first decisions a development team will need to make is whether to write stories on paper note cards or to store them electronically. Many in the Extreme Programming community advocate the use of paper note cards because of their simplicity. Extreme Programming places a premium on simple solutions and paper note cards are definitely simple. On the other hand, there are software products designed specifically for tracking stories (XPlanner, Select Scope Manager, and VersionOne) as well as general purpose software that can be used with stories (spreadsheet, defect trackers, and wikis).

One of the main advantages that cards have over software is that their low-tech nature is a constant reminder that stories are imprecise. When shown in software, stories may take on the appearance of IEEE 830-style requirements and those writing stories may add additional, unnecessary detail because of that.

The typical note card can only hold so much writing. This gives it a natural upper limit on the amount of text. This limitation does not exist in most software alternatives. On the other hand, a common practice among those using note cards is to write acceptance tests for a story on the back of the card. In many cases, the size of the card can work against it when writing the test cases.

A project pursuing ISO (International Organization for Standardization), or similar, certification that requires traceability from a requirement statement down through code and tests will probably favor software. It should be possible to achieve ISO certification with hand-written note cards but putting in place and demonstrating adequate change control procedures over a deck of cards probably outweighs the other advantages cards may have.

Similarly, a team that is not collocated will probably prefer software over note cards. When one or more developers, or especially the customer, is remote it is too hard to work with paper.

Stories Are Promises To Converse

Because stories are written as reminders to have a conversation, stories do not need to include all relevant details. However, if at the time the story is written some important details are known they should be included as annotations to the story, as shown in User Story 6.1. The challenge comes in learning to include just enough detail.

USER STORY 6.1 A story card with notes providing additional detail.

A user can pay with a credit card.

Note: Accept Visa, MasterCard, and American Express. Consider Discover.

User Story 6.1 works well because it provides the right amount of information to the developer and customer who will talk about the story. When a developer starts to code this story she will be reminded that it's already been decided to accept the three main cards and she can ask the customer if a decision has been made about accepting Discover cards. The notes on the card allow a developer and the customer to resume a conversation where it left off previously. Ideally, the conversation can be resumed this easily whether or not the same developer was involved in both halves of the discussion. Use this as a guideline when adding detail to stories.

On the other hand, consider a story that is annotated with too many notes, as shown in User Story 6.2. This story has too much detail ("Collect the expiration month and date of the card") and also combines what should probably be a separate story ("The system can store a card number for future purchases").

USER STORY 6.2 A story card with too much detail.

A user can pay with a credit card.

Note: Accept Visa, MasterCard, and American Express. Consider Discover. On purchases over $100, ask for card id number from back of card. The system can tell what type of card it is from the first digit of the card number. The system can store a card number for future purchases. Collect the expiration month and date of the card.

Working with stories like User Story 6.2 is very difficult. Most readers of this type of story will mistakenly associate the extra detail with extra precision. However, in many cases specifying details too soon just creates more work. For example, if two developers discuss and estimate a story that says simply "a user can pay with a credit card" they will not forget that their discussion is somewhat abstract. There are too many missing details for them to mistakenly view their discussion as definitive or their estimate as accurate. However, when a story adds as much detail as User Story 6.2, discussions about the story are much more likely to feel concrete and real.

If we think about the story as a reminder for the developer and customer to have a conversation then it is useful to think of the story as containing:

- a phrase or two that act as reminders to hold the conversation
- notes about issues to be resolved during the conversation

Details that have already been determined through conversations become tests. Tests can be noted on the back of the story card if using note cards or in whatever electronic system is being used. User Story 6.3 and User Story 6.4 show how the excess detail of User Story 6.2 can be turned into tests, leaving just notes for the conversation as part of the front of the story card. In this way, the front of a story card contains the story and note about open questions while the back of the card contains details about the story in the form of tests that will prove it works as expected.

USER STORY 6.3 The revised front of a story card with only the story and questions to be discussed.

A user can pay with a credit card.

Note: Will we accept Discover cards?
Note: UI Constraint: Don't ask for a card type (it can be derived from first digit on the card).

USER STORY 6.4 Details that imply test cases are separated from the story itself. Here they are shown on the back of the story card

Test with Visa, MasterCard and American Express (pass).
Test with Diner's Club (fail).
Test with good, bad and missing card id numbers.
Test with expired cards.

Stories Are Valuable To Customers Or Users

It is tempting to say something along the lines of "Each story must be valued by the users." But, that would be wrong. Many projects include stories that are not valued by users. Keeping in mind the distinction between *user* (someone who uses the software) and *customer* (someone who purchases the software), suppose a development team is building software that will be deployed across a very large user base, perhaps 5,000 computers in a single company. The customer of a product like that may be very concerned that each of the 5,000 computers is using the same configuration for the software. This may lead to a story like "All configuration information is read from a central location." Users don't care where configuration information is stored but customers might.

Similarly, stories like the following may be valued by customers making a purchasing decision but not valued by actual users.

- ✧ Throughout the development process, the development team will produce documentation suitable for an ISO 9001 audit.
- ✧ The development team will produce the software in accordance with CMM Level 3.

What is to be avoided are stories that are only valued by developers. For example, developers should not write stories like these:

- ✧ Users connect to the database through a connection pool.
- ✧ All error handling and logging is done through a set of common classes.

As written, these stories are focused on the technology and the advantages to the programmers. It is very possible that the ideas behind these stories are good ones but they should instead be written so that the benefits to the customers or the user are apparent. Better variations of these stories could be the following:

- ✧ Up to fifty users should be able to use the application with a five-user database license.
- ✧ All errors are presented to the user and logged in a consistent manner.

In exactly the same way it is worth attempting to keep user interface assumptions out of stories, it is also worth keeping technology assumptions out of stories. For example, the revised stories above have removed the implicit use of a connection pool and a set of error handling classes.

Stories Are Independent

As much as possible, care should be taken to avoid introducing dependencies between stories. Dependencies between stories lead to prioritization and planning problems. For example, suppose the customer has selected as high priority a story that is dependent on a story that is low priority. Dependencies between stories can also make estimation much harder than it needs to be. For example, suppose a project includes these three stories:

1. A customer can pay with a Visa card.
2. A customer can pay with a MasterCard.
3. A customer can pay with an American Express card.

Suppose you think it will take three days to support the first credit card type (regardless of which it is) and then one day each for the second and third. With highly dependent stories such as these you don't know what estimate to give each story.

When presented with this type of dependency there are two ways around it:

- Combine the dependent stories into one larger but independent story
- Find a different way of splitting the stories

Combining the stories about the different credit card types into a single large story ("A customer can pay with a credit card") works well in this case because the combined story is only five days long. If the combined story is much longer than that, a better approach is usually to find a different dimension along which to split the stories. Continuing with this example, support for three credit cards could be added with these stories:

1. A customer can pay with one type of credit card.
2. A customer can pay with two additional types of credit cards.

Stories Are Testable

Stories must be written so as to be testable. Successfully passing its tests proves that a story has been successfully developed. If the story cannot be tested how can the developers know when they have finished coding?

Untestable stories commonly show up for non-functional requirements, which are requirements about the software but not directly about its functionality. For example, consider these non-functional stories:

- Users must find the software easy to use.
- Users never have to wait long for any screen to appear.

As written, these stories are not testable. The tests to prove a story is complete do not necessarily have to be repeatable, easy or quick to run. Naturally, repeatable, quick and easy tests are desirable but a user story that says "Novice users are able to complete common workflows without training" can be tested. Testing this story will likely involve having a human factors expert design a test that involves observation of a random sample of representative novice users. That type of test can be both time consuming and expensive but the story is testable and may be appropriate for some products.

The story "users never have to wait long for any screen to appear" is not testable because it says "never" and because it does not define what "wait long" means. Demonstrating that something never happens is impossible. A far easier, and more reasonable target, is to demonstrate that something rarely happens. This story could have instead been written as "New screens appear within two seconds in 95% of all cases."

Stories Are The Right Size

Like Goldilocks in search of a comfortable bed, some stories can be too big, some can be too small, and some can be just right. Story size does matter because if stories are too large or too small you cannot use them in planning. Large stories, also known as *epics*, are difficult to work with because they frequently contain multiple stories. For example, in a travel reservation system "User can search for a hotel" is an epic. Searching for a hotel is central functionality for a travel reservation system and there are probably many search fields and ways of searching. The epic should be split into smaller stories.

Splitting Stories

Epics typically fall into one of two categories:

- The compound story
- The complex story

The compound story refers to epics that comprise multiple shorter stories. For example, a travel reservation system may include the story "A user can enter and store her preferences." During the initial planning of the system this story may be appropriate. But, when the developers talk to the customer they find out that by "store her preferences" the customer meant all of the following:

- One or more credit card numbers and expiration dates
- Whether she prefers a window or aisle seat
- Whether she wants a special meal on flights
- Size of bed preferred
- Whether she is a smoker or non-smoker
- Frequent-flier and frequent-guest numbers
- The size, style and the make of her preferred rental car.

Depending on how long these will take to develop, each could become its own unique story. However, that may just take an epic and go too far in the opposite direction, turning it into a series of stories that are too small. A better solution may be to group the smaller stories as follows:

1. Users can enter one or more credit card numbers and expiration dates.
2. Users can delete or modify previously entered credit cards.
3. Users can enter hotel preferences
4. Users can enter flight preferences.
5. Users can enter rental car preferences.
6. Users can revise previously entered hotel, flight and car preferences.

Unlike the compound story, the complex story is a user story that is inherently large and cannot easily be decomposed into a set of constituent stories. If a story is complex because of uncertainty associated with it, you may want to split the story into two stories: one investigative and one developing the new feature. For example, suppose the developers are given the story "Users can pay with credit cards" but none of the developers has ever done credit card processing before. They may choose to split the stories like this:

1. Investigate credit card processing over the web.
2. Users can pay with credit cards.

In this case the first story will send one or more developers on what Extreme Programming calls a *spike*, which is a brief experiment to learn about an area of the application. When complex stories are split in this way, always define a maximum amount of time (called a *timebox*) around the investigative story, or spike. Even if the story cannot be estimated with any reasonable accuracy, it is still possible to define the maximum amount of time that will be spent learning.

Complex stories are also common when developing new or extending known algorithms. One team in a biotech company had a story to add novel extensions to a standard statistical approach called

Consider putting the spike in a different iteration

When possible, it works well to put the investigative story in one iteration and the other stories in one or more subsequent iterations. Normally, only the investigative story can be estimated. Including the other, non-estimatable stories in the same iteration with the investigative story means there will be a higher than normal level of uncertainty about how much can be accomplished in that iteration.

expectation maximization. The complex story was rewritten as two stories: the first to research and determine the feasibility of extending expectation maximization; the second to add that functionality to the product. In situations like this one it is difficult to estimate how long the research story will take.

The key benefit of breaking out a story that cannot be estimated is that it allows the customer to prioritize the research separately from the new functionality. If the customer is given only the complex story to prioritize ("Add novel extensions to standard expectation maximization") and an estimate for the story (10 days) she may prioritize the story based on the mistaken assumption that the new functionality will be delivered in approximately that timeframe. If instead, the customer is given an investigative, spike story ("research and determine the feasibility of extending expectation maximization") and a functional story ("extend expectation maximization") she must choose between adding the investigative story that adds no new functionality this iteration and perhaps some other story that does.

Combining Stories

Sometimes stories are too small. A story that is too small is typically one that the developer says she doesn't want to write down or estimate because doing that may take longer than making the change. Bug reports are a common example of stories that are often too small. A good approach for tiny stories, common among Extreme Programming teams, is to combine them into larger stories that represent from about a half-day to several days of work. The combined story is given a name and is then scheduled and worked on just like any other story.

For example, suppose a project has five bugs and a request to change some colors on the search screen. The developers estimate the total work involved to be about sixteen hours. These stories should be grouped together and considered one two-day story. If you've chosen to use paper note cards, you can do this by stapling them together with a cover card.

Summary

- ✧ Stories may either be written on paper note cards or stored in a software system.
- ✧ Both note cards and software have their own advantages. There is no compelling advantage for either approach that makes it right for all teams.
- ✧ Teams in regulated environments or distributed teams should probably store user stories electronically.
- ✧ Stories may be annotated with details but too much detail obscures the meaning of the story and can give the impression that no conversation is necessary between the developers and the customer.
- ✧ One of the best ways to annotate a story is to write test cases for the story.
- ✧ Stories should be written so that their value to users or the customer is clear.
- ✧ Ideally, stories are independent from one another. This isn't always possible but to the extent it is, stories should be written so that they can be developed in any order.

- Stories should be written so that they are testable.
- Stories should generally take from one-half to perhaps five or ten days to be developed.
- If they are too big, compound and complex stories may be split into multiple smaller stories.
- If they are too small, multiple tiny stories may be combined into one bigger story.

Developer Responsibilities

- You are responsible for choosing appropriately between note cards and software based on the unique needs of the project.
- You are responsible for helping write stories that are promises to converse rather than detailed specifications, have value to users or the customer, are independent, are testable, and are appropriately sized.
- If tempted to write a story about the use of a technology or a piece of infrastructure, you are responsible for instead writing the story in terms of its value to users or the customer.

Customer Responsibilities

- You are responsible for choosing appropriately between note cards and software based on the unique needs of the project.
- You are responsible for helping write stories that are promises to converse rather than detailed specifications, have value to users or to yourself, are independent, are testable, and are appropriately sized.

Questions

6.1 Either rewrite each of the stories below so it focus on value to either the user or the customer, or explain who would value the story as written.

a The database is backed up automatically once a day.

b All runtime errors are logged to an XML file.

c The program can share the same database with other products sold by our company.

d Credit card numbers that are reported as stolen are flagged and cannot be used later.

e The Java servlet-based middle tier will run on any J2EE-compatible application server.

f The user must be authenticated before she can use the system.

6.2 Break this epic up into two or more component stories: "The user can make and change reservations for a rental car."

Chapter 7

Gathering Stories

Users do not always know their exact needs and stories are not out there just waiting to be found. You need to know the right techniques to help users discover their needs.

How do you gather the stories? It's not sufficient to hand the customer a stack of note cards and say "have at it." This chapter offers advice on working with users and the customer and how to identify stories in your conversations with them. The advantages of various approaches will be described. This chapter describes effective methods for getting at a user's real needs by asking the right types of questions.

Elicitation and Capture Should Be Illicit

Even some of the best books on requirements use words like *elicitation*[1,2,3] and *capture*[4] to describe the practice of identifying

1. Kovitz, B.L., 1999. *Practical Software Requirements: A manual of content and style*. Greenwich, CT, Manning.

requirements. Terms like these imply requirements are out there somewhere and all we need to do is have them explained to us and then we can lock them in a cage. Requirements are not out there in the project space waiting to be captured. Similarly, it is not the case that users already know all the requirements and we need only elicit them.

Robertson and Robertson[5] introduce the term trawling to describe the process of gathering requirements. Trawling for requirements leads to the mental image that requirements are captured in a fishing net being pulled behind a boat. This metaphor works on a variety of levels.

First, it is consistent with the idea that different-sized nets can be used to capture different-sized requirements. A first pass can be made over the requirements pond with a large mesh net to get all the big ones. You can get a feel for the needed software from the big requirements and then make a subsequent pass with a smaller mesh net and get the medium-sized requirements, still leaving the small ones for later. This metaphor works whether we think of size as business value, essentialness to the software, and so on.

Second, trawling for requirements expresses the idea that requirements, like fish, mature and possibly die. My net may miss a requirement today because the requirement is not important to the system. However, as the system grows in unpredictable directions based on the feedback from each iteration, some requirements will grow in importance. Similarly, other requirements that were once considered important will decrease in importance to the point where we can consider them dead.

---

2. Lauesen, S., 2002. *Software Requirements: Styles and techniques.* London, England, Addison-Wesley.
3. Wiegers, K.E., 1999. *Software Requirements.* Redmond, WA, Microsoft Press.
4. Jacobson, I., G. Booch, J. Rumbaugh, 1999. *The Unified Software Development Process.* Reading, MA, Addison-Wesley.
5. Robertson, S. and J. Robertson, 1999. *Mastering the Requirements Process.* Reading, MA, Addison-Wesley.

Finally, the metaphor of trawling for requirements captures the important reality that skill plays a factor in finding the requirements. A skilled requirements trawler will know where to look for requirements while the unskilled trawler will waste time with inefficient techniques or in the wrong locations. This chapter is about learning the techniques that make us efficient in trawling for user stories.

A Little Is Enough, Or Is It?

One of the easiest ways to spot a traditional plan-driven process is to look at its approach to requirements. Plan-driven processes are characterized by their heavy emphasis on getting all the requirements right and written early in the project. Agile projects, on the other hand, acknowledge that it is impossible to use a net with such a fine mesh that we can get all of the user stories in one pass. Agile processes also acknowledge that there is a time dimension to stories: the relevance of a story changes based on the passage of time and on what stories were added to the product in prior iterations.

However, even though we acknowledge the impossibility of writing all of the stories for a project, we should still make an initial upfront attempt to write those that we can, even if many are written at a very high level. As we saw in Chapter 3, "Why User Stories?," a tremendous advantage of user stories is that it is very easy to write them at different levels of detail. Because of this, it is very easy to write stories for a great deal of an application with less work than with other requirements techniques.

This is not a recommendation to start a new project by spending three months writing user stories. Rather, it means to look into the future for approximately one release (perhaps six months) and then write user stories that decrease in detail as the time horizon increases. For example, if the customer or users have said they "probably want reports in this release" then write a card that simply says "Users can run reports." But stop there: don't determine if they need to configure their own reports, whether reports are formatted in HTML, or whether reports can be saved.

Techniques

Because stories will be evolving, coming and going throughout the project we need a set of techniques for gathering them that can be used iteratively. The techniques we use must be sufficiently light-weight and non-obtrusive that they can be applied more or less continuously. Some of the most valuable techniques for gathering stories are:

- User interviews
- Questionnaires
- Observation
- Story-Writing Workshops

Each of these techniques will be considered in the following section.

User Interviews

Interviewing users is the default approach many teams take to trawling for stories and is probably one you will want to use. One of the keys to success with interviews is the selection of interviewees. As discussed in Chapter 4, "Who's The User?," there are many user proxies available; but, you should obviously interview real users whenever possible. You should also interview users who fill different user roles. A user role map, as introduced in Chapter 5, "User Roles and Personas," is helpful in this regard.

It is not sufficient to ask the user "So, what do you need?" Most users are not very adept at understanding, and especially at expressing, their true needs. I learned this once from a user who walked into my office and acknowledged, "You built exactly what I asked for but it's not what I want."

At another company they were developing software for delivering surveys. Each survey would be delivered over the phone, via email, and via interactive voice response. Different types of users would use different survey types. The surveys were very complicated: specific answers to one set of questions would determine which question would be asked next. The users needed a way to enter the surveys and they presented the development team with examples of a complicated mini-language they proposed using to formulate questions. This entirely text-based approach seemed needlessly complicated to one of the developers. The developer showed the users how they could instead create surveys visually by dragging and dropping icons that represented different types of questions in a survey. The users ripped up their mini-language and worked with the developers to create a visual survey design tool. Just because these users had the problem does not mean they were uniquely qualified to propose its solution.

Open-Ended and Context-Free Questions

The best technique for getting to the essence of a user's needs is through the questions you ask her. One project team was torn between putting their application in a browser or writing it as a more traditional platform-specific program. They struggled between the ease-of-deployment and lower training costs provided by the browser-based version and the more powerful platform-specific client. The intended users would certainly like the advantages of the browser but they also valued the richer user experience provided by the platform-specific client.

It was suggested that the target users for the product be asked their preference. Since the product would be a new generation rewrite of a legacy product the marketing group agreed to contact a representative sample of current users. Each user in the survey was asked "Would you like our new application in a browser?"

This question was like going to your favorite restaurant and having the waiter ask if you'd like to have your meal for free. Of course, you would! And of course the surveyed users responded that they would love to have the new version of the software in a browser.

The mistake the marketing group made was that they asked a closed-ended question and failed to provide sufficient detail for it to be answered. The question assumed that anyone being interviewed would know the tradeoffs between the browser and the unstated alternatives. Perhaps a better version of the question could have been:

> Would you like our new application in a browser rather than as a native Windows application even if it means reduced performance, a poorer overall user experience, and less interactivity?

This question still has a problem because it is closed-ended. The respondent is given no room for anything other than a simple yes or no. It is far better to ask open-ended questions that let respondents express more in-depth opinions. For example, "What would you be willing to give up in order to have our next generation product run within a browser?" A user answering that question can go in a variety of directions. Where she goes—and does not go—with her answer will provide you with a more meaningful answer to the question.

It is equally important to ask context-free questions, which are ones that do not include an implied answer or preference. For example, you would not ask, "You wouldn't be willing to trade performance and a rich user experience just for having the software in a browser, would you?" It's pretty clear how most people are going to answer that question.

Similarly, instead of asking "How fast do searches need to be?" ask "What kind of performace is required?" or "Is performance more important in some parts of the application?" The first question is not context-free because it implies there is a performance requirement to searching. There may not have been but, having been asked, a user is unlikely to say so; she's more likely to take a guess.

At some point you will need to move from context-free questions to very specific questions. However, by starting with context-free questions you leave open the possibility for a wider range of answers from users. That will lead you to stories that may have remained undiscovered if you jumped right into very specific questions.

Questionnaires

Questionnaires can be an effective technique for gathering information about stories you already have. If you have a large user population then a questionnaire can be a great way to get information about how to prioritize the stories. Questionnaires are similarly useful when you need answers from a large number of users to specific questions.

However, you questionnaires are usually inappropriate as a primary technique of trawling for new stories. Questionnaires do not lend themselves to followup questions. Also, unlike in a conversation, it is impossible to follow a user down an interesting path that comes up.

As an example use of a questionnaire, you may survey current users about how often they use features in the software today and what their reasons are for not using some feature more. This may lead to prioritizing some usability stories higher than they have been prioritized in the past. As another example, a questionnaire that asked "What new features would you like to see?" will be of limited use. If you give the user a list of choices then you may miss hearing about the five critical features you've never thought of. Alternatively, if the user is allowed to respond with free-form text it will be difficult to tabulate answers.

Observation

Observing users interact with your software is a wonderful way to pick up insights. Every time I have had the chance to observe someone using my software I have left flush with ideas about how to improve their experience, productivity, or both. Unfortunately, opportunities for user observation are rare unless you are developing for in-house customers. Too many commercial products take the approach that it's possible to guess what users want. If you have the chance to observe users work with your software, take it.

At one company the users were nurses working in a call center. The nurses answered medical questions from callers. The nurses indicated that they needed a large text field that could be used for documenting the results of the call when the call was finished. An initial version of the software included a large text field on the call wrapup screen. However, after the initial release each member of the development team spent a day observing the users. One of the things discovered was that the large text field was used for entry of things that could have been tracked by the system. By observing the users, the developers discovered that the real need was for the system to track the decisions made by the user as she worked with the software. The large text field was replaced with a feature that logged all searches and recommendations the nurse selected. The real need—tracking all instructions given to a caller—had been obscured by the nurses' description of the need and only came out during observation.

Story-Writing Workshops

A story-writing workshop is a meeting that includes developers, users, the product customer and other parties who can contribute by writing stories. During the workshop the participants write as many stories as they can. No priorities are associated with the stories at this point; the customer will have a chance to do that later.

A properly conducted story-writing workshop can be a very rapid way to write a great number of stories. From my experience a good story-writing workshop combines the best elements of brainstorming with low-fidelity prototyping. A low-fidelity prototype is done on paper, note cards, or a white board and maps very high level interactions within the planned software. The prototype is built up iteratively during the workshop as the participants brainstorm the things a user may want to do at various points while using the application. The idea is not to identify actual screens and fields, as in traditional prototyping or Joint Application Design (JAD) sessions. Rather, the conceptual workflows are identified. Figure 7.1 shows the start of a low-fidelity prototype for a travel reservation system.

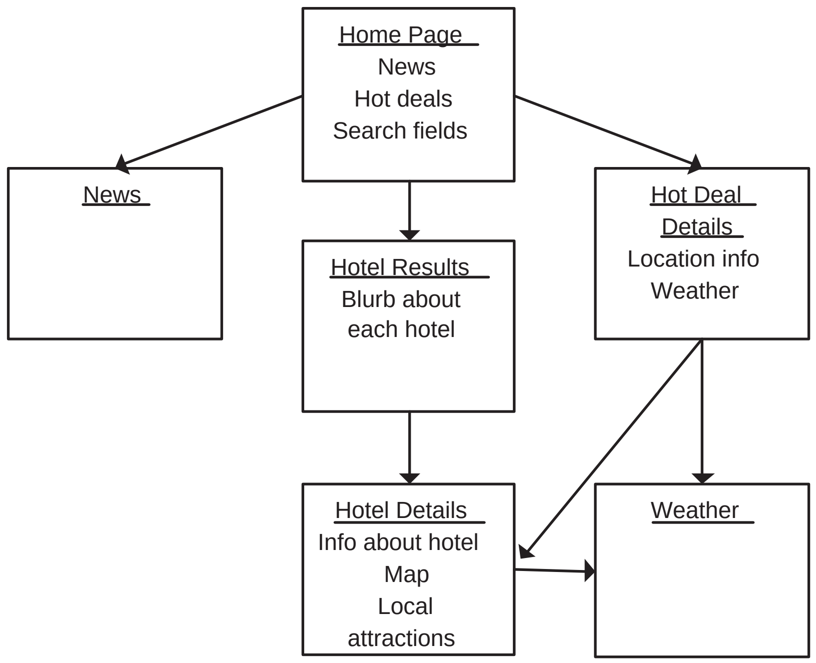


FIGURE 7.1 A low-fidelity prototype.

Each box represents a new component of the travel website. The component's title is underlined in the box. Below the title is a very short list of what the component does or contains. The arrows between the boxes represent links between the components. A link may indicate that a new page appears or that the information is displayed on the same web page. For example, the Hotel Results component shows a blurb about each hotel that matches some search criteria. The Hotel Results could be an entire screen in the system or it could be a part of the home page. The distinction is not important as you'll have plenty of time to go over that with the customer later.

To start low fidelity prototyping draw an empty box, tell the participants that it is the main screen of the software, and ask them what can a user do from there. It does not matter that you haven't figured out what the main screen is yet and what's active on that screen. The meeting participants will start throwing out ideas about what actions users can take. For each action draw a line to a new box, label that box, and write a story. So, while Figure 7.1 may not indicate if the Hotel Results are displayed on the Home Page, the discussion that leads to Figure 7.1 will lead to stories such as the following:

- After searching for a hotel a user sees a list of hotels that match the search criteria. Each hotel is described by a short blurb.
- Users can search for hotels with one or more search criteria.
- Users can view detailed information about a hotel including a map and attractions near the hotel.

None of these stories requires knowledge about how the screens will be designed. However, walking through the workflows will help everyone involved think of as many stories as possible. You should take a depth-first approach: For the first component write down its salient details then move to a component connected to the first and do the same. Then, move to a component connected to that one, rather than going back to the first component and first describing each component connected to it.

Throw It Away

Be sure to throw away or erase the low-fidelity prototype within a few days of creating it. A prototype is not a long-term artifact of your development process and you don't want to cause any confusion by keeping it around. If you leave a story-writing workshop with the feeling that you didn't finish then keep the prototype around for a few more days, revisit it, try to write any missing stories, and then get rid of it.

A low-fidelity prototype does not need to go into the trash at the end of the day you draw. But, it needs to end up there soon.

As you walk through the prototype ask questions that will help you identify missing stories, such as:

- What will the user most likely want to do next?
- What mistakes could the user make here?
- What could confuse the user at this point?
- What additional information could the user need?

Think about the user roles and personas as you ask these questions. Many of the answers can change based on the user role being considered.

Maintain a parking lot of issues to come back to. For example, in discussing the travel reservation system someone may ask if the system will support cruise ship reservations. If no one has thought of that prior to the workshop, write it down some place where you can see it and can come back to it later—either at the end of the workshop or after some followup work after the workshop.

During a story writing workshop the focus should be on quantity rather than quality. Just let the ideas come and write them down. A story that seems like a bad idea now might seem brilliant in a few hours or it may be the inspiration for another story. Additionally, you do not want to get bogged down in lengthy debate over each story. If a story is redundant with or becomes replaced by a better story later then you can just rip up the story. Similarly, when the customer prioritizes stories for a release she can assign a low priority to low quality stories.

Occasionally some participants in a story-writing workshop have a hard time either getting started or moving past a sticking point. In this case it can be very beneficial to have access to competitive or similar products.

Pay attention to who is contributing during a story writing workshop. Occasionally a participant will remain silent through much or all of the meeting. If this is the case, talk to the participant during one of the breaks and make sure she's comfortable with the process. Some participants are reluctant to speak up in front of their peers or supervisors, which is why it is important that story ideas not be judged during these sessions. Once participants become comfortable that their ideas will simply be noted, not debated, at this point they will contribute more readily.

Finally, let me reiterate that discussion during a user story workshop should remain at a very high level. The goal is to write as many user stories in as short a time as possible. This is not the time to design screens or solve problems.

Summary

- The idea of eliciting and capturing requirements is wrong. It leads to the twin fallacies that users already know all the requirements and that requirements can be captured and locked in a cage where they will remain unchanged.
- The metaphor of trawling for requirements is far more useful: it captures the ideas that there are different sizes of requirements, that requirements may change over time, and that it takes skill to find the requirements.
- While agile processes are supportive of requirements that emerge late in the process, you should still start by looking forward to approximately the end of the intended release and write the user stories you can easily see.
- User stories can be found by interviewing users, observing users, questionnaires, and holding story-writing workshops.
- The best results are acheived by using a combination of methods rather than overreliance on any one method.
- The most useful answers are given in response to open-ended, context-free questions, such as "Tell me about how you'll search for a hotel?" rather than "Will you search for a hotel by chain name?"

Developer Responsibilities

- ✧ You are responsible for understanding and using multiple techniques to use while trawling for user stories.
- ✧ You are responsible for knowing how to best make use of open-ended and context-free questions.

Customer Responsibilities

- ✧ You are responsible for writing as many user stories as early as possible.
- ✧ As the main representative of the software's users, you are responsible for understanding your options in communicating with them.
- ✧ You are responsible for knowing how to best make use of open-ended and context-free questions.
- ✧ If you need or want help in writing the stories you are responsible for scheduling and running one or more story-writing workshops.
- ✧ You are responsible for making sure all user roles are appropriately represented while trawling for stories.

Questions

7.1 On a project being developed for use within the company, the developers are all in Chicago and the customers are spread between New York and Los Angeles. What techniques would you use to write the stories for this project? Why?

7.2 Rephrase the following questions to be open-ended. Do you think the user should have to enter a password? Should the system automatically save the user's work every

15 minutes? Can one user see database entries saved by another user?

7.3 Rephrase your answers to the previous question to now also be context free.

7.4 When might it be best to ask the context-free form of these questions?

7.5 Draw a low fidelity prototype for the website of a dental practice.

Part 2

Estimating and Planning

Chapter 8

Principles of Estimating

Before estimating, everyone on the team must agree on what the estimates mean.

Before estimating anything, you must first lay down the ground rules. For example, everyone on the team must agree and know the answers to the following questions:

- Are we estimating duration or magnitude?
- Are we estimating the amount of time the task will take if it's the only thing someone works on or are we including all the usual interruptions?
- How confident should we be in each estimate?
- What is the unit of measure we're using?

Duration or Magnitude?

If we choose to estimate the duration it will take to complete a story we end up with estimates like "Adding a login screen will take

6 hours." Alternatively, estimating the magnitude of each story results in estimates like "Adding a login screen is a small task" and "Adding that new report is a medium task." Another variation of estimating magnitude involves picking a random scale (say one through 10, for example) and estimating purely by relative complexity: "That story is a 4 and this one seems like it will take twice as long so it must be an 8."

There are three problems with estimating magnitude. First, the values on the magnitude scale you select must be sufficienty meaningful that you can truly distinguish a story of one magnitude from a story of the next magnitude. This means that a scale of 1 to 100 is probably inappropriate because it will be too difficult to distinguish a 67 story from a 68 story.

Second, at some point you must convert a magnitude effort into a duration effort for it to be of any use. You cannot go to the customer or your boss with "We'll be done in 8 mediums, 3 larges, and 6 smalls." They want to know how many days or weeks the project will take.

In order to move from magnitude to duration you're forced to make relatively arbitrary decisions such as saying that a long task will take 5 days, a medium task 3 days, and a short task 1 day. The problem is only slightly better if you estimate in an arbitrary measure of magnitude such as 1-10 or 1-100 Gummi Bears.[1] In these cases you can guess how many of your chosen unit you will complete per iteration but until you've run a few iterations you'll be completely guessing at that number.

The third and final problem with estimating magnitude is that developers may perform an implicit conversion into duration when asked for the estimate. For example, suppose that each developer on a team of four knows that the team completes 40 units each two-week iteration. Those four developers will very quickly make the

1. Jeffries, R., Anderson, A. and Hendrickson, C. 2001. *Extreme Programming Installed.* Upper Saddle River, NJ, Addison-Wesley.

leap to thinking of 10 units as equalling two weeks of one developer's time.

It is far better to skip all this hassle and estimate duration directly.

Ideal Time

XP projects traditionally estimate durations using *ideal time*. Ideal time is meant to capture the reality that developers never get to spend all of their time developing. Take a look at Figure 8.1, which shows fictional programmer Rob's schedule for next week.

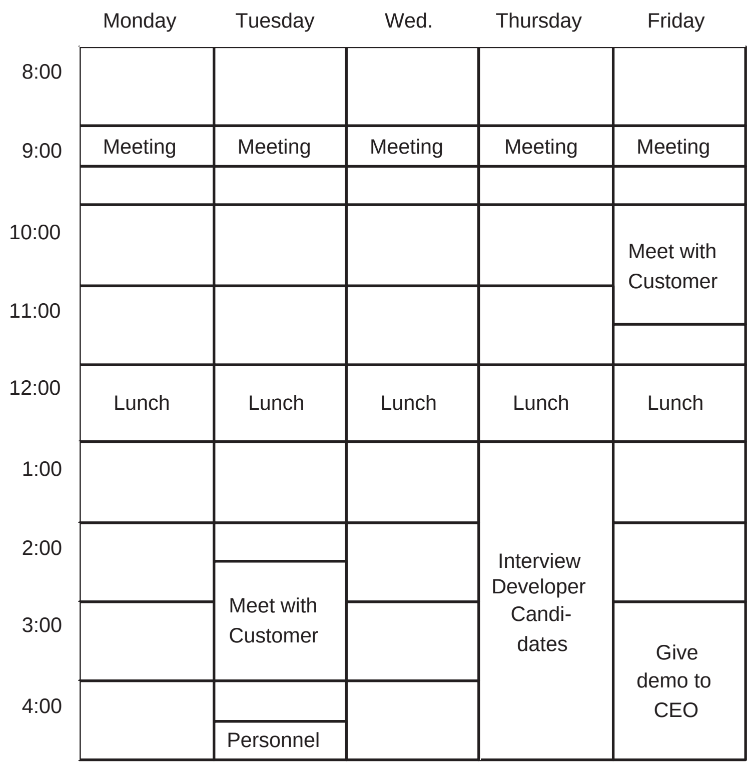


FIGURE 8.1 Rob's schedule for next week already includes many planned interrruptions.

Rob already has eleven hours of meetings scheduled. Now add an hour each day for responding to email and assume a few other interruptions and half of his week is gone. If Rob has committed to completing a 40-hour story next week we can look at this and know he won't be able to do it without working 20 hours of overtime. On the other hand, if we know that Rob gets about 20 hours of hands-on programming time each week then we know that when he gives an estimate of 40 hours it will take two weeks of elapsed time to complete that 40 hours of ideal time.

It is far simpler to estimate the ideal time for a story than it is to estimate the elapsed time on a story. If you ask me how long it will take to code a new screen in our system and I want to give an estimate in ideal time all I need to think about is that screen, its requirements, and how it will be coded and tested. If my estimate will be in elapsed time then I need to think about all that plus all other claims on my time while I'll be working on the task.

Experienced Senior Programmer Days

The main problem with estimating in ideal time is that most teams never specify whose ideal time it is they are estimating. Are we estimating how long it will take Rob, our superstar programmer who has years of experience both with the language and the domain, or are we estimating how long it will Jody, a recent college graduate? Or does Rob estimate in Ideal-Rob-Time and Jody in Ideal-Jody-Time? If so, how do we add Ideal-Rob-Time to Ideal-Jody-Time?

You can avoid this problem by providing estimates in *Experienced Senior Programmer Days.* Prior to estimating, the team hypothesizes an archetypal experienced, senior programmer. To make sure everyone is thinking the same way, the programmer is defined in terms of her experience level with the relevant technologies and the domain. For example, an archetypal programmer might be described as a solid Java programmer with three years of Java experience, moderate exposure to SQL but hasn't done anything advanced with it, and has worked at the company for one year. In some cases the archetypal programmer can be defined in terms of someone already on the team.

Factors affecting elapsed time

There are many factors that make calendar time differ from elapsed time. Among them are the following:

- Vacations
- Sick time
- All-company meetings
- Department meetings
- Human resources
- Training
- Email
- Phone calls
- Demos
- Special projects
- Reviews, inspections and walkthroughs
- Interviewing new candidates
- Spikes
- Leaves of absence
- Sabbaticals

The exact definition of the archetypal programmer does not matter except that everyone agrees on the definition. I always target her to be somewhere near the middle of the team so that everyone can identify with her. For most teams that works out to be an Experienced Senior Programmer; but, if your team has a different profile then use a different archetypal programmer.

Expressing estimates for a faceless archetypal programmer who doesn't exist definitely introduces a new potential source of error in the estimates. Most of us have a hard time estimating how long we'll take to do something, so estimating how long it will take someone else might seem to be even more difficult. Fortunately, this isn't the case and it is actually easier to give estimates of how long it will take the archetypal Experienced Senior Programmer.

When I'm freed from giving estimates in how long it will take *me* to do something, I feel freer to express an honest opinion. Developers suffer from a bias for giving short estimates for many reasons—a desire to please, an incomplete understanding of the problem, pure optimism, and a fear of getting questioned about an estimate that is "too long." If I'm not estimating how long it will take *me* to finish a story then much of this bias toward short estimates goes away.

Also, remember that we are not looking for highly accurate estimates at this point. Our goal is simply to put an estimate with each story so that the stories can be prioritized and then batched together into a tentative release plan.

Confidence

We've agreed that we'll estimate in Ideal Time for an archetypal-Experienced Senior Programmer but we still haven't removed all room for misunderstanding. If a manager asks for an estimate does she want one we can be totally confident an Experienced Senior Programmer can meet 100% of the time? Or does she want one that an Experienced Senior Programmer can meet 50% or 75% of the time?

The probability distribution of possible completion times looks something like Figure 8.2.

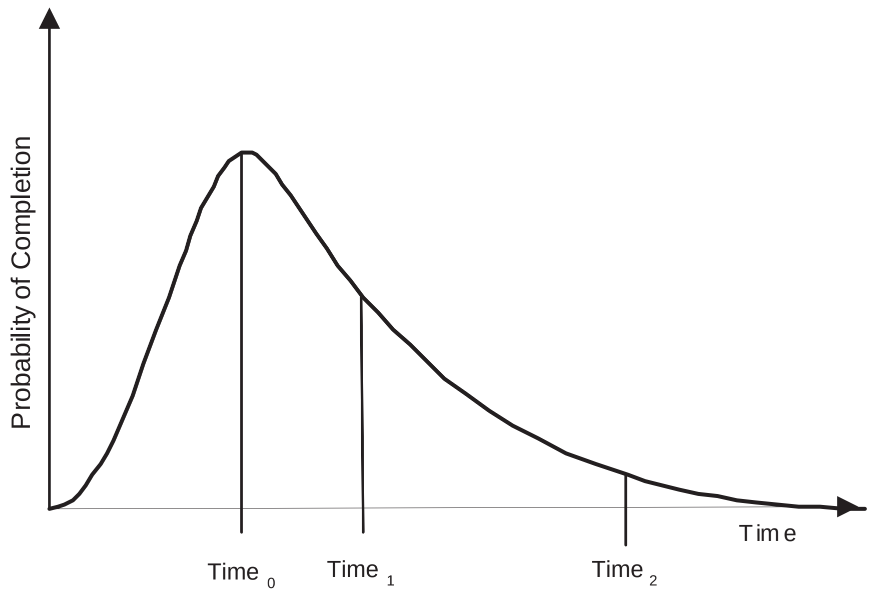


FIGURE 8.2 Probability distribution of possible completion times.

The curve takes on this general shape because there is normally not much that can be done to accelerate the completion of a task but there are an infinite number of things that can go wrong and delay the completion of a task. For example, just as I'm about to complete the story my computer crashes and I lose unsaved changes. Then lightning hits our building and fries our source repository. We request a backup tape to be delivered tomorrow morning but the delivery service loses the tape but I don't care as I'm run over by the proverbial bus on the way to work.

Figure 2 shows that the most common duration for the story will be at Time$_0$. However, since less than 50% of the curve is to the left of Time$_0$ there is greater than a 50% chance that the story will not be complete by then. Time$_1$ indicates the duration where half the time the story is finished ahead of schedule and half the time it is finished behind schedule. When asked for an estimate, many developers will estimate the time to either Time$_0$ or Time$_1$. But, of course, out there at Time$_2$ is the duration that is sufficient to complete the task even if many things go wrong.

As an example, suppose a programmer is staring at a story card that says "The user can view detailed information about all of his past stays at any hotel." She thinks about it and decides there is a good chance the story can be finished in two ideal days. (We're not yet concerned with how many calendar days that may be.) But, the programmer can certainly think of enough things that either may be implied by the story or that will be complicated when it's coded that she would not be surprised if it took three days. It would, however, be surprising if it took five days but even that could happen if the story is more complicated than it seems from thinking about it briefly. In this case, Time$_0$ is two days, Time$_1$ is three days, and Time$_2$ is five days. Even though the single most likely estimate for how long the story will take is two days there is a good chance it will take longer.

Another way to think about it is to assume that 100 different Experienced Senior Programmers each program this story. How long would it take each to complete it? The results might be similar to those shown in Table 8.1. Here we see that 40 of them finish on the second day. But if we use two days as our estimate we're giving an estimate that will hold true for less than 50% of the cases. We don't get 50% (Time$_1$) until sometime the third day.

TABLE 8.1 Number of developers finishing a story on a given day.

Day	Number Finishing
Day 1	5
Day 2	40
Day 3	25
Day 4	15
Day 5	10
Day 6	5

So, how confident should we be in each estimate?

Clearly we don’t want to give estimates we can be 100% sure of. My local grocery store is 2 miles away and is usually a short five minute drive. Sometimes it takes six or seven minutes if I get stuck behind a slow driver. But my 100% confident estimate skyrockets to a week or more. If you need me to be 100% confident then I need to factor in worst case scenarios. For example, I may get in an accident and end up in the hospital for a few days and it takes me a week to recover and then try the five minute drive a second time. An estimate that is 100% reliable is impossible because I can always think of of one more delay. So, an estimate we’re 100% confident of does no one any good.

On the other hand, I feel 90% certain that I can make it to the grocery store in 7 minutes. That information is far more useful than the 100% estimate. Similarly, it's useful to know that 50% of the time I can be there in 5 minutes.

Since both the 50% and 90% estimates convey extremely useful information—and because most developers find it easier to give an estimate range rather than a specific estimate—you should estimate at both levels of confidence. We’ll see an easy way of doing this, and how the two estimates combine to create a usable release plan, in Chapter 11, “Planning a Release.”

The Unit of Measure

Since we’re estimating duration rather than magnitude, selecting the unit of measure is actually pretty easy. Weeks are too big to be useful. We’ll probably have plenty of stories we want to estimate at one week and we may even have a few that we want to estimate at two weeks. However, any stories bigger than that will almost certainly be split. A unit of measure that only gives us two choices—one week or two—is not very useful so we won’t use weeks.

How about hours? Clearly, if you estimate in hours you’ll have tremendous precision to your estimates. Right? Not really—what you’ll have is false precision. Suppose I estimate the story “the user can view detailed information about all of his past stays at any hotel” at 13 hours. Your first question to me should me how do I know it won't take 14 hours? Or 12 hours? I almost certainly don't know and when I give the estimate in hours I give the false impression that my estimate is accurate at that level of precision.

Double the Unit of Measure

Back in the 1980s I was introduced to a very unusual but surprisingly effective technique for estimating. My boss would take any estimate given to him, double it and then increase the unit of measure. So, if a programmer gave him an estimate of two days he would translate that into an estimate of 4 weeks. Similarly, a 4 hour estimate was turned into an 8 day estimate. As much as it sounds like this would lead to overly large estimates this project was staffed entirely with programmers with less than two years of experience and this technique was reasonably accurate.

If hours are out and weeks are out, what about estimating in days? Suppose instead of estimating 13 hours for that story I told you two days. If you ask me why it won't be one day or three days I can answer that:

"It won't be one day because we don't have a screen that shows detailed stay information so I need to code that from scratch. The query shouldn't be too hard but I have to find out if 'at any hotel' means 'any single hotel' or if the user can select a set of hotels perhaps with a checkbox next to each. Even if it's the latter and I have to code a new screen for that I just can't see this taking three days."

My recommendation is to estimate stories in days. You may occasionally feel the need to estimate some stories in half–days. At least initially, you should allow this only for stories you feel are between 0 and 1 day. That is, you can have a half-day story but avoid stories that are 3-1/2 days. For any story longer than a day simply estimate in full days. Once you've got enough experience estimating with stories you may want to allow half day estimates but only do it when you have a good answer if someone asks you, “Why 3-1/2 days instead of 3 or 4 days?”

Story Points

Let’s recap. We now know we are going to estimate:

- Duration rather than magnitude.
- In Ideal Time for an archetypal programmer defined specifically for the project.
- Using one-day increments, except for stories small enough to be considered as a half day.

It gets tedious giving estimates like “That story will take an archetypal experienced, senior programmer 3 ideal days.” It is preferrable to use the term *story point* to refer to one ideal day of time by the project’s archetypal programmer. Note that this is consistent with but slightly different from Wake’s definition of a story point as an ideal week.[2]

Summary

- Estimating the amount of calendar time is difficult. We will instead estimate in ideal time.
- Ideal time represents pure development time, not time spent answering email, attending corporate-wide meetings, giving demos, interviewing candidates, and so on.
- To avoid problems of adding ideal time from two developers with different levels of proficiency, estimates are given for an archetypal programmer.

2. Wake, W.C. 2002. *Extreme Programming Explored*. Upper Saddle River, NJ, Addison-Wesley.

- When communicating about estimates it is important to be in agreement on the level of confidence associated with each estimate. It is best to estimate stories at both the 50% and 90% levels of confidence.
- A story point is equivalent to one ideal day of work by the project's archetypal programmer.

Additionally, each of these assumptions will be made explicit so that everyone on the project estimates the same way.

Developer Responsibilities

- You are responsible for knowing all of the parameters surrounding any estimate you are asked to give. For example, are you being asked to give an estimate you are 90% confident in or one you are 50% confident in?
- You are responsible for knowing how any estimate you give will be used.
- You are responsible for clearly expressing any estimate you give. Do not say "That will take three days." Instead say, "That will take three ideal days."

Customer Responsibilities

- You are responsible for understanding the nature of any estimate you are given. Know if the estimate is in ideal or calendar time; know how confident the estimator is in the estimate.
- You are responsible for asking clarifying questions until you understand the assumptions underlying an estimate.
- You must treat estimates as what they are—estimates and not guarantees.

Questions

8.1 What accounts for the difference in the duration between an estimate you are 50% confident of and one you are 90% confident of?

8.2 Write a two or three sentence description of the archetypal programmer who would be appropriate to use for estimating ideal time on a project you are familiar with.

8.3 How long will it take you to travel from your current location to a town 1,000 miles away? Justify your estimate.

Chapter 9

Estimating User Stories

"Everything in the world takes four hours." —Paul Buchman

Having established the ground rules we'll use to estimate we are ready to move on to the actual estimation. In order to have estimates that can be combined and assembled into a release plan it is not necessary to put a lot of work into estimating. It is not, for example, necessary to create a work breakdown structure for each story and then estimate each task. This chapter looks at four techniques that can be used independently or combined to quickly and easily estimate stories. The techniques considered are the following:

- Estimation by gut feel
- Analogy
- Decomposition
- Wideband Delphi

Estimating by Gut Feel

Everyone has used this approach sometime before. You look at a story or task and a duration comes to mind. Typically this happens when your boss or customer asks you for an estimate but adds “Don't worry, it’s just a preliminary estimate and we won’t hold you to it.”

The best use for a gut feel estimate is often as a reasonableness check on estimates derived in other ways. You should always do a gut check on any estimate you come up with. However, while estimating by gut feel has some value it is not hard to improve upon so you should not rely on it too heavily.

Gut feel and project management tools

Back in 1994 I still believed one could identify all the tasks on a complex project, enter them into a project management tool like Microsoft Project™, and get an accurate deadline for a project. I dutifully entered a thousand tasks into a project and told the tool to take care of leveling resources. Leveling is the process of making sure no person (or other resource) on the project is planned to be used for more time in a week than it is available. I scrolled to the right on the generated Gantt chart to find the deadline and found out the project would take two years.

A quick gut check told me that was wrong. There was no way the project could take two years. I looked into it and found a problem with how the tool leveled resources. This problem stretched the project out from about nine months to two years. A quick gut check had told me the schedule estimated by the project planning tool was wrong.

Estimating by Analogy

A significant improvement over gut feel comes from estimating by analogy. When estimating by analogy you compare the story you are considering to one or more other stories. Ideally you can compare it to stories you've already completed. If not, it is possible to estimate by analogy to a story you haven't implemented yet. But you need to be careful because you don't yet know if that story was accurately estimated.

One of the worst things you can do when estimating is produce estimates that suffer from systematic errors. A systematic error is one that affects many or all of the estimates you are producing. Estimating by analogy can be particularly prone to this type of bias.

For example, assume Story 1 is estimated to be four days and that Stories 2–10 will be estimated through analogy to Story 1. Story 2 is "a little harder" so it's five days. Story 3 is "about twice as hard" so it will take eight days. Story 4 is a little easier so it's three days, and so on. If the initial four-day estimate for Story 1 is wrong then the other estimates will almost certainly be wrong as well.

Systematic error on the user interface

This team had significant Windows C++ programming experience in client/server environments. On a new project they were tasked with writing the user interface in Java, a language they had recently begun to learn. The lead developer looked into the Java Swing user interface classes and decided that they wouldn't be much more difficult to use than the C++ classes the team was so familiar with.

Unfortunately, the lead developer underestimated the time it would take to correctly place Java user interface components on their screens. This task had been trivial in C++ so he expected the same in Java. He was wrong. The team had estimated that the first few screens would each take two days to develop. When the first two screens took twice that long we knew our estimates were going to be off for all the remaining user interface coding.

This does not mean you should forego estimating by analogy. Rather it means you need to be aware of the possibility of introducing systematic error. The best way to avoid systematic error is early validation of the estimates with the chance to re-estimate.

Fortunately, the agile processes advocate early validation and re-estimation. Within XP, for example, team velocity is used to determine how much work will be done in each iteration. If the team is achieving less (or more) than initially predicted this quickly becomes apparent and you can re-estimate. At that point you will be able to estimate by analogy to stories you have coded so your estimates will be much more reliable.

Estimating by Decomposition

It is sometimes very useful to decompose a story into smaller stories, or to decompose a small story into its constituent tasks. But you need to be careful not to go too far with this approach. The easiest way to illustrate the problem is with a non-software example. Let's use decomposition to estimate my golf score this weekend. Assume the course I am playing has 18 holes each with a par of 4. (If you're unfamiliar with golf scoring, the par score is the number of shots it should take a decent player to shoot his ball into the cup at the end of the hole.)

To estimate by decomposition we need to estimate my score for each hole. I occasionally but rarely birdie a hole (that is, finish it in less than par, or 3 shots in this case). I frequently shoot balls into lakes, parked cars, and so on. It would not be unheard of for me to shoot as high as 10 on a bad hole. So, on our hypothetical course I could shoot as low as 3 and as high as 10 on each hole. That means my range of scores for an 18-hole round could be from 54 to 180. I can assure you I am never going to shoot a 54 and on my worst days never come close to a 180.

The same effect occurs if you decompose a story into too much detail. Contrast this with a more holistic approach in which I might reason that I'll score 7 to 10 on two holes and average a 5 on each remaining hole. That would give me an estimate of perhaps 7 + 10 + (5 * 16) = 97, which is a much more reasonable estimate.

Everything takes four hours

One of my favorite television shows was *Mad About You,* which is about a recently married couple living in New York. In one episode the husband (Paul Reiser) is being pestered by his wife (Helen Hunt) to go shopping for a couch. She insists the trip will take only an hour. He tells her that "Everything in the world takes four hours. You gotta go there, you gotta do whatever, eat, talk about where you should have eaten, and then come home. That's four hours minimum."

This isn't a bad rule of thumb for software development and is a perfect example of how projects get in trouble when they don't distinguish between ideal time and elapsed time. How many of us have been on a project like the following:

Your boss asks you to estimate how long something will take. You tell him it will take "two weeks." He looks at a calendar and puts a big red X fourteen days from then and informs you that is now your deadline. Two weeks elapse and you either meet the deadline by working 25 hours of overtime during the two weeks or you miss the deadline because you didn't.

The problem in this case isn't that your estimate was wrong; rather, it was that when you said "two weeks" what you really meant was "two weeks if I only work on that task and you don't interrupt me with other 'special projects,' no one sends me email, I can skip all status meetings, and I can skip the mandatory all-company meeting scheduled for next Tuesday." You subconsciously gave your estimate in ideal time. But, before you could realize the difference and object your boss turned this into a deadline.

Alternatively, suppose your boss had taken your "two week" estimate and said: "OK, two weeks is 80 hours and you'll probably average 5 hours of time on task each day so that's 16 working days. Let me put this big red X three weeks from now on your calendar."

One further problem with decomposition is that it is a great way to forget some tasks. Suppose I build up my golf estimate by thinking about each hole: There's the first hole and that's pretty easy so let's give me a 3 on that but then I usually hit into the lake on the next hole so that's a 7. Then there's the hole with the sandtraps, let's say a five. And so on. But if I'm mentally recreating an entire golf course it is very likely I'll forget one of the holes. Of course, in this case I have an easy check for that as I know there must be 18 individual estimates. But, when decomposing a story there is no such safety check.

Accurately decomposing a story is hard work and requires you to make reliable guesses about how the design you'll use, the components that may pre-exist and so on. You then come up with estimates based on these guesses and add those estimates up. Unless you invest a lot of time in doing that decomposition you are frequently better off without it.

So, how far should you decompose stories?

During the early planning parts of a project such as Release Planning you should leave stories alone if they are of approximately the right size: anywhere from a day or two up to two or three weeks of work. When mapping out a release strategy over perhaps three to six months you have too many stories to further decompose them with any accuracy.

However, if during Release Planning you are estimating a story and think it is bigger than two or three weeks then you should split the story into two or more stories and estimate those. Splitting an epic into shorter one or two week stories is is very different from decomposing a story into 3 hour tasks at this point.

When you have moved from Release Planning into Iteration Planning, however, you do need to further decompose stories into tasks. See Chapter 12, "Planning an Iteration," for information on decomposing stories into tasks for iteration planning.

Estimating by Wideband Delphi

One of the best techniques for deriving usable estimates is the Wideband Delphi approach.[1] Just like Extreme Programming is an iterative approach to developing software, Wideband Delphi is an iterative approach to developing estimates. Here's how it works:

First, you identify a small group of estimators and tell them which stories they'll be estimating. Each estimator is asked to read the stories and think about them. Next you schedule an estimation meeting that will be attended by all of the estimators.

At the estimation meeting each story is estimated independently. A meeting moderator (typically an XP coach or project manager) starts the meeting by reviewing the estimation assumptions: Estimates are in full days (except tiny tasks that may be estimated at ½ day), are in ideal time for an experienced senior programmer on the project, and are unpadded estimates at the 50% level of confidence. At this point the estimators do not need to worry about the 90% estimates; those will come later.

The moderator reads an initial story card and each estimator is asked to write down her estimate for the story. This is yet another use for 4" x 6" note cards and each estimator is given a handful at the start of the meeting. Estimates are written privately on the cards and when everyone has finished the moderator has the estimators turn over their cards and read the estimate so everyone else can see or hear each estimate.

It is very likely at this point that the estimates will differ significantly. This is actually good news. If estimates differ, the high and low estimators explain their estimates. It's important that this does not come across as attacking those estimators. Rather, you want to learn what it is they were thinking about.

As an example, the high estimator may say, "Well, to test this story we're going to need to create a mock database object and that might take us a day. Also, I'm not sure if our standard compression algorithm will work and we may need to write one that is more memory efficient." The low estimator may respond with, "I was thinking we'd store that information in an XML file—that would be easier than a database for us. Also, I didn't think about having more data—maybe that will be a problem."

1. Boehm, B. *Software Engineering Economics,* 1981. Englewood Cliffs, NJ, Prentice-Hall.

At this point the group discusses it for up to a few minutes. Other estimators will undoubtedly have opinions on whatever reasons the high and low estimators were at the extremes. The moderator should take notes on the story cards of anything that needs future clarification from the customer. However, refrain from running out of the room to ask questions of the customer unless it is just impossible to estimate without an answer.

After the group has discussed the story the moderator asks everyone to re-estimate. The estimators scratch out their old estimates and write new ones on their cards. At the moderator's signal, the cards are again displayed.

In many cases the estimates will already converge by the second round. But, if they have not, repeat the process of having the high and low estimators explain the thinking behind their estimates. In many cases the high and low estimators will not be the same as in the first round. In fact, I have sometimes seen cases where the high and low estimators went to opposite extremes after gaining new knowledge during the discussion.

The moderator should be careful to make sure that all estimators participate in the discussion. I have seen at least a couple of cases where an estimator seemed to be trying to always estimate in the middle so that he would never have to defend or express his position.

The goal is for the estimators to converge on a single estimate that can be used for the story. It rarely takes more than three rounds but continue the process as long as estimates are moving closer together. It isn't necessary that everyone in the room turn over a card with exactly the same estimate written down. If I'm moderating a Wideband Delphi meeting and on the second round four estimators tell me 4, 4, 4, and 3 days I will ask the low estimator if she is OK with a four day estimate. Again, the point is not absolute precision but reasonableness.

Once consensus is reached we have an estimate for the story that the estimators feel can be achieved 50% of the time. Next we turn their attention to creating a 90% estimate for each story. There are two ways to get a good 90% estimate: either repeat the exact same process but remind everyone their estimate is now one they need 90% confidence in or talk about one of the high estimates that came up while estimating the 50% case.

Frequently the latter approach is faster and arrives at the same number. For example, consider the two rounds of estimating shown in Table 9.1. If I were the moderator for this meeting, after the second round I would ask if we could go with a four day estimate with 50% confidence. Depending upon why Jody initially thought the story would take six days I might also ask if we could go with a six day estimate at 90% confidence. If Jody had thought this because he was worried about a few things that the rest of group convinced him were very unlikely I'd probably suggest six days. On the other hand, if Jody initially estimated six days because he'd forgotten we'd already completed some other story that would accelerate this story then I'd probably let the team repeat the process to specifically address the 90% confident estimate.

Anonymity of estimates

When Barry Boehm first wrote about the Wideband Delphi Technique in 1981 he recommended that estimates be given anonymously. Each estimator wrote her estimate on a form that was submitted to the moderator prior to the meeting. The moderator collected this information into a report showing the value of each estimate but no names were associated with the individual estimates. During an estimation meeting the group was asked to discussed the estimates but there was no discussion along the lines of "Jody, you think this will take longer than everyone else does. Why?"

I've never found the need to introduce this level of anonymity into the estimates. I think this is because I ask programmers to estimate in generic Experienced Senior Programmer days rather than in their own days. Additionally, XP's emphasis on courage and communication reinforce that estimates do not need to be made anonymously.

TABLE 9.1 Using Wideband Delphi to estimate a story.

Estimator	Round 1	Round 2
Susan	4	4
Jody	6	5
Ann	3	4
Sherri	4	4

Summary

- There is no one technique that is entirely suitable for estimating stories. To be a successful estimator you must draw upon a variety of techniques including gut feel, decomposition, analogy, and Wideband Delphi.
- Gut feel works best as a check against estimates developed using other techniques.
- Estimation by analogy can be very accurate, especially after some of the stories have been developed.
- Estimation by analogy can be prone to systematic error in which errors in one estimate ripple through many other estimates.
- Decomposition is vital but can be taken too far.
- Wideband Delphi is an iterative approach to estimating that makes use of multiple viewpoints.

Developer Responsibilities

- You are responsible for combining multiple estimation techniques.
- You are responsible for learning how to estimate well. Pay attention to your past estimates and learn from them.

Customer Responsibilities

- ✧ You are responsible for quickly answering any questions developers may have while estimating.
- ✧ You are responsible for remembering that estimates developed early in a project will quite likely be very wrong and will need to be changed as new information is available and as learning occurs.
- ✧ You are responsible for fostering a culture that encourages the the developers to re-estimate as often as necessary.

Questions

9.1 When asked for an estimate, which estimate do you give?

9.2 Does everyone on your team give the same type of estimate?

9.3 Are there other unstated assumptions that underlie the estimates given on your project?

Estimating this book

One of the first things an editor asks you when you start a book is how long it will take to write. My approach was to use a combination of these approaches outlined in this chapter. I had already used decomposition to break the book into chapters and the topics within each chapter. I used my gut feel to come up with initial estimates for how long each chapter would be and how long it would take to write. I then wrote one chapter and looked at each remaining chapter and compared it to the chapter I'd written and adjusted my estimates through the analogy to the initial chapter.

By the way, I finished XXXXXXXXXX schedule.

Chapter 10

Why Plans Go Wrong

In order to put together a useful plan, it is necessary to know why most plans go wrong.

Before looking at how to successfully plan projects with stories it is important to know why so many conventional project plans are so inaccurate as to be useless. There are three main reasons why conventional software plans are so often wrong:

1. Tasks are assumed to be independent
2. Lateness is passed down the schedule; earliness is not
3. The Student Syndrome

Task Independence

Let's start by considering the assumption that tasks are independent. Suppose I ask you to estimate the value of one roll of a die. Since a die has six sides with pip values of one through six your estimate will be (1+2+3+4+5+6) / 6 = 3.5. Of course that estimate will never be exactly right because a die can't land on 3.5; but, it is the best single estimate of the value of one die roll. Instead, what if I ask you to estimate the sum of 5 dice? In that case the best estimate is 5 * 3.5 = 17.5 and the distribution of sums will be as shown in Figure 10.1.

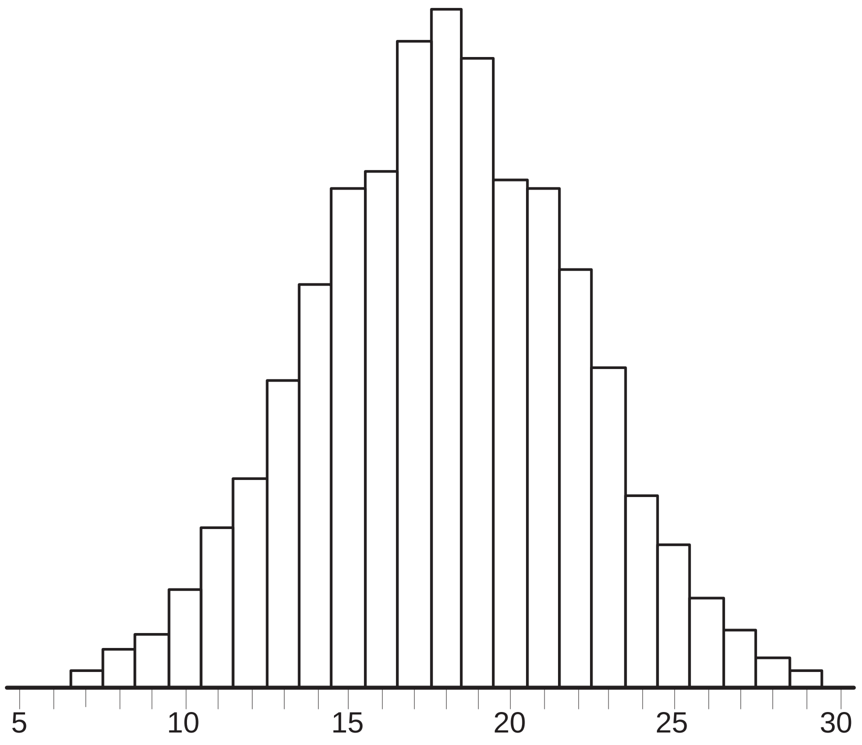


FIGURE 10.1 Distribution of the sum of five dice.

Figure 1 is an illustration of the Central Limit Theorem. The Central Limit Theorem tells us that the sum of a number of independent samples from any distribution is approximately normally distributed. In this case the independent samples are the summed dice rolls.

But does the Central Limit Theorem hold when applied to software?

If we have software development tasks with normally distributed schedule variations this means that some tasks will finish ahead of schedule and about the same number will finish behind schedule. The distribution may not center around zero but the variations will tend to balance out. Unfortunately, many software tasks are not independent of each other. For example, if I'm writing the client portion of an application and the first screen takes 50% longer than scheduled there is a good chance that all of the remaining screens are going to take longer than planned as well. If the tasks of a development effort are not independent then the schedule variations of the individual tasks will not balance out.

Lateness is Passed Down the Schedule

A second problem with conventional project schedules is that they fail to acknowledge that lateness is passed along a schedule but being early is not. For example, look at Figure 10.2, which shows that Story 3 cannot start until both Story 2 and Story 4 are completed. If either of those stories finishes late then Story 3 will start late. The lateness of either story is passed along the schedule. Earliness is passed along, however, only if *both* stories finish early and if the person planning to work on Story 3 is available early.

Student Syndrome

Student Syndrome[1] is yet another reason why most conventional project plans fail. The student syndrome refers to the tendency not to start tasks earlier than necessary. For example, consider the situation where I think a task will take me two days to complete but I don't want to be late so I tell my boss it will take 3 days. Assume my boss even adds a day (because she doesn't know I've already added one). The Gantt chart now shows four days for a task that I really expect will take two days. Because of this expectation I'll probably waste the first two days—I'll surf the web a bit, I'll read those arti-

1. Goldratt, E., *Critical Chain,* 1997. Great Barrington, MA, North River Press.

cles I've been saving. Perhaps I'll make a little progress on the task during the first two days but will only start the task in earnest when I have two days left. If things go well, great. But, if I need the third day that I acknowledged was a possibility I am out of luck and will be late on the task.

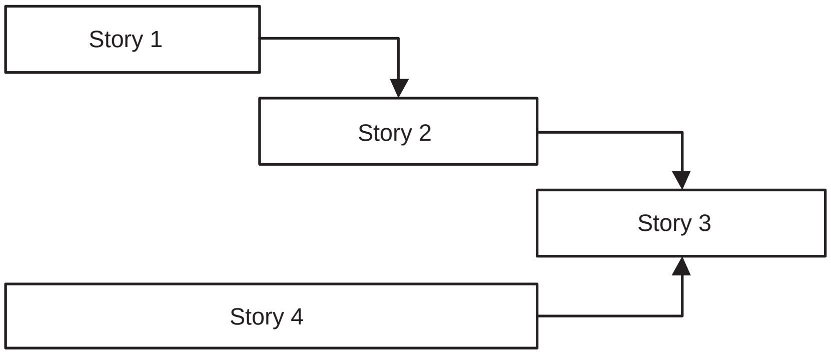


FIGURE 10.2 Story 3 will start late if either Story 2 or Story 4 finishes late.

Local Safety

In the last chapter we learned to estimate at both the 50% and the 90% levels of confidence. An estimate that is only 50% likely to be accurate may not appear very useful because there is no margin of safety built in. However, this is precisely why the estimate is useful.

Developers typically include a margin of safety in their estimates because they've been burned too often before by exceeding their estimates. They've had bosses attempt to hold them accountable for estimates that went "too long" and have learned to include these safety margins. However, to accurately plan a project with stories the safety margin should be applied to a release, not to each story or task.

As an example of why we don't want to include local safety margins, look at Table 10.1, which shows my estimates for a trip to the airport. The 50% estimates represent tasks without any buffers for safety. The 90% estimates include buffers that will accommodate most things that could prolong each task. Because I live 35 miles from the airport I estimate that it takes me 45 minutes to drive there about half the time. However, if there's traffic or I get stopped at a train crossing it takes me 75 minutes. It has even taken me longer a few times.

TABLE 10.1 Estimates of how long it takes me to get to the airport.

Task	50%	90%
find keys	1	5
drive to airport	45	75
park	5	10
check in	7	30
go through security	7	30
Total	**65**	**150**

If I add up all the 90% estimates it appears that I should leave 150 minutes before I want to board the plane. That's an awfully long time, though, for a trip that could take 65 minutes. If all of my estimates come in at the 50% numbers I could be waiting at the airport for 87 minutes. If I sum numbers that each include safety buffers then my overall estimate is too long. I may have a few things go wrong on the way to my flight but I probably won't have them all go wrong.

On the other hand, what is the likelihood of each of the five tasks coming in at its 50% number? We know the likelihood of each is 50% and since the tasks are independent (losing my keys doesn't make the security line take longer) the probability of all tasks coming in at or below the 50% number is $(50\%)^5$= 3.125%. There is only a 3% chance that all of these tasks come in "on schedule." What this tells me is that I should leave somewhere between 63 and 150 minutes before I want to board my flight.

We can't forecast the duration of a project by summing the 50% tasks—that won't be enough time. Similarly, we can't forecast the duration of a project by summing the 90% tasks because that will be too much time. The solution is to construct a plan using the 50%

estimates but add to this a buffer to accommodate the fact that some tasks will be completed faster than their 50% estimates while others will take longer.

Figure 3 shows both a conventional project plan and one that has all local safety moved to a project buffer. In the top Gantt chart the shaded area represents the time difference between feeling 50% and 90% confident in the estimate for the given story. In the bottom Gantt chart of Figure 10.3, the local safety buffers have been removed from the individual stories but have been aggregated into an overall buffer following the last story

A Buffer Isn't Padding

A buffer isn't padding. Padding is extra time added to a schedule that you don't really think you need but that you add just to feel confident in the estimate. Padding is when I take a conventional approach to building a Gantt chart, come up with three months, but tell my boss four months.

A buffer, on the other hand, represents a legitimate expectation of time we expect the project to consume. While I can, if all the stars align, make it to the airport in 63 minutes it would not be prudent for me to plan on that. I should perhaps plan on 90 minutes. If I arrive in 85 minutes I will have consumed 22 of my allocated 27 minute buffer. If I take 95 minutes I will have overconsumed my buffer and will miss my flight. The size of the buffer should be sized based on the duration of the items being buffered and on the consequences of overconsuming the buffer. In the next chapter we will look at how to correctly size the buffer..

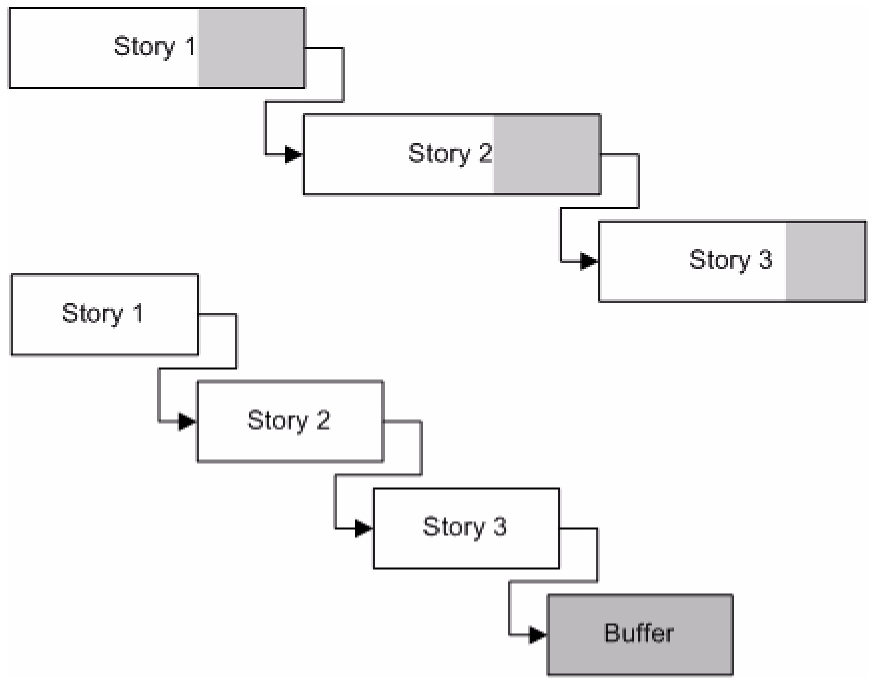


FIGURE 10.3 A conventional plan with local safety margins for each task and the same plan with the local safety replaced by a project buffer

Why Agile Plans Work

We know that conventional approaches to project planning frequently fail because tasks are assumed to be independent but are not, because lateness is passed down the schedule but being early is not, and because of the student syndrome. How do agile processes, and planning with user stories in particular, improve upon this situation?

Just like their plan-driven counterparts, agile approaches assume that the items being estimated are independent. However, a plan-driven approach estimates tasks where in XP we're estimating stories. We've already established that tasks such as "code the search screen," "code the search results screen" and "code the options screen" may seem independent but they are not. The estimate for each of these tasks will be highly dependent upon the user interface programming skills of the programmer coding them. If that programmer overestimates her skills then her estimates will almost certainly be wrong on all tasks.

Estimates of story duration are not prone to these same types of systematic error. Because stories are almost always at a higher level than tasks in a plan-driven process each story will comprise a variety of tasks. For example, a story "allow the user to search for books by title, author or keyword" is broader than the plan-driven task "code the search screen." The story will include that task but will also include additional tasks for the database work. Because XP estimates during release planning are based on stories, rather than tasks, the stories are much more independent than plan-driven tasks.

Of course if a project has ten stories that each include a user interface screen and the team thinks that coding those screens is easier than it turns out to be then the user interface coding portion of each story will have been underestimated. But, since stories (unlike plan-driven tasks) are typically broader than just one task there will be other factors affecting the estimate for each story. This complicated mixture of factors combine to make the estimates independent.

Agile approaches, such as Extreme Programming and Scrum, also have a solution to the problem of lateness but not earliness being passed down a schedule. Where a plan-driven project will have a Gantt showing exactly what each person is to work on each day, XP and Scrum have simply a list of tasks to be done during the iteration. At the start of each day each team member picks a task and works on it. My late completion of the import features yesterday won't affect your start on the export features today. In an agile plan, items are not linked together the way they are in a plan-driven one. Naturally, there are some technical dependencies between tasks—I can't do the "tune the SQL" task until you complete the "write the SQL task." However, these dependencies are far fewer than are found in a plan-driven schedule. When dependencies between stories do exist, agile teams have learned to find ways to create stubs, or temporary implementations, to allow development to proceed without waiting for the completion of a predecessor task or story.

Finally, agile processes overcome the Student Syndrome in a combination of ways. First, agile processes ask the team to commit to achieving the goals and completing the work of each iteration. It's hard to imagine a process that doesn't ask for this commitment from the people who will do the work but processes like XP do more than pay lip service to the request. In return for their commitment, compensatory promises are made to the team. For example, XP promises the team that they'll be able to work at a sustainable pace, will have frequent opportunities for feedback on their creation, and will be able to constantly improve the code they are working on through refactoring.

Second, agile processes increase the visibility of work through practices such as daily standup meetings and pair programming. I cannot waste away the first two days of a two-to-four day task if I have to stand in front of my peers each morning and say what I accomplished the previous day. Similarly, if I can only code when paired with another programmer it will be obvious I'm not coding if no one sees me paired for the first two days.

Finally, the Student Syndrome is avoided because there is no overall project plan showing what should be done on each day. The team works with a do-it-as-fast-as-we-can approach. If there is no schedule telling a developer that she has up to 4 days to do what she thinks will be a two-day task she will not waste the first two days.

Summary

- One of the reasons many conventional plans go wrong is that the tasks on the planned are assumed to be independent.
- Typical software tasks are not independent. If a developer takes longer than planned on one task, there are typically many similar or related tasks that she will also take longer than planned to complete.

- Another reason many conventional plans fail is because lateness is passed along the schedule but being early is not.
- Yet another reason many conventional plans fail is because of student syndrome, which is the tendency to avoid starting a task until the last possible moment.
- Most task estimates include a large margin of local safety.
- Rather than include local safety in each individual task, a plan should collect some portion of the local safety in a single, shared buffer that is used for the entire project.
- Agile plans work because they avoid these problems with more traditional approaches to project planning.

Developer Responsibilities

- You share a responsibility with the customer to ensure that the stories are as independent as possible. Stories should not be so small that systematic error and hidden dependencies are allowed to effect estimates of their duration.
- You are responsible for preventing lateness from being passed down the schedule. You do this by avoiding overly detailed plans that specify artificial dependencies between stories or tasks.
- You are responsible for avoiding the student syndrome by making sure you are always working on the highest priority activities you can.
- You are responsible for eliminating all local safety from your 50% estimates.

Customer Responsibilities

- You share a responsibility with the developers to write stories that are as independent as possible.

- You are responsible for understanding why it is important that estimates be made without local safety buffers. You are also responsible for fostering a project culture where developers know that they are expected to exceed their unbuffered 50% estimates a full one half of the time.
- You are responsible for understanding why a project buffer isn't padding.

Questions

10.1 In what ways could the following stories from an online travel reservation system be interdependent? Could these stories be written differently to make them more independent?

a The user can search for a flight from one city to another on a specified date.

b The user can search for a round-trip flight between two cities.

c The user can restrict a flight search to one or more airlines.

10.2 For which of these events do you think the central limit theorem applies? Under what assumptions do your answers change?

a Pulling successive cards randomly from a deck of cards.

b Tossing a coin 1,000 times and counting the occurrences of heads.

c The number of children in a set of families.

10.3 Besides causing late starts to individual tasks, are there other ways the student syndrome could affect software projects?

Chapter 11

Planning a Release

A high-level release plan shows approximately what will get accomplished during each iteration leading up to a release. There are many variables that go into planning a release. We can use good techniques to estimate each but a release plan can never be a guarantee.

Before we can even begin to plan a release there are two questions that our customer must answer:

- When would you like the release?
- What is the priority of each story

Once we have the answers to these questions we plan the release by estimating how much work the team will be able to accomplish in each iteration. Using this estimate of the team's velocity we can use the 50% and 90% estimates for each story to make a reasonable prediction about how many iterations it will take to produce a release that meet's the customer's expectations.

When Would You Like The Release?

Most customers will start with "yesterday" and only grudgingly move up from there. In most cases your goal is not to get a specific target date from a customer. Instead, you want to know an approximate range of dates: "We'd like it in May but as long as we have it sometime in July that's fine." To plan the release we're going to combine the customer's target date range with a stack of prioritized user story cards. In most cases you'll have too much work to complete everything by the target date so having a range of dates will allow you to offer a broader set of options to your customer.

In some cases the date truly is fixed. Most commonly this occurs when preparing a release for a trade show, a key customer release or some similar milestone. If this is the case, release planning is actually a bit easier as there are fewer variables to consider. However, the decisions about which stories to include will usually be more difficult.

What Would You Like In It?

In order to plan a release the customer must prioritize the stories. One common approach to prioritization, especially common in IEEE 830–style software requirements specifications, is to assign a high, medium, or low priority to each story. There are two problems with this. First, there's no guarantee that the stories will be distributed in any useful manner through the three priority levels. Stories do not need to be split one-third in each priority but a prioritization effort that leaves 90% of the stories as high priority will probably not be helpful. Second, prioritizing stories into three (or even five) levels does not give enough information to identify the stories that will be done in each iteration.

As an example, suppose a project has thirty high priority, thirty medium priority, and thirty low priority stories. For simplicity all stories are estimated to be the same size and the developers believe they can complete thirteen stories in each iteration. Which thirteen of the thirty high priority stories should they choose to do in the first iteration? It’s impossible to know which are the “highest priority” so the developers will need the customer to refine her prioritization.

It is far simpler and better prioritize each story individually so that the seventh story in the pile is more urgent than the eighth but less urgent than the sixth. If the developers are to always focus on the highest priority work, then the customer must assist them by prioritizing that work in sufficient detail.

Prioritizing The Stories

There are many dimensions along which we can sort stories. Among the technical factors we can use are:

- The risk that the story cannot be completed as desired (for example, with desired performance characteristics or with a novel algorithm)
- The impact the story will have on other stories if deferred (we don't want to wait until the last iteration to learn that the application is to be three-tiered and multi-threaded)

Additionally, customers and users have their own set of factors they could use to sort the stories, including the following:

- The desirability of the feature to a broad base of users or customers
- The desirability of the feature to a small number of important users or customers
- The cohesiveness of the story in relation to other stories (for example, a “zoom out” story may not be high priority on its own but may be treated as such because it is complementary to “zoom in,” which is high priority)

Collectively, the developers have a sequence in which they would like to implement the stories as will the customer. When there is a disagreement to the sequence the customer wins. Every time.

However, customers cannot prioritize without some information from the development team. Minimally, a customer needs to know approximately how long each story will take. Before the stories are prioritized they have already been estimated and the estimates written on the story cards, as shown in User Story 11.1. This card shows that the 50% estimate is 3 days and the 90% estimate is 5 days. For simplicity, your customer can think of these as "best case" and "worst case" estimates.

USER STORY 11.1 Provide links back to previously viewed items.

> The site always tells a shopper what the last 3 (?) items she viewed are and provides links back to them. (This works even between session.)
>
> Estimate: 3–5 days

At this point the customer does not sum the estimates and make decisions about what will or won't fit in a release. Instead, she uses the estimates to verify that the likely cost of implementing the story is justified by its value.

A few years ago my team was building a Windows user interface for a customer who was transitioning a large application from an old DOS-based system. In the DOS system the Enter key was used to move forward between fields. The customer wanted us to do the same in her new Windows system. From her customer perspective it was logical that it would take the same amount of development time to use either Enter or Tab. However, we estimated that it would take about an extra person week to use the Enter key. After hearing that, our customer quickly lowered her priority on that story. It was a high priority to her when she thought it was a few hours; when it was a week there were many other things she decided she would rather have.

Mixed Priorities

If a customer is having trouble prioritizing a story, the story may need to be split. The story doesn't need to be split because it is an epic but because it is really multiple stories that can be better prioritized when split.

On one project I had the story shown in User Story 11.2. The customer struggled to prioritize the story because searching by author and title were essential while the other search fields were considered nice but were not essential. The story was split into three. One story for searching by author or title, another for searching by publication name or date, and a third allowing for the criteria to be combined.

USER STORY 11.2 Search criteria.

> *Users can search for magazine articles by author, publication name, title, date, or any combination of these.*

Risky Stories

Looking back over earlier approaches to software development it is clear that there has been an ongoing debate about whether a project should first go after the riskiest parts or the most valuable parts of the project. Probably the leading proponent of risk-driven development has been Barry Boehm whose spiral model[1] focuses on the early elimination of risk. On the other end has been Tom Gilb[2] who advocates doing the "juicy bits" first.

Agile approaches are firmly in the camp of doing the juicy bits first; but, that does not mean risk can't factor into the prioritization of the stories. Many developers have a tendency to want to do the riskiest stories first. Sometimes this is appropriate but the decision must still be made by the customer. However, the customer considers input from the technical team when prioritizing the stories.

On one recent project in the biotech space some of the stories called for novel extensions to a standard statistical approach called expectation maximization. Because the work being undertaken was truly new, the team could not be sure if it could be accomplished at all or how long it would take. The product would still have been saleable without the inclusion of these stories so the customer prioritized them to be near the middle of the stack. However, once the customer was made aware of the extremely high risk associated with these stories, enough of them were given higher priorities in order to determine what was involved in developing the novel algorithms.

Prioritizing Infrastructural Needs

Frequently the risky stories are associated with infrastructural or non-functional needs such as performance. I was on a project to develop a web program that would display charts of stock prices. One of our stories is shown in User Story 11.3. For the baseline web server machinery that had been specified, this level of performance could be a significant challenge. The difficulty of meeting this performance requirement would have a profound impact on our architectural decisions.

1. Boehm, B., "A Spiral Model of Development and Enhancement," *IEEE Computer,* May 1988, 61-72.

2. Gilb, T., *Principles of Software Engineering Management,* 1988. Reading, MA, Addison-Wesley.

USER STORY 11.3 Generate 50 images per second.

Be able to generate 50 stock chart images per second.

We'd already committed to Java as our server-side language but could we achieve 50 images per second with Java? Would we instead need to go with native C or C++ code for the image generation? Or could we achieve our throughput goal with a strong caching algorithm that would serve up the same chart for requests that were only seconds apart?

In this case the customer had written User Story 11.3 for us. However, he prioritized it fairly low. Our first few iterations would be targeted at developing features that could be shown to prospects and used to generate initial sales and interest in the product. Our customer reasoned that we could always add scalability in later. In some cases it is easy to refactor a system to improve its scalability. In other cases, that type of refactoring can be very difficult. It is up to the developers to help the customer by identifying stories that can be deferred but may become much more costly to develop if implemented later. Developers must not, however, take this as a license to steer the customer toward early implementation of their favorite technical features.

On another project, the customer clearly wanted the application to be deployable as a three–tiered application with a database server, a client machine, and a middle-tier that would route requests and data between them. The customer had talked with the team about this in various meetings and the marketing literature she was preparing described the system as three-tiered. However, none of the stories was written so that it required the addition of the middle tier.

This was becoming troubling to the technical team. They did not mind starting with a simple two-tiered system (database server and client machine) but after a couple of iterations had gone by they became more and more concerned that the middle tier had not been added. They knew that adding the middle tier would still be easy but that it would get a little harder with each iteration. Also, because the user stories were written with their focus entirely on end-user functionality it was not clear when such an infrastructural need would be added.

The solution was to write a story that made the three-tier capability a higher priority to the customer who was prioritizing the team's work. In this case we added the story: “During installation the user can decide to install everything locally on her PC or to install the client, middle-tier and server separately.”

From Ideal Days to Calendar Days

Suppose that our customer has prioritized all of the story cards. We add up the 50% estimates on them and get 100 ideal, experienced senior programmer days. We add up the 90% estimates and get 180 ideal, experienced senior programmer days. Using ideal days made it easier for estimating the stories but now we need a way to convert ideal days into elapsed calendar days.

The answer, of course, is to use velocity. Velocity represents the amount of work that gets done in an iteration. Once we know a team’s velocity we can use it to turn ideal days into calendar days. For example, if we estimate our project at 100 ideal days then with a velocity of 25 we can estimate that it will take 100 / 25 = 4 iterations to complete the project.

The Initial Velocity

There are three ways to get an initial value for velocity:

1. Use historical values.

2. Run an initial iteration and use the velocity of that iteration.
3. Forecast velocity based on the same principles we used to estimate the stories.

Using historical values is the best option but it is only viable if we have an existing team that is rolling off a project similar to the new project and if no one is joining or leaving the team. Unfortunately, it is rare that the exact same team gets to work on two consecutive similar projects.

Running an initial iteration is a great way to get a starting velocity. However, there are many times when this is not viable. For example, suppose your boss comes to you with a new product idea. She's written the user stories she thinks are needed in the first version. She's used those stories to do market research and thinks the product will earn $500,000 the first year. If the product can be developed cheaply enough the company will pursue it. If not, they'll pass. When your boss asks you what it will cost to develop you are not always given the freedom to say, "Let me run a sample iteration for two weeks and get back to you." In cases like this you need a way to forecast velocity.

Forecasting Velocity

If you don't have historical velocity measures, the best approach is to build a bottom-up estimate of the team velocity. As you'll recall from Chapter 9, "Estimating User Stories," estimates of ideal time are cast in the context of experienced senior programmer days where the entire day is spent programming (as opposed to answering email, attending meetings, and so on). We can use this context to determine how each developer on the team compares to the archetypal experienced senior programmer we used while estimating the stories. We also take into consideration each programmer's availability and the amount of time we expect him to dedicate to planned work each day. The result of doing this for a hypothetical project and team is shown in Table 11.1.

TABLE 11.1 Building a bottom-up estimate of velocity.

Developer	Iteration 1	Iteration 2	Iteration 3	Thereafter
Susan	.5	.6	.7	.7
Ann	.5	.5	.5	.5
Randy	.2	.3	.4	.4
Clark		.2	.3	.4
Vlade	.5	.6	.7	.7
Chris	.8	.9	1.0	1.0
Total	**2.5**	**3.1**	**3.6**	**3.7**

Table 11.1 shows a team of six programmers that takes three iterations to come up to speed and then each hit a constant rate of productivity. For example, in the first iteration Susan is expected to contribute at a rate equal to one-half of what we forecast for the archetypal experienced senior programmer we imagined when estimating. Does this mean Susan is a bad or inefficient programmer? Not at all. Rather, it means Susan will spend time answering email, participating in team discussions, meeting with the project customer, writing tests, and performing other activities that we are not directly estimating. In fact, Susan's profile (which is identical to Vlade's) is what I consider a very standard one for a good, solid programmer who is a close match to our archetypal experienced senior programmer.

Susan is expected to become more productive in the second and third iterations. This is typical because it frequently takes a couple of iterations for the team to become accustomed to each other, there are sometimes lingering deployment issues from whatever project was completed prior to this one, there may be new development tools and technologies to learn, or the project support infrastructure such as build servers and version control systems needs to be set up during the initial iterations. Table 1 shows Susan's productivity stabilizing from the third iteration forward.

Continuing to look at Table 1, Ann has other minor commitments to other projects or teams so we do not anticipate any increases in her productivity. Randy's numbers are low but he's just a less experienced developer; we don't expect him to contribute as much as an experienced senior programmer. Clark is similar to Randy but he's on vacation for two weeks and will not be available during the first iteration.

Chris actually progresses all the way up to a 1.0 by the third iteration. This indicates that from the third iteration on Chris will be contributing exactly the amount we expected from an experienced senior programmer on an ideal day. What this means is that Chris is actually far more productive than our archetype because he is able to fit a full ideal day into his email- and meeting-reduced calendar day.

If we sum the individual velocity values we get an expected team velocity, which is shown in the Total column at the bottom of Table 1. In this case, the team will have an initial velocity of 2.5 ideal days per calendar day. This will increase to a sustained velocity of 3.7 ideal days per calendar day. Another way of thinking about this is that the team's initial velocity will be 2.5 ideal days for every 6 person-days of work.

Handling Additional Uncertainty

Sometimes it is necessary to predict velocity and a release plan without knowing who will be on the team. This can happen when you need to figure out how many developers it will take to complete a project by a given date or it can happen when you are asked to plan a project that will happen sometime in the more distant future. There are obvious additional risks to this; but, in many companies this type of planning is a fact of life.

When you build a release plan for these cases you can still estimate velocity in the same bottom-up manner. However, since you cannot use known programmers you must make assumptions about how the project will be staffed. Table 11.2 shows a bottom-up estimate of velocity for a team consisting of two experienced programmers and two more junior ones.

Vacation and Sick Time

You need to consider vacation and sick time when creating a release plan. If the project is short you can ask the team members to identify their vacation plans and you can then adjust individual contributions to velocity as was done for Clark in Table 1 during the first iteration.

If the project is long enough that you cannot reasonably identify vacation plans in advance then you can reduce the planned velocity slightly to accommodate the vacations you expect will occur. The easiest way to do this is to assume that vacation is taken evenly throughout the year. For example, if your company gives each employee twelve vacation days per year then you can assume each employee will take one day per month. You can handle sick time this same way since specific sick days cannot be predicted in advance.

To see how this works, assume our company gives us 15 days of vacation per year. We are also given up to ten days of sick time but each person uses an average of five days. That means we can count on each person being gone about 20 days per year for either vacation or illness. Most employees work about a 250 day year. Since 20 is 8% of 250 velocity estimates should be reduced by about 8% to accommodate the impact of vacation and sick time.

In the case of the example in Table 1 you would want to reduce the sustained velocity from 3.7 to 3.4 unless you had specifically accounted for vacation and sick time in those calculations.

TABLE 11.2 A bottom-up velocity estimate for an unknown team.

Developer	Iteration 1	Iteration 2	Iteration 3	Thereafter
Programmer 1	.5	.6	.7	.7
Programmer 2	.5	.6	.7	.7
Programmer 3	.3	.4	.5	.5
Programmer 4	.3	.4	.5	.5
Total	**1.6**	**2.0**	**2.4**	**2.4**

Creating a Simple Buffered Plan

Regardless of whether velocity is determined from historical values, an initial iteration or forecast, it can be used to estimate the duration of the project. We learned in Chapter XX, Why Plans Go Wrong, that we cannot simply divide either the 50% or 90% estimates by the team velocity to come up with a reasonable project duration. If we do that we'll have a plan that is too long (if we used the 90% estimates) or too short (if we used the 50% estimates). As you'll recall, the solution was to use the 50% estimates but to add a buffer to the schedule.

The idea of creating a schedule in this way comes from Eli Goldratt who uses the term "Critical Chain."[3] As a simple approach, Goldratt suggests that the buffer be sized as one half the duration of the tasks being buffered.

As an example, assume our 50% estimates are summed to be 100 ideal days. If we follow Goldratt's suggestion we add 50% and come up with an estimate of 150 ideal days.

To convert ideal days into something meaningful on the calendar we use the team velocity we've chosen either through historical observation, an initial iteration, or by forecasting. Let's use the forecasted velocity shown in Table 11.1 and assume that we are doing two-week (ten working day) iterations. This means we'll complete 25 ideal days during the first iteration, 31 ideal days during the second, and so on as shown in Table 11.3.

TABLE 11.3 Calculation of cumulative ideal days by iteration.

Iteration	Duration (Days)	Velocity	Ideal Days in Iteration	Cumulative Ideal Days
Iteration 1	10	2.5	25	25
Iteration 2	10	3.1	31	56

3. Goldratt, E., *Critical Chain,* 1997. North River Press Publishing Corporation, Great Barrington, MA.

TABLE 11.3 Calculation of cumulative ideal days by iteration.

Iteration	Duration (Days)	Velocity	Ideal Days in Iteration	Cumulative Ideal Days
Iteration 3	10	3.6	36	92
Iteration 4	10	3.7	37	129
Iteration 5	10	3.7	37	166

The project will be complete when we've accumulated 150 ideal days. According to Table 11.3 this will be in the middle of the fifth iteration. Based on this we tell our customer it will take five two-week iterations to develop all the functionality she has asked for.

More Sophisticated Buffering

Goldratt's rule of thumb of using a 50% buffer is a reasonable starting point for sizing the project buffer. However, we can do better by using our 90% estimates. Following the method suggested by Leach[4] we can assume that the difference between the individual 50% and 90% estimates is some multiple of the standard deviation of each task's completion time. By taking the square root of the sum of the squared differences we can calculate a buffer size that will protect the overall project to the same 90% level the individual estimates had been protected.

For an example of how to calculate the buffer in this way see Table 11.4. This table shows a short project that includes seven stories with their 50% and 90% estimates as shown. Summing the 50% and 90% estimates indicates that the project will require between 20 and 33 ideal days. The sum of the squared differences between the 90% and 50% estimates is 31. The square root of 31 is 5.57, indicating that we need a 6 ideal day buffer to the project. This means that

4. Leach, L. P., *Critical Chain Project Management,* 2000. Norwood, MA, Artech House.

the overall project estimate is 26 ideal days, which is four days shorter than the result of using a simple 50% buffer.

TABLE 11.4 A more sophisticated calculation of buffer size.

Story	50% Estimate	90% Estimate	$(90\%–50\%)^2$
Story 1	2	5	9
Story 2	3	5	4
Story 3	1	1	0
Story 4	1	3	4
Story 5	3	5	4
Story 6	5	8	9
Story 7	5	6	1
Total	**20**	**33**	**31**

You should calculate the buffer size using both methods and use your intuition in selecting a final buffer size. The buffer size determined using the 90% estimates will not always be shorter than the simple buffer. In cases where the 90% estimates represent a great deal of uncertainty the more sophisticated calculation will usually lead to a larger buffer size.

Handling Specialists

How do we account for specialists (such as testers or database administrators) who may be on the project? Take a look back at Table 11.2 and notice that each of the lines is for a programmer. It is generally too difficult to directly predict the velocity of a tester. Rather, the impact of testers is best accommodated by adjusting the programmer velocity numbers. If the team will not have any testers then use lower values than you would if the team included testers. Also, keep in mind that estimates of initial velocity are guesses and they will be replaced with measured actuals as soon as they become available after a few iterations.

Other specialists can be handled the same way as long as they are not the limiting factor in the productivity of the overall team. If a specialist resource, such as a database administrator, is central to so many stories on the project that she becomes the critical path then you must plan the release around the specialist as the project constraint.

If you find your project in this situation: First, you should do whatever you can to correct your dependence on a bottlenecked specialist. For example, while you may need to rely on a highly-skilled database administrator for some tasks there may be other database-related tasks that can be picked up by database-savvy non-specialist developers. Second, you need to plan your release around the work that can be accomplished by the constraint specialist. This means that in addition to estimating how long each story will take the programmers to complete you need to estimate how long each will take the specialist. You can then prepare two release plans using calculations similar to those shown in Table 11.3.

A Warning

Be careful not to put too much faith in your estimates. The techniques described in this chapter will help you estimate the approximate duration of a project and allow you to make statements like "The product will be ready for release in approximately 7–9 iterations." They don't give you enough precision, however, to say things like "We'll be done on June 3."

The estimates of story durations and planned velocity contain a great deal of subjective information. This makes them useful for high-level release planning but not for anything more precise. In the aggregate, a plan derived using these techniques will be sufficiently accurate as to be useful. But remember that each individual estimate has a 50% likelihood of being too short.

Use the information you gain in this way to set initial expectations but then constantly reset those expectations as you gain new information. Monitor each iteration's velocity and re-estimate the stories whenever you learn something new that affects the estimates.

Summary

- Before planning a release it is necessary to know approximately when the customer would like the release and the relative priorities of the stories.
- Stories should be prioritized into a specific order (first, second third, and so on) rather than into groups (very high, high, medium, and so on).
- Stories are prioritized by the customer but with input from the developers.
- Estimates, which are in ideal days, are converted into calendar time using velocity.
- It is often necessary to estimate a team's initial velocity.
- The duration of a set of stories can be estimated using the 50% estimates and adding a buffer that represents some portion of the difference between the 50% and 90% estimates for each story.

Developer Responsibilities

- You are responsible for providing information to the customer in order to help her prioritize the stories.
- You are responsible for resisting the urge to prioritize infrastructural or architectural needs higher than they should be.
- You are responsible for creating a release plan that is built on realistic estimates yet includes an appropriately sized project buffer.

Customer Responsibilities

- ✧ You are responsible for prioritizing the user stories into the precise order you value them. It is not sufficient to sort them into stacks of high, medium, and low priority.
- ✧ You are responsible for expressing your honest deadlines for the release. If you need it on July 15th, don't tell the developers you need it on June 15th just to be safe.
- ✧ You are responsible for understanding the difference between ideal time and calendar time.
- ✧ You are responsible for splitting stories that contain components that you want prioritized differently.
- ✧ You are responsible for understanding why a programmer with a personal velocity of 0.6 should not be reprimanded or criticized because her velocity is less than 1.0.

Questions

11.1 How large should the buffer be for a project containing the following stories?

Story	**50% Estimate**	**90% Estimate**
Story 1	4	7
Story 2	3	5
Story 3	2	3
Story 4	3	5
Story 5	2	5
Story 6	3	4
Story 7	5	8
Story 8	4	6
Story 9	3	4
Total	**29**	**47**

11.2 Assuming one week (five working day) iterations, how many iterations will it take for the following team to complete the project?

Developer	Iteration 1	Iteration 2	Iteration 3	Thereafter
Programmer 1	.5	.6	.7	.7
Programmer 2	.5	.6	.7	.7
Programmer 3	.3	.4	.5	.5
Total	**1.3**	**1.6**	**1.9**	**1.9**

Chapter 12

Planning an Iteration

Iteration planning adds detail to the release plan but does it an iteration at a time. By looking into the future only when it's close enough to be somewhat predictable we are able to make plans that are useful.

Release planning has left us with a coarse-grained allocation of stories to the iterations that comprise a release. This level of planning—not so much detail that we give the false feeling of precision and accuracy but enough planning that we can base actions on it—is ideal for planning a release. However, at the start of each iteration it is important to take the planning process one step further.

Iteration Planning Overview

To plan an iteration the team holds an iteration planning meeting. All of the developers (that is, programmers, testers and others) on the team attend and participate in this meeting. Ideally the customer participates; but, depending on how familiar the team is with the product, the domain, and the stories in the iteration, it is possible (albeit difficult) to plan the iteration without the customer in the room.

Because the team will be looking at the stories in detail they will undoubtedly have some questions about them. If the customer isn't present to answer those questions then the team will need to ask the questions after the meeting. Depending on the answers they get they may need to alter the results of their iteration planning.

Determining whether the customer should be present during the iteration planning meeting is largely a cultural issue. If the culture is such that the team had a hard time getting a dedicated, designated customer then they may be better off with some inefficiency during the iteration planning meeting but allowing the customer to go about her normal work. So, instead of being tied up in a four-hour meeting the customer will perhaps instead get asked two hours of questions separately. Less efficient overall because the team will have to adjust their plans based on her answers, but perhaps more appropriate in some environments.

The general sequence of activities for an iteration planning meeting is as follows:

- discuss a story
- decompose the story into its constituent tasks
- one developer accepts responsibility for each task
- after all stories have been discussed and all tasks have been accepted, developers individually estimate the tasks they've accepted to make sure they are not over-committed

Each of the activities is discussed in one of the following sections.

Discussing the Stories

As input to the iteration planning meeting the team has the release plan that was established at the start of the project, and has perhaps been updated since. The release plan lists the stories that are planned for development during the iteration. Of course, everyone on the project knows more now than they did when the release plan was written. This means that the release plan with its list of stories for this iteration is taken as a guide, not a mandate, for what the team will now plan for the iteration. If, over the iterations thus far, team velocity has been better than initially forecast then the team will likely choose to pull in more story points than initially planned. Similarly, perhaps the team has learned that some of the stories planned for this iteration will be much more difficult (or much easier) than initially planned.

An interesting approach

Joshua Kerievsky of Industrial Logic takes an interesting and novel approach to prioritization during iteration planning. He projects a list of candidate stories onto a whiteboard. The customer can then circle and annotate the stories she wants worked on during the iteration.

After the meeting the customer's markings and notes are transcribed onto the image that was projected. Kerievsky reports that the image becomes useful later as a very simple but effective way to review how velocity changed from iteration to iteration.

An even easier way to do all this would be to use one of the older-style overhead projectors and a transparency listing the stories. The customer could write directly on the transparency and the transparencies saved.

Just as programmers may change their opinions about the difficulty of programming a story, the customer may change her mind about the priority of a story. The iteration planning meeting is the perfect time for the customer to express these priority changes to the team. To start the meeting, the customer starts with her highest priority story and reads it to the developers. The developers then ask questions until they understand the story sufficiently to decompose it into constituent tasks. It is not necessary to understand every detail of the story, and diving too deeply into the details of each story can make the meeting very lengthy and inefficient, since not everyone in the meeting needs to hear all the detail on all stories. The developers will still be able to work out the fine details of the stories with the customer after the planning meeting.

Changing Priorities

It is usually best if the customer can withstand the desire to change priorities *during an iteration*. It is very easy for a team to get whipsawed during an iteration if the customer changes her mind frequently. On one project, for example, the customer and a programmer met and agreed about how a database search feature would work. Five days into a ten-day iteration (and about two-thirds through coding the search feature), the customer came up with what she thought was a better solution that was completely different from the original, partially coded solution. At that point, in her mind the customer was comparing two uncoded solutions and she naturally favored the one she thought was better. She urged the team to abandon the current approach and immediately start on the new approach. We politely asked her to wait until the end of the iteration and she agreed. At that point, she was comparing a fully working solution that did most of what she wanted in a search feature and another version that was undoubtedly better but that would take 10 days to develop.

Even though she (and the rest of the team) thought the new search feature would be better it was not worth adding at that point when compared to a fully working adequate feature. Users were better served by having developers work on entirely new features.

Decomposing Into Tasks

There is really no art to decomposing a story into tasks. Many developers have been doing this for much of their careers. Since stories are already fairly small (typically taking the project's archetypal programmer one to five days of ideal time) there is usually not that much decomposition necessary.

In fact, why decompose at all? Why not leave the story alone as a discrete unit of work?

Even though stories are small enough to serve as units of work, projects are generally well served by decomposing them into even smaller tasks. First, for many teams the story will not be implemented by just one developer (or one pair of developers). The story may be split among developers either because the developers specialize in certain technologies or because splitting the work is the fastest way to complete the story.

Second, stories are descriptions of user– or customer–valued functionality; they are not to-do lists for developers. The act of converting a story into its constituent tasks is often useful because it helps point out tasks that might have been forgotten. Because the decomposition into tasks happens in a group setting the entire strength of the team is brought to the effort. While one developer may forget that it's necessary to update the install program as part of a story it is less likely that everyone will forget.

As various team members call out the tasks that comprise a story someone on the team needs to write the tasks on something. My personal preference is to write them on a white board in a shared team meeting room.

As an example of decomposing a story into tasks, suppose we have the story "Users can search for a hotel on various fields." That story might be turned into the following tasks:

- Code basic search screen
- Code advanced search screen
- Code results screen
- Write and tune SQL to query the database for basic searches
- Write and tune SQL to query the database for advanced searches
- Document new functionality in help system and user's guide

In particular notice the inclusion of the task for updating the user's guide and help system. Even though this story did not explicitly say anything about documentation the team knew from prior iterations that there are a help system and a user's guide and that they need to be accurate at the end of each iteration. If there was any question about this the team could have asked the customer.

Guidelines

Because stories are already fairly small it is not necessary to set very precise guidelines around the desired size of a task. Use these guidelines when decomposing stories into tasks:

- If one task of a story is particularly difficult to estimate (for example, a list of supported data formats requires approval from a remote Vice President who is slow to respond) separate that task from the rest of the story.
- If tasks could easily be done by separate developers, then split those tasks. For example, in the case above, coding of the basic and advanced search screens was separated. There may be some natural synergy to having the same programmer or pair work on both but it isn't necessary. Decomposing tasks in this way is useful because it lets multiple developers work on the same story. This is frequently necessary near the end of an iteration as time starts to run out. Similarly, the task "Code basic search screen" could have been split into two tasks: "Design screen layout for basic search screen" and "Code basic search screen" if the team is using a user interface designer or group.
- If there is benefit in knowing that a part of the story is done, break that part out as a task. In the example above, the coding of the basic and advanced search screens were made separate tasks. This will allow the developer who writes the database access code to connect her SQL to the search screens one at a time as they become available. This means that a delay in the

completion of the advanced search screen does not delay completion of the two tasks for the basic search screen.

Accepting Responsibility

Once all the tasks for a story have been identified someone on the team needs to volunteer to perform each task. Even if the team will be doing pair programming it is generally best to associate a single name with each task. This person assumes responsibility for completing the task. If he needs additional information from the customer, he gets it. If he chooses to pair program, he solicits a pair. Ultimately, though, it is his responsibility to make sure the task gets completed during the iteration.

Actually, it's the responsibility of everyone on the team to make sure the task gets completed. The team needs a "we're all in this together" mentality. And, if near the end of the iteration, one developer is not going to complete all the tasks he accepted then others on the team are expected to take on that work to the extent possible. As we learned in Chapter 10, "Planning an Iteration," one of the reasons agile planning succeeds is because there is no overall Gantt or PERT chart showing unnecessary task sequences. When work needs to be done, anyone available does the work.

If you wrote the tasks on a white board, simply write the name of the developer responsible next to each story.

Estimate and Confirm

If a project team's velocity is forty story points per iteration then the previous steps—discussing a story, decomposing it into tasks, and accepting responsibility for the tasks—are repeated until the team has discussed the customer's top forty story points worth of stories. At that point each developer is responsible for estimating the amount of work she has accepted responsibility for. The best way to do this is still to estimate in ideal time but to estimate in your own ideal time, rather than in the ideal time of the project's archetypal program as was done in Chapter 8, "Principles of Estimating."

By this point the tasks should be small enough that they can be estimated with some reliability. But, if not, don't worry about it. Take a best guess at the duration of the task and move on. Estimate at the 50% level of confidence. If, as was recommended earlier in this chapter, you wrote all the tasks and the names of the responsible developers on a white board, each developer can now add his or her estimates to the board. The result will look something like Table 12.1.

TABLE 12.1 It's easy to track tasks, the developer doing each task, and estimates on a white board.

Task	Who	Estimate
Code basic search screen	Susan	6
Code advanced search screen	Susan	8
Code results screen	Jay	6
Write and tune SQL to query the database for basic searches	Susan	4
Write and tune SQL to query the database for advanced searches	Susan	8
Document new functionality in help system and user's guide	Shannon	2

Once a developer has estimated all of her tasks, she needs to add them up and make a realistic assessment about whether they can all be completed during the iteration. For example, suppose a new two-week (80 hour) iteration is beginning and I have accepted tasks that I now estimate will take 53 hours of actual time on task. It's doubtful that with everything else I have to do that I will be able to put that much time directly onto these tasks. Also, the 53 hours are my 50% estimates so they could definitely take longer than 53 hours. At this point I have the following options:

- Keep all the tasks and hope.

- Request that someone else on the team take some of my tasks.
- Talk with the Product Owner about dropping a story (or splitting a story and dropping part of it).

Unless one or more of the stories brought into the iteration was larger than the number of story points it was estimated, it is very likely that another developer has capacity to take on some of the work I signed up for. On the other hand, everyone else may be equally over-allocated because of the common tendency to estimate longer durations for ourselves than for a faceless archetypal programmer. This is local safety creeping back into our estimates.

My recommendation is that you start the iteration with all of the planned work. Assuming this is not the team's first iteration, you have *evidence* that the team produces working software with a certain velocity. You have pulled an amount of work into this iteration based on that velocity. Now you've got a *hunch* that you may be wrong because when you add up the tasks they don't all fit. However, when forced to choose between the evidence of prior velocities and a hunch, I'll side with the evidence. After you get partway through the iteration you can better assess whether you've brought in too much work. In Chapter 13, "Measuring and Monitoring Velocity," we will look at iteration burndown charts, which are a useful tool for assessing a team's progress through the work in an iteration.

You should consider asking the customer to remove a story only when your velocity is suspect. That is, during the first iteration when you've only taken a guess at the velocity or during a later iteration when you have a strong belief that one or more of the stories being included should have been estimated as more story points.

Summary

- Iteration planning takes release planning one step further but only for the iteration being started.

- To plan the iteration the team discusses each story and decomposes it into its constituent tasks.
- There is no mandatory size range for tasks (for example, three to five hours). Instead, stories are decomposed into tasks to facilitate estimation or to encourage more than one developer to work on various parts of the story.
- One developer accepts responsibility for each task.
- Developers assess whether they have over-committed themselves by estimating each task they have accepted.

Developer Responsibilities

- You are responsible for participating in the iteration planning meeting.
- You are responsible for helping to decompose all stories into tasks, not just the stories you are likely to work on.
- You are responsible for accepting responsibility for the tasks you will work on.
- You are responsible for ensuring you take on an appropriate amount of work.
- Throughout an iteration you are responsible for monitoring the amount of work you have left as well as the amount of work your teammates have left. If you're likely to finish your work, you are responsible for taking on some work from your teammates.

Customer Responsibilities

- You are responsible for prioritizing the stories that will be included in the iteration.
- You are responsible for directing the developers toward the greatest business value they can deliver. This means that if

higher-value stories have come to light since the release plan was first established you are responsible for adjusting priorities to deliver maximum business value.

- ✧ You are responsible for participating in the iteration planning meeting, or at least understanding the impact if you do not.

Questions

12.1 Decompose the following stories into their constituent tasks:

a A user can pay with a credit card.

b Users can view detailed information about a hotel.

Part 3

Stories In The Lifecycle

Chapter 13

Measuring and Monitoring Velocity

Velocity is the primary indicator of how long a project will take. It is worth measuring and monitoring over time.

As you recall from Chapter 11, "Planning a Release," the release plan was created by breaking the project into a series of iterations, where each iteration included a certain number of story points. The number of story points completed in an iteration is known as the project's velocity. When planning the project we either used a known velocity (if we had one, perhaps from another similar project) or we made one up. Velocity can be a useful management tool so it is important to look at the team's velocity at the end of each iteration as well as during the iterations.

Measuring Velocity

Because velocity can be such an important measure it is important to think about how we'll measure it. Most stories are easy to count: the team completed them during the iteration so they are counted at full value. Assume, for example, that a team completed the stories shown in Table 13.1 during an iteration. As can be seen in this table, the team's velocity is 23, which is the sum of the story points for the stories completed in the iteration. If the project plan assumed a velocity that is significantly different from 23 it may be necessary to reconsider the project plan. However, be careful about adjusting a release plan too early. Not only is an initial velocity notoriously difficult to forecast it can also be very volatile during early iterations. You may want to wait two or three iterations until you have a longer-term view of velocity.

TABLE 13.1 Stories completed during an iteration.

Story	**Story Points**
Story 1	4
Story 2	3
Story 3	5
Story 4	3
Story 5	2
Story 6	4
Story 7	2
Velocity	**23**

However, what about stories the team only partially completed? Should they be included in velocity calculations?

No, you should not include partially complete stories when calculating velocity. There are a number of reasons why. First, there's the natural difficulty of figuring out what percentage of a story is complete. Second, we don't want to imply a false precision to velocity by reporting it with fractional values like 43.8. Third, incomplete stories do not typically represent anything of value to users or customers. So, even though they may have been partially coded, such stories are often left out of formal end-of-iteration builds if the software is to be delivered to any users. Fourth, if stories are so big that including a partial story routinely affects velocity, say from 41 to 44, then the stories are too large. Finally, we desperately want to avoid a situation where many stories are 90% done, yet few are 100% done. A great deal of complexity can lurk in that last 10% so it is important to finish each story completely before counting it.

If you find yourself tempted to include partially complete stories, evaluate the length of your average story and consider striving for shorter stories. At the end of an iteration when determining velocity, it is far easier to forego half of a one-point story than it is to ignore a twelve-point story. Additionally, if you frequently find iterations finishing with many partially complete stories (even if they are all half-point stories) this may be a symptom of a lack of teamwork on the team. With an all–for–one approach the team will learn they are better off joining together to complete some stories, leaving others unstarted than they are partially completing all of them.

Velocity Does Not Use Actual Hours

Notice that velocity calculations are made using the story point values assigned before the start of the iteration. Do not adjust story point estimates in an attempt to more accurately calculate velocity. For example, suppose a story you estimated to be four story points turned out to be much larger. In fact, knowing what they know now the team would estimate it as eight story points. This story contrbutes four points to the velocity calculation, not eight.

Upcoming iterations undoubtedly have some stories that will have similar errors; but, you probably can't tell what stories they are yet. So, by not adjusting the story point value of the completed story, the calculated velocity will accommodate similar errors that will be found in later iterations.

A good way to monitor whether actual velocity is deviating from planned velocity—and, more importantly, whether it's something you need to act on—is to graph planned and actual velocity for each iteration. This can be seen in Figure 13.1, which shows planned velocity starting low but then increasing and stabilizing by the third iteration. Actual velocity, graphed through the third iteration, exceeded planned velocity for the first iteration. However, the actual improvements during the second and third iterations were not as great as planned and so actual velocity is slightly less than planned velocity.

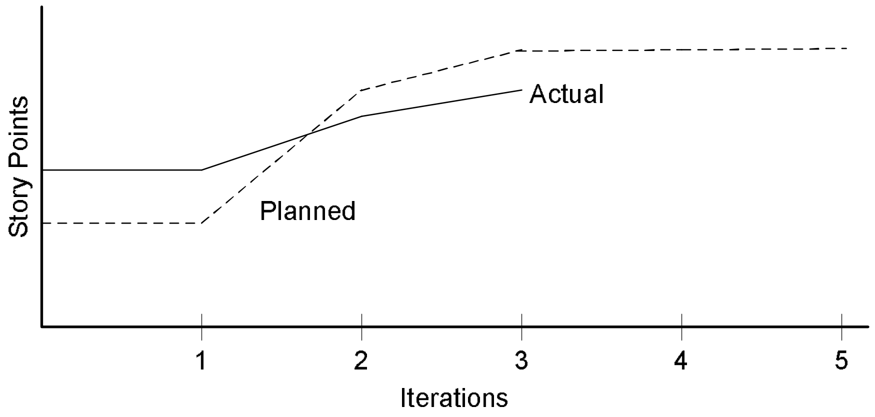


FIGURE 13.1 Planned and actual velocity after the first three iterations.

The team of Figure 13.1 would have been wrong if, at the conclusion of the first iteration, they told the customer they were exceeding planned velocity and could move up the delivery date. What about after the three iterations shown? Can the team tell if they should adjust the customer's expectations about the release plan? To answer that question the team needs both the velocity graph of Figure 13.1 as well as the cumulative story point graph of Figure 13.2.

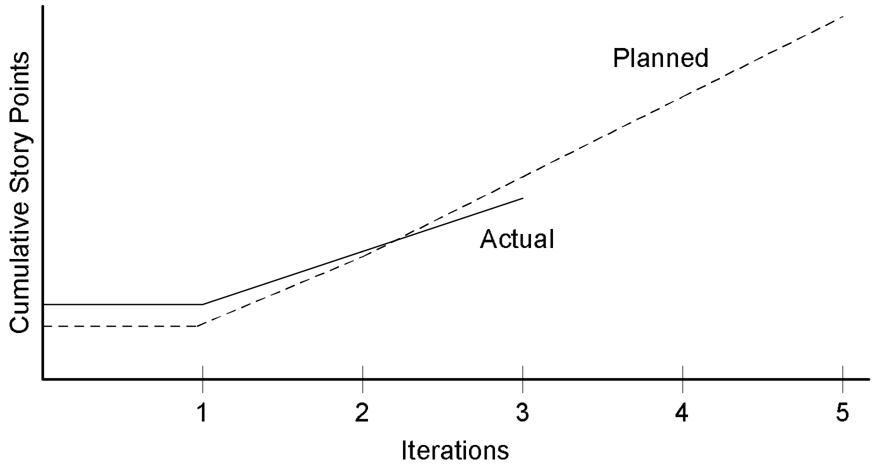


FIGURE 13.2 Plotting cumulative planned and actual story points.

The cumulative story point chart shows the total number of story points completed through the end of each iteration. So, we can see in Figure 13.2 that through the end of the second iteration the team had completed more story points than planned, even though progress in the second iteration was much slower than planned. However, by the end of the third iteration the advantage of the team's good start in the first iteration has been eroded by slower progress during the second and third iterations.

By the end of the third iteration it appears probable that the team will not complete as much functionality as planned. If the customer is not aware of this from daily interaction with the team, the situation should be made clear to her.

Iteration Burndown Charts

Another useful way of looking at progress is through the use of a *iteration burndown chart*. An iteration burndown chart shows the amount of work, expressed in story points, remaining at the end of each iteration. An example is shown in Figure 13.3.

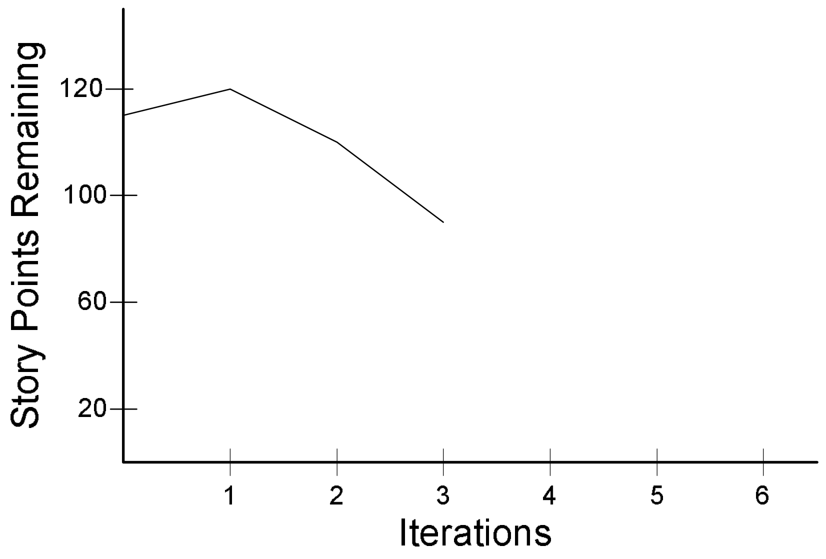


FIGURE 13.3 An iteration burndown chart.

An interesting feature of a burndown chart is that it reflects both progress made in the form of story points completed as well as changes to the number of story points planned for the remainder of the release. For example, suppose a team completes twenty story points during an iteration but the customer adds fifteen story points worth of work to the project. The twenty story points reflects a net gain of only five; if the team is working optimally then the customer may have to slow the introduction of new work if she expects the project to be finished quickly.

Figure 13.3 shows a team that actually had negative overall progress during the first iteration. They started the first iteration with 115 story points to complete and ended the iteration with 120. Managers and customers need to be careful that they don't read a burndown chart like Figure 13.3 and go yell at the team. What we can't tell from a burndown chart is how fast the team is moving. The team on Figure 13.3 may have completed 90 story points; but the customer may have added 95. To know how many story points the team is completing look at a velocity chart (like the one shown in Figure 13.1) or a cumulative story point chart (like the one shown in Figure 13.2).

Burndown charts are useful—even though they don't show the speed of the development team—because they present a better overall view of the project's progress. One of the strengths of agile software development is that projects can begin without a lengthy upfront complete specification of the project's requirements. Agile teams acknowledge that it is impossible for customers to know everything in advance. So, agile teams instead ask customers to tell them as much as they can and allow customers to change or refine their opinions as the project progresses and everyone learns more about the software being built. This means that stories will come and go, that stories will change size, and that stories will change in importance. As an example of this consider the project shown in Table 13.2.

TABLE 13.2 Progress and changes during four iterations.

	Iteration 1	**Iteration 2**	**Iteration 3**	**Iteration 4**
Story points at start of iteration	130	104	69	22
Completed during iteration	45	47	48	22
Changed estimates	7	4	–3	
Story points from new stories	12	8	4	
Story points at end of iteration	**104**	**69**	**22**	**0**

The team on this project thought they could do around 45 story points each iteration. They started with 130 story points and a plan to run for three iterations. They completed exactly 45 story points in the first iteration. But, in completing those stories they decided that a few of the remaining stories were bigger than initially thought and they increased the estimates on the unstarted stories by seven story points. Additionally, the customer wrote six new stories, each of which was estimated to be two story points. This means that while the team completed 45 story points their net progress is 45–

7–12, or 26 story points. This means that at the end of the first iteration they have 104 story points remaining.

The second iteration followed a similar trend with the team completing 47 story points but increasing estimates of unstarted stories by 4 points. The customer slowed the pace of change but still added new stories worth 8 story points. Net progress during the second iteration was 47–4–8=35 story points.

The team started the third iteration with 69 story points left. Things went well and the team completed stories worth 48 story points. They also reduced estimates of unstarted stories by three story points. (Remember that in the previous two iterations estimates of unstarted work had gone up.) The customer added a story or two worth a total of four story points. Net progress during the third iteration was 48+3–4=47. This left only 22 story points for the fourth iteration, which the team completed without any further changes.

A burndown chart for this project is shown in Figure 13.4. As early as the conclusion of the first iteration the customer should have been able to see from the slope of the burndown line that the project would not be finished in three iterations.

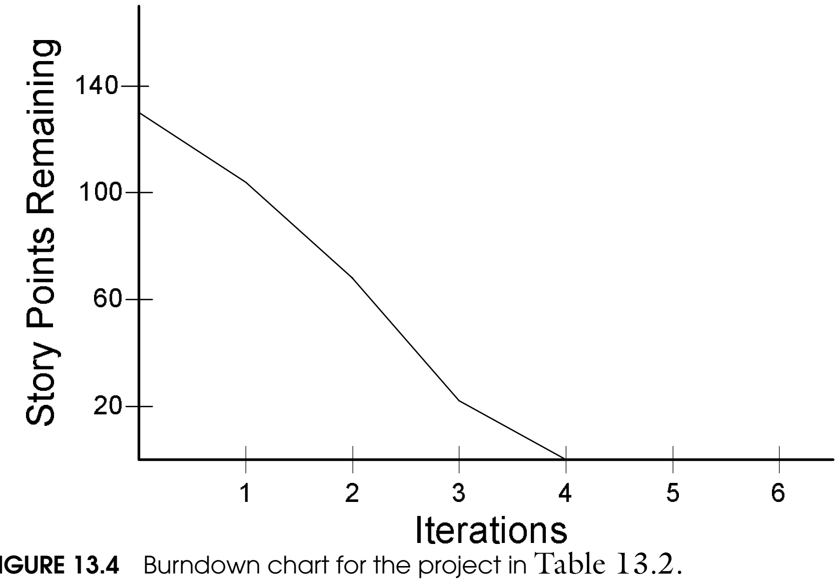


FIGURE 13.4 Burndown chart for the project in Table 13.2.

Burndown Charts During An Iteration

Beyond their usefulness for tracking progress at the end of iterations, burndown charts are also a great management tool during the iteration. During an iteration a ***daily burndown chart*** shows the estimated number of hours left in the iteration. For example, see Figure 13.5, which shows a daily tracking of the hours remaining in an iteration.

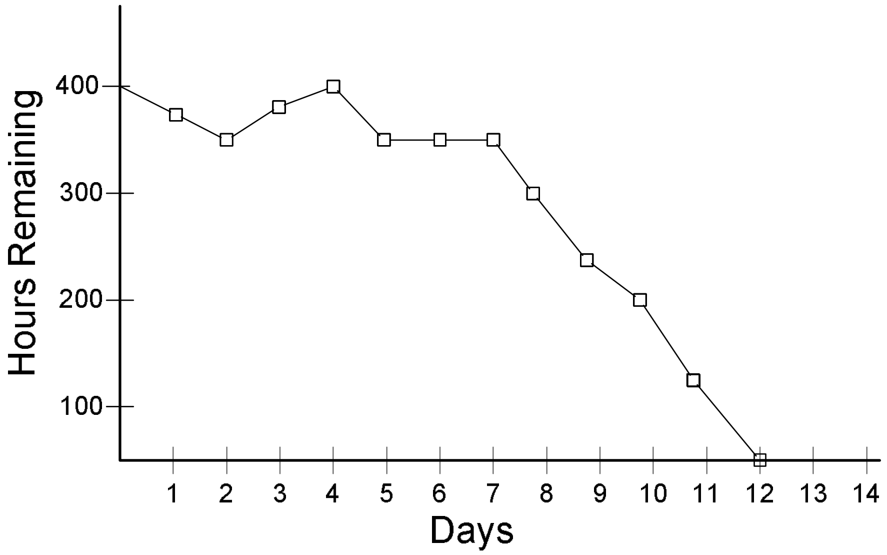


FIGURE 13.5 A daily burndown chart.

I prefer to collect information on the remaining level of effort by having each team member adjust his hours remaining on a common whiteboard. When iteration planning is done, the notes are left on the whiteboard that was used. At that point the board will contain a list of stories with each story decomposed into one or more tasks. Next to each task is a place for the programmer to sign up for the tasks. About once a day I add up the numbers on the whiteboard and add them to a burndown chart for the iteration. Near the start of an iteration a portion of such a whiteboard will look similar to Figure 13.6.

Create HTML Page	Thad	~~2~~ 0
Read search fields from HTML form.	Thad	2
Create better sample data	Mary	~~2~~ 4
Write servlet to perform search	Mary	4
Generate results page	Thad	2

FIGURE 13.6 Estimates are written, and frequently revised, on a whiteboard.

Figure 13.6 shows that the task "Create HTML page" has been completed as its estimate of remaining work has gone from two hours to none. However, Mary has increased her estimate for the "Create better sample data" task. It doesn't matter whether Mary hasn't started but changed her mind or whether she's already spent two (or four or six) hours on it and thinks there are four to go. All that matters is that the estimate on the board reflect her current thinking about how much work is left. These numbers do not include any local safety—that is, they are the 50% likely estimates of Chapter 9, "Estimating User Stories."

Hours Remaining, Not Hours Expended

Notice that daily burndown charts reflct the amount of work remaining, not the amount of work expended on a story or task. There may be a few good reasons for tracking hours expended (such as to compare actual to planned effort to improve estimating skills or to monitor the number of productive hours spent each week). However, these are outweighed by the reasons not to capture hours expended (such as the feeling of being micromanaged most developers will feel and the effort or imprecision in the numbers).

Besides, what really matters is how much effort is left, not how much effort has been applied so far.

Everyone on the team knows to keep the numbers relatively current. It usually works best if the board is updated whenever a task is finished and at the end of the day. This way everything is always relatively current. Everyone on the team needs to be encouraged to have the estimates of remaining work be as accurate as possible. Because the estimates do not have any safety margins built in, about half of the tasks should take longer than initially expected. This means that team members can increase the estimate of time remaining any time they want.

Naturally, new tasks can be added, too. However, a new task should be added only when someone realizes that a task was forgotten and the forgotten task is necessary for the completion of a story already included in the current iteration. New tasks should not be added just because someone wants something new included in the iteration; that type of change needs to be prioritized into an iteration at the next planning session.

Monitoring Quality

Occasionally, when a team tries to increase its velocity, quality suffers. Because of this it is worth monitoring the relationship between velocity and the number of defects. Figure 13.7, for example, shows the number of defects found per story point at various velocities reported by a team of eight developers running two-week iterations.

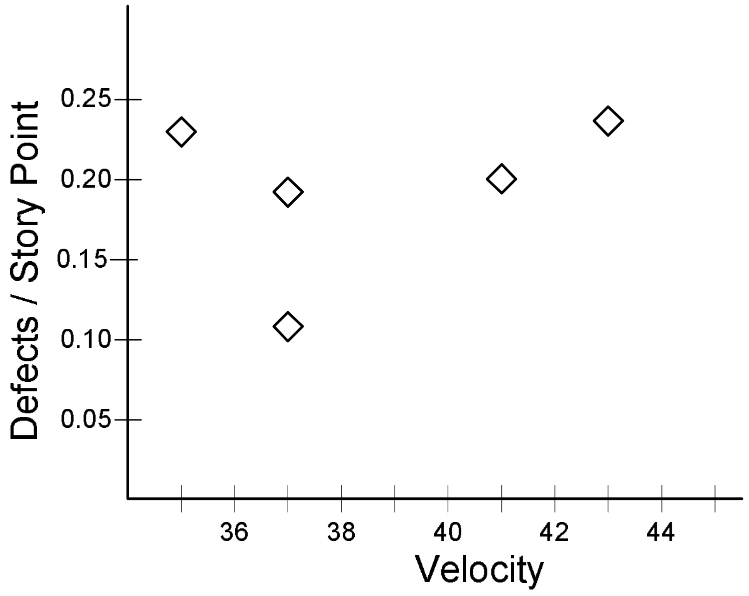


FIGURE 13.7 The relationship between velocity and quality.

Figure 13.7, whose underlying data is shown in Table 13.3, shows that at a velocity of 41 there were 0.20 defects per story (or one defect for every five story points). I count only defects found by the testers on the team, not defects found by programmers unit testing.

TABLE 13.3 Calculation of defects per story point.

Iteration	Velocity	Defects	Defects / Story Point
1	35	8	0.23
2	37	7	0.19
3	43	10	0.23
4	37	4	0.11
5	41	8	0.20

Figure 13.7 does not show a strong correlation between velocity and the number of defects per story point. This team ranges between 35 and 43 story points per iteration without the defect frequency changing much. They did have one iteration with a really low frequency of defects but that looks more like an anomaly itself than an indicator that quality suffers when velocity increases.

Summary

- When determining velocity count only finished stories. Do not count stories the team partially completed during the iteration.
- A good way to monitor differences between actual and planned velocity is to graph both the number of story points planned and actually completed for each iteration.
- Don't try to predict trends in velocity after only one or two iterations.
- The number of actual hours spent completing a task or a story have no bearing on velocity.
- A cumulative story point (as shown in Figure 13.2) is useful because it shows the the total numer of story points completed through the end of each iteration.
- An iteration burndown chart (as shown in Figure 13.3) shows both progress in the form of story points completed as well as changes to the number of story points planned for the remainder of the release.
- A daily burndown chart, showing the hours left on each day of an iteration, is very useful during an iteration.
- Charting and watching a team's defects per story point helps indicate if increases in velocity are coming at the expense of defects.

Developer Responsibilities

- To the extent possible, you are responsible for completing one story before moving onto the next story. It is preferable to have a small number of completed stories than to have a slightly larger number of stories all incomplete.
- You are responsible for understanding the impact of any decision you make on the velocity of the project.
- You are responsible for understanding how to read and iterpret each of the charts shown in this chapter.
- If you are a manager or perhaps in the Tracker role on an XP project you are responsible for knowing how and when to create the charts shown in this chapter.

Customer Responsibilities

- You are responsible for understanding how to read and iterpret each of the charts shown in this chapter.
- You are responsible for knowing the velocity of the team.
- You are responsible for knowing how actual velocity compares to the planned velocity and whether plan corrections are needed.
- You are responsible for understanding the impact of adding new stories to a release.

Questions

1. A story estimated at one story point actually took two days to complete. How much does it contribute to velocity when calculated at the end of the iteration?
2. What can you learn from an daily burndown chart that you can't see on an iteration burndown chart?

3. What conclusions should you draw from Figure 13.8? Does the project look like it will finish ahead, behind, or on schedule?

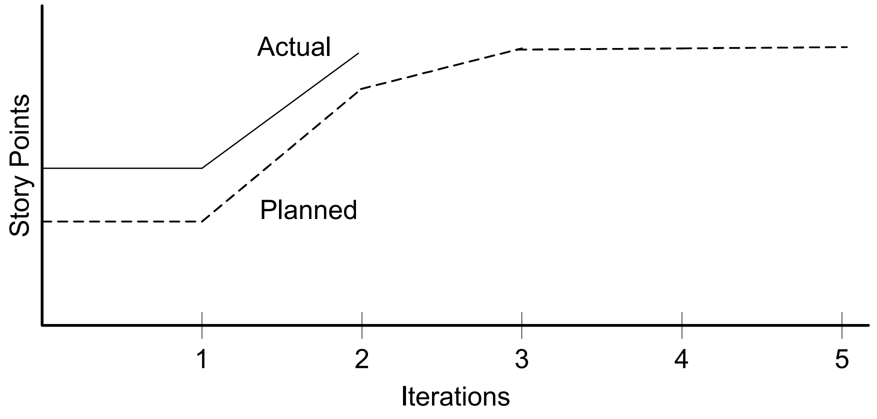


FIGURE 13.8 Will this project finish ahead, behind or on schedule?

4. What is the velocity of the team that finished the iteration shown in Table 13.4?

TABLE 13.4 Stories completed during an iteration.

Story	Story Points	Status
Story 1	4	Finished
Story 2	3	Finished
Story 3	5	Finished
Story 4	3	Half finished
Story 5	2	Finished
Story 6	4	Not started
Story 7	2	Finished
Velocity	**23**	

5. What circumstances would cause an iteration burndown chart to reflect an upward trend?

6. Complete Table 13.5 by writing the missing values into the table.

TABLE 13.5 Fill in the missing values..

	Iteration 1	**Iteration 2**	**Iteration 3**
Story points at start of iteration	100		
Completed during iteration	35	40	36
Changed estimates	5	–5	0
Story points from new stories	6	3	
Story points at end of iteration	**76**		**0**

Chapter 14

Acceptance Testing User Stories

Acceptance tests fill in the details of a story. They also provide criteria that can be used to determine when a story is fully implemented.

One reason for writing acceptance tests is to express many of the details that result from the conversations between customers and developers. Rather than writing lengthy lists of "The system shall…" style requirements statements, tests are used to fill in the details of a user story. When writing stories on note cards it is easiest to write the tests on the back of the card. For example, the story "A user can pay with a credit card" may be tested with the following:

- Test with Visa, MasterCard and American Express (pass)
- Test with Diner's Club (fail)
- Test with good, bad and missing card id numbers
- Test with expired cards

- Test with different purchase amounts (including one over the card's limit)

Second, tests capture assumptions made by the customer. Suppose the customer writes the story "Users can view details about a specific hotel." The customer and a developer discuss the story and identify a set of facts that will be displayed about a hotel—number of rooms, types of rooms, number and size of pools, types of restaurants, and so on. However, the customer knows that it will be impossible to get this information for all hotels initially and she *assumes* that a default page will be displayed. That should be reflected as a test because the programmer may *assume* that every hotel will have information available.

Third, acceptance tests are written to provide basic criteria that can be used to determine if a story is fully implemented. Having criteria that tell us when something is done is the best way to avoid putting too much, or too little, time and effort into it. For example, when my wife bakes a cake her acceptance test is to stick a toothpick into it. I acceptance test her cake by running a finger through the frosting.

Tests Are Written Before Coding

Acceptance tests provide a great deal of information that the programmers can use in advance of coding the story. For example, consider "Test with different purchase amounts (including one over the card's limit)." If this test is written before a programmer starts coding, it will remind her to handle cases in which a purchase is declined because of insufficient credit. Without seeing that test some programmers would forget to support this case.

Naturally, in order for programmers to benefit in this way the acceptance tests for a story must be written before programming begins on that story. Tests are generally written at the following times:

- Whenever the customer and developers talk about the story and want to capture explicit details
- As part of a dedicated effort at the start of an iteration but before programming begins
- Whenever new tests are discovered during or after the programming of the story

Ideally, as the customer and developers discuss a story they reflect its details as tests. However, at the start of an iteration the customer should go through the stories and write any additional tests she can think of. A good way to do this is to look at each story and ask questions similar to the following:

- What else do the programmers need to know about this story?
- What am I assuming about how this story will be implemented?
- Are there circumstances when this story may behave differently?
- What can go wrong during the story?

The Customer Writes the Tests

Because the software is being written to fulfill a vision held by the customer the acceptance tests should be written by the customer. Ideally this means the customer does the physical writing of the tests but in many cases the tests are written by developers who learn the tests by talking with the customer. Additionally, a development team (especially one with experienced testers on it) will usually augment some of the stories with tests they think of.

How Many Tests Are Too Many?

The customer should continue to write tests only as long as they add value and clarification to the story. For example, it is not necessary to write a story that says "Test that hotel rooms can be reserved on February 29, 2005" or "Verify that hotel rooms can be rented on June 30 but not June 31." Tests that verify proper date handling should be performed but any decent programmer should recognize common tests like this. A good programming team will have unit tests in place for tests like this. The customer is not responsible for identifying every possible test. The customer should focus her efforts on writing tests that clarify the intent of the story to the developers.

The Framework for Integrated Test

Acceptance tests are meant to demonstrate that an application is acceptable to the customer paying for the application (or its development). This means that the customer should be the one to execute the acceptance tests. Acceptance tests should be executed at the end of each iteration. Because working code from one iteration may be broken by development in a subsequent iteration it is important to execute acceptance tests from all prior iterations. This means that executing acceptance tests gets more time consuming with each passing iteration. If possible, the development team should look into automating some or all of the acceptance tests.

One excellent tool for automating acceptance tests is Ward Cunningham's Framework for Integrated Test[1], or FIT for short. Using FIT, tests are written in a familiar spreadsheet or tabular format. Bob and Micah Martin have led the development of a FitNesse[2], an extension of FIT that makes test writing even easier.

1. The Framework for Integrated Test, FIT, is available at fit.c2.com.
2. FitNesse is availabe from www.fitnesse.org.

FitNesse (which uses FIT) is rapidly becoming a very popular approach for writing acceptance tests on agile projects. Because tests are expressed in spreadsheet-like tables, the effort for customers to identify and write tests is greatly reduced. Table 14.1 is a sample of the type of table that can be processed by these tools. Each row represents one set of data. In this case the first data row identifies a Visa card that expires in May of 2005 and has the number 4123456789012. The final column indicates whether this card should pass a validity check in the application. In this case the card is expected to be considered valid by the application.

TABLE 14.1 Testing for valid credit cards with a table that can be used by FIT and FitNesse.

CardType	Expiration	Number	Valid()
Visa	05/05	4123456789012	true
Visa	05/23	4123456789012345	false
MasterCard	12/04	5123456789012345	true
MasterCard	12/98	5123456789012345	false
MasterCard	12/98	42	false
American Express	4/05	341234567890123	true

In order to execute the tests in Table 14.1 a programmer on the team needs to write code to respond to simple FIT commands. She writes code that invokes code in the application being tested to determine the validity of the credit card. However, to write the acceptance tests, the customer only needs to create a simple table like this one that shows data values and expected results.

When the tests in Table 14.1 are run the results are displayed by coloring the test columns (the Valid column in this example) either green (for a passed test) or red (for a failed test). FitNesse and FIT make it simple for either a customer or developer to run the acceptance tests.

Summary

- Acceptance tests are used to express details that result from conversations between a customer and a developer.
- Acceptance tests document assumptions about the story a customer has that may not have been discussed with a developer.
- Acceptance tests provide basic criteria that can be used to determine if a story is fully implemented.
- Acceptance tests should be written by the customer rather than by a developer.
- Acceptance tests are written before the programmer begins coding.
- Stop writing tests when additional tests will not help clarify the details or intent of the story.
- FIT and FitNesse are excellent tools for writing acceptance tests in a familiar table or spreadsheet format.

Developer Responsibilities

- You may be responsible for automating the execution of acceptance tests if your team chooses to do so.
- You are responsible for thinking about additional acceptance tests when you start development of a new story.
- You are responsible for unit testing your code so that acceptance tests do not need to be specified for all the minutiae of a story.

Customer Responsibilities

- You are responsible for writing the acceptance tests.
- You are responsible for executing the acceptance tests.

Questions

14.1 Write tests for the story “When making an airline reservation users can request special meals.”

14.2 Write tests for the story “Users can revise previously entered hotel, flight and car preferences.”

Chapter 15

A Catalog of Story Smells

Things are not always rosy when using stories. This chapter describes what to do when your project starts smelling less like a rose than it should.

This chapter will present a catalog of "bad smells," that is, indicators that something is amiss in a project's application of user stories. Each smell will be described and one or more solutions provided.

Stories Are Too Small

Small stories often cause problems with the estimating and scheduling of stories. This happens because the estimate assigned to a small story can change dramatically depending on the order in which the story is implmented. For example, consider these two small stories from a hypothetical word processor:

- Text may be made bold by pressing Ctrl–B.
- Text may be made bold by selecting a button on the toolbar.

Suppose it takes an hour for a programmer to write the code that makes text display in bold. In addition to that hour it probably takes the programmer five minutes to make the code respond to a press of the Ctrl-B keys and five minutes to add a toolbar button that invokes the code. This means that both stories can be added in a total of 70 minutes. However, we cannot split the 70 minutes equally among the two stories. Neither story is likely to take 35 minutes. Instead, whichever story is implemented first will take 65 minutes. The story implemented second will take 5 minutes. However, since there's nothing about these stories that implies which will be developed first, the stories should be combined.

Interdependent Stories

Similar to the bad smell of stories that are too small is the smell of interdependent stories. When two or more stories are dependent upon one another it becomes difficult to plan the individual stories into iterations. The team finds itself in a situation where a particular story may only be added to an iteration if another story is also added to the iteration. But that story may only be added if a third story is also added, and so on. As with stories that are too small, the solution to too many interdependent stories is to write larger stories.

Goldplating

"Goldplating" refers to the addition of unnecessary features by the developers. Some developers have a tendency to add enhancements or to go beyond what is needed to satisfy the customer's stated needs. For example, on one project I was on we had a story to take a very crowded screen and rewrite it as a tabbed dialog to improve usability. After he finished coding this, the developer enhanced the low-level tab dialog code in this application so that a tab could be torn from its current location and moved about the screen. This was not something the customer asked for.

In this case the customer liked the new functionality so this developer's goldplating was not a complete waste of time. However, the new functionality was probably not something the customer would have prioritized into the iteration if she had been given the choice.

If a project is experiencing significant goldplating by developers it can be prevented by increasing the visibility of the tasks that everyone is working on. For example, hold brief daily meetings where everyone says what he or she is working on. Hold end–of–iteration review meetings where all of the new functionality is demonstrated in detail for the customer and other stakeholders. If the project has a QA organization they can also help identify goldplating, especially if they were involved in the conversations between the programmer and the customer.

Too Many Details

One of the benefits to writing stories on notecards is that the space available for writing the story is quite limited. It is hard to cram lots of detail onto a small notecard. Including too many details in a story is indicative of placing too much value on documentation and favoring it over conversation.

Tom Poppendieck[1] has made the observation that "If you run out of room, use a SMALLER card." This is a great idea because it will force the story writers to very consciously include fewer details in the stories.

Including User Interface Detail Too Soon

At some point on a project the team will definitely write user stories with very direct assumptions about the user interface. For example, "From the Hotel Information screen the user can access

1. Poppendieck, Tom. 2003. *The Agile Customer Toolkit,* Software Development West.

photographs of the hotel." However, for as long as possible you should avoid writing stories with that level of detail.

Early in the project you do not know that there will be a "Hotel Information screen" so avoid constraining the project by associating stories with it.

Thinking Too Far Ahead

Teams that are used to a large upfront "requirements engineering" effort and move to using stories may have a tendency to think too far ahead. The result will be that they add too much detail to user stories and spend too much time estimating.

Indicators of this smell may be that stories are hard to fit on note cards, there is interest in using a software system instead of note cards, someone proposes a story template to capture all of the details necessary for a story, or perhaps that a suggestion is made to give estimate in finer precision (for example, hours instead of days).

For a team to overcome this smell they may need a refresher course on the strengths of stories. Fundamental to the use of stories is the recognition that for most problems it is impossible to identify all requirements in advance. Good software emerges through repeated iterations in which increasing amounts of detail are added to the software. Stories fit well with this approach because of the ease with which detail can be expressed in later versions of a story. The team may need to remind itself what it was about their prior development process that led them to adopt stories.

Splitting Too Many Stories

When the developers and customer select the stories they will move into an iteration they will sometimes need to split a story into two or more constituent stories. Typically a story needs to be split during planning for one of two reasons:

1. The story is too large to fit into the iteration

2. The story contains both high and low priority aspects and the customer only wants the high priority portions done during the coming iteration

Neither of these cases is representative of a problem. Many projects and teams will have occasions when it will be useful to split a story to accommodate the duration of a sprint and fit with the team's observed velocity. However, the situation begins to smell when the team finds itself splitting stories frequently. If stories are being split more often than feels reasonable, then the stories are probably too large. Consider taking a pass through the remaining stories and looking for stories that should be split. This should make it easier to estimate those stories and to prioritize all of the stories.

Customer Has Trouble Prioritizing

To start each iteration the project's customer prioritizes the remaining stories and helps the development team select the stories they will plan on implementing during the coming iteration. Choosing among and prioritizing stories is often difficult; but sometimes prioritizing the stories is so difficult that it can be considered a smell.

If a customer is having trouble prioritizing stories the first thing to consider is the size of the stories. If stories are too small they become difficult to prioritize both because there are more of them and because it can be difficult to compare very small new capabilities. For example, consider these two small stories from a travel reservation system:

- ✧ Users can search for a hotel by quality rating (for example, 3 stars).
- ✧ Users can search for a hotel by type of room (for example, suite, penthouse, and so on).

Depending on the system being built these stories are probably too small for the customer to easily choose between. The customer would probably be better off with a more generic story like "Users can access an advanced search capability whenever the basic search isn't powerful enough."

Large stories can cause just as many problems. Suppose, at the extreme, the travel reservation system included only the following three stories:

- ✧ Users can search for and reserve hotels.
- ✧ Users can search for and book flights.
- ✧ Users can search for and reserve rental cars.

The poor customer who has to prioritize only among these stories! Her reaction is probably to say, "But, can't I have a little of this and some of that?"

Finally, it may be difficult to prioritize the stories because they have not been written to express business value. For example, suppose the customer is presented with stories like these:

- ✧ Users connect to the database via a connection pool.
- ✧ Users can view detailed error information in a log file.

Stories like these can be very difficult for the customer to prioritize because the business value of each is not clear. The stories should be rewritten so that the value of each is clear to the customer. How each is rewritten will vary depending on the customer's technical knowledge, which is why the best recommendation is to have the customer write the stories herself. For example, consider these rewritten versions of the preceding stories:

- ✧ There will be no noticeable lag when a user starts the application and it connects to the database.
- ✧ Whenever an error occurs users are given enough information to know how to correct the error.

Summary

In this chapter we learned about the following smells:

- Stories that are too small
- Interdependent stories
- Goldplating
- Adding too many details to stories
- Including user interface details too soon
- Thinking too far ahead
- Splitting too many stories
- Trouble when prioritizing stories

Developer Responsibilities

- You share a responsibility with the customer to be aware of these smells—as well as any others you may detect—and then to work to correct any that affect your project.

Customer Responsibilities

- You share a responsibility with the developers to be aware of these smells—as well as any others you may detect—and then to work to correct any that affect your project.

Chapter 16

Using Stories Outside XP

Extreme Programming is not the only place where stories can be used. In this chapter we consider another agile process, Scrum, and see how it benefits from the use of stories.

User stories originated as part of Extreme Programming. Naturally, stories fit perfectly with the other practices of Extreme Programming. However, stories also work well as the requirements approach for other processes.

In this chapter we'll look at Scrum, another agile process, and will see how stories can be integrated as an important part of Scrum. Terms that are part of the Scrum lexicon will be italicized when first used.

Scrum is Iterative and Incremental

Like XP, Scrum is both an iterative and an incrmental process. Since these words are used so frequently without definition, we'll define them.

An iterative process is one that makes progress through successive refinement. A development team takes a first cut at a system, knowing it is incomplete or weak in some (perhaps many) areas. They then iteratively refine those areas until the product is satisfactory.

Additionally, an iterative process is characterized by the repeated application of a series of steps. For example, XP progresses from planning to testing to coding to testing to delivery and then repeats. When I look up a word in the dictionary I iteratively apply the process of

- See if the word is between the first and last word on the two open pages
- Turn an appropriate number of pages in the right direction until the word is between the first and last word on the two pages
- Find the word on the page and read its definition

Each time the steps of an iterative process are applied the software is improved through the addition of greater detail. For example, in a first iteration the search screen may be coded to support only the simplest type of search. The second iteration may add additional search criteria. Finally, a third iteration may add error handling. Larman and Basili refer to this as "evolutionary refinement."[1]

A good analogy is sculpting. First the sculptor selects a stone of the appropriate size. Next the sculptor carves the general shape from the stone. At this point one can perhaps distinguish the head and torso and discern that the finished work will be of a human body rather than a bird. Next, the sculptor will refine her work by adding detail. However, the sculptor is unlikely to look on any one area as complete until the entire work is complete.

An incremental process, on the other hand, is one in which software is built and delivered in pieces. Each piece, or increment, repre-

1. Larman, Craig and Victor R. Basili, IEEE Computer, June 2003, pp. 47-56.

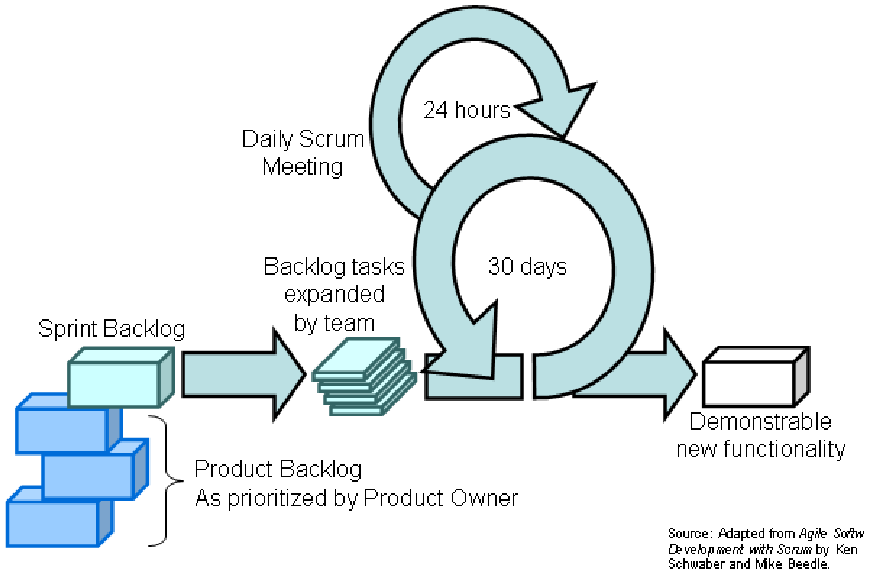
sents a complete subset of functionality. The increment may be either small or large; perhaps ranging from just a system's login screen on the small end to a highly flexible set of data management screens. Each increment is fully coded and tested and the common expectation is that the work of an iteration will not need to be revisited.

An incremental sculptor would pick one part of her work and focus entirely on it until it's finished. She may select small increments (first the nose, then the eyes, then the mouth, and so on) or large increments (head, torso, legs and then arms). However, regardless of the increment size the incremental sculptor would attempt to finish the work of that increment as completely as possible.

Scrum and XP are both incremental and iterative. They are iterative in that they plan on evolutionary refinement. They are incremental because completed work is delivered throughout the project.

The Basics of Scrum

Scrum projects progress through a series of month-long iterations called *sprints.* At the start of each sprint the team determines the amount of work it can accomplish during that sprint. Work is selected from a prioritized list called the *product backlog.* The work the team believes it can complete during the sprint is moved onto a list called the *sprint backlog.* A brief daily meeting, the daily scrum, is held to allow the team to inspect its progress and to adapt as necessary. Graphically, Scrum is as shown in Figure 16.1.



24 hours
Daily Scrum Meeting
30 days
Backlog tasks expanded by team
Sprint Backlog
Demonstrable new functionality
Product Backlog
As prioritized by Product Owner
Source: Adapted from *Agile Softw* *Development with Scrum* by Ken Schwaber and Mike Beedle.

FIGURE 16.1 Graphical representation of the Scrum process.

The Scrum Team

A Scrum team consists of typically four to seven developers. While a Scrum team may involve specialist developers—testers, and database administrators, for example—a Scrum team shares a "we're all in this together" attitude. If there is testing to be done and there is no dedicated tester available, then someone else does the testing. All work is owned collectively. Scrum teams are self-organizing. That is, there is no management directive stating that Mary codes and Bill tests. Everything about how a team accomplishes its work is left up to the team.

This core team is supplemented by two key people: the *product owner* and the *ScrumMaster*. The product owner is essentially the customer of Extreme Programming. The product owner is largely responsible for placing items onto and prioritizing the product backlog list of needed functionality. The ScrumMaster is similar to a project manager except that the role is much more one of leadership than of managing. Because Scrum teams are self-organizing and given great leeway in how they complete the work of a sprint the project's ScrumMaster serves the team rather than directs it. A ScrumMaster typically serves her team by removing obstacles to progress and by helping the team follow the few simple rules of Scrum.

The Main Rules of Scrum

- A sprint planning meeting is held at the start of each sprint.
- Each sprint must deliver working and fully tested code that demonstrates something of value to end-users or the customer.
- The product owner prioritizes the product backlog.
- The team collectively selects the amount of work brought into the sprint.
- Once a sprint begins, only the team may add to the sprint backlog.
- The product backlog may be added to or reprioritized at any time.
- A short scrum meeting is held every day. Each project participant answers: What did you do yesterday? What will you do today? What obstacles are in your way?
- Only active participants in the sprint (not interested observers or removed stakeholders) may speak during the daily scrum meeting.
- The result of a sprint is demonstrated at a sprint review meeting at the end of the sprint.
- Working software is demonstrated during the sprint review. No slideshows are allowed.
- No more than two hours may be spent preparing for the sprint review.

The Product Backlog

The product backlog is the master list of all functionality desired in the product. When a project is initiated there is no comprehensive effort to write down all foreseeable features. Typically, the product owner and team write down everything obvious, which is almost always more than enough for a first sprint. The product backlog is then allowed to grow and change as more is learned about the product and its customers.

An example product backlog from a real project appears as shown in Table 16.1. As you can see from this table, backlog items can be technical tasks ("Refactor the Login class to throw an exception") or more user-centric ("Allow undo on the setup screen").

TABLE 16.1 A sample product backlog list.

Number	Description
1	Finish database versioning
2	Get rid of unneeded Java in the database
3	Refactor the Login class to throw an exception
4	Add support for concurrent user licensing
5	Add support for evaluation licenses
6	Support wildcards when searching
7	Save user settings
8	Allow undo on the setup screen

The product owner sorts the product backlog into priority order. Even better, the product owner is allowed (actually encouraged) to shuffle the product backlog items as priorities change from month to month.

An approach I've had great success with is expressing Scrum backlog items in the form of user stories.[2] Rather than allowing product backlog items to describe new features, issues to investigate,

2. "The Upside of Downsizing," *Software Test and Quality Engineering,* vol. 5, no. 1, January 2003.

defects to be fixed, and so on the product backlog is constrained to only user stories. Each story in the product backlog must describe some item of value to a user or to the customer.

By constraining the product backlog to only user stories it becomes much easier for the product owner to prioritize the backlog. With all backlog items in terms she can understand it becomes easy for her to make decisions to trade one feature for another.

As with XP, in Scrum it is not important for the product owner to identify all of the requirements upfront. However, there is often a benefit to jotting down as many of them as possible at the outset. Scrum has no prescribed, or even recommended, approach to initially stocking the product backlog. Traditionally this has happened through an unstructured discussion between the product owner, ScrumMaster, and one or more developers. However, by first identifying user roles and then focusing on the stories for each user role Scrum is combined with a powerful requirements identification technique.

The Sprint Planning Meeting

A *sprint planning meeting* is held at the start of each sprint. The meeting usually lasts up to a full day and is attended by the product owner, the ScrumMaster, the entire team of developers. Also attending may be any interested and appropriate management or customer representatives.

During the first half of the sprint planning meeting the product owner describes the highest priority features to the team. The team asks enough questions that during the second half of the meeting they can determine which items they will move from the product backlog to the sprint backlog.

The product owner does not have to describe every item being tracked on the product backlog. Depending on the size of the backlog and the velocity of the team it may be sufficient to describe only the high priority items, saving the discussion of lower priority items for the next sprint planning meeting. Typically, the Scrum team will provide guidance when they start to get further into the backlog list than they know can be done in the upcoming sprint.

Collectively the team and the product owner define a *sprint goal*, which is a short description of what the team plans to achieve during that sprint. The success of the sprint will later be assessed during the *sprint review meeting* against the sprint goal, rather than against each specific item selected from the product backlog.

During the second half of the sprint planning meeting, the team meets separately to discuss what they heard and decide how much they can commit to during the coming sprint. Conceptually, the team starts at the top of the prioritized product backlog list and draws a line after the lowest of the high priority tasks they feel they can complete. In practice it is not unusual to see a team select, for example, the top five items and then two items from lower on the list but that are associated with the initial five. In some cases there will be negotiation with the product owner but it will always be up to the team to determine how much they can commit to completing.

Once the sprint starts only the team may add work to the sprint. The CEO cannot approach the team and ask for something the product owner left out. A salesperson cannot ask for one more feature for a special customer. And the product owner cannot change her mind and ask for additional features. The only time work may be added to a sprint is if the team finds itself ahead of schedule. At that time they can ask the product owner to help identify an additional item or two.

In exchange for committing to completing the work they commit to the organization commits that it will not change the contents of sprint for the duration of the sprint. If something so significant happens that the organization feels it needs to change the sprint then the current sprint is aborted and a new sprint is started by beginning with another sprint planning meeting.

As the team selects items from the product backlog they expand them into the more task-based sprint backlog. Each item on the product backlog may expand into one or more items on the sprint backlog, allowing the team to more effectively share the work.

Because, as we saw in Chapter 12, "Planning an Iteration," stories may be readily decomposed into their constituent tasks the fit between Scrum and user stories at this point is a good one.

Additionally, stories fit well into this model because they force the product backlog items to each deliver value that the customer can assess. Conventional Scrum backlog items like "refactor the Login class to throw an exception" can be meaningless to a product owner. When all backlog is expressed as user stories the Scrum product owner has a much easier time prioritizing the product backlog.

The Sprint Review Meeting

Each sprint is required to deliver a *potentially shippable product increment*. This means that at the end of each month-long sprint the team has produced a coded, tested, and usable piece of software. The software must be potentially shippable, meaning that the organization could choose to ship the software to customers (or use it internally) if enough new functionality is included to justify the overhead of shipping or deploying the new version. A commercial software distributor, for example, would probably not choose to ship a new version every month because of the upgrade hassles that could cause their customers. However, with Scrum that vendor would be required to produce a potentially shippable version each month.

At the end of each sprint a sprint review meeting is held. During this meeting the team shows what they accomplished during the sprint. Typically this takes the form of a demo of the new features.

The sprint review meeting is intentionally kept very informal, typically with rules forbidding the use of PowerPoint slides and allowing no more than two hours of preparation time for the meeting. A sprint review meeting should not become a distraction or significant detour for the team; rather, it should be a natural result of the sprint.

The entire team as well as the product owner and ScrumMaster participate in the sprint review meeting. Others who are interested in the project (such as management, customers, or engineers from other projects) may attend the sprint review if they desire.

During the sprint review meeting the project is assessed against the sprint goal that was previously determined during the sprint planning meeting. Ideally the team has completed each item planned for the sprint but it is more important that they achieve the overall goal of the sprint.

The use of stories benefits Scrum during the sprint review meeting because stories make it simpler to assess what parts of a sprint have been completed. On a Scrum project that uses a random collection of technical tasks, requirements, issues and bug fixes as its product backlog it can be difficult for the team to demonstrate that each item made its way into the product built during that sprint. When the entire product backlog is comprised of stories describing items of customer or user value it is generally easier to demonstrate those items.

The Daily Scrum Meeting

Scrum is probably the first documented process to include a daily stand-up meeting.[3] The idea, however, has quickly spread to many of the other agile processes such as XP and Feature-Driven Development. Whenever possible we want to choose the least time-consuming and the least-intrusive method of gathering and sharing project information. For many purposes, a daily standup meeting achieves exactly that goal.

The daily stand-up meeting is typically held as early as possible each day but after the entire team has arrived for work. This is typically 9:00 or 9:30. Everyone on the team is required to attend, including programmers, testers, the product owner, and the ScrumMaster. The meeting is kept short: usually 15 minutes or less and

3. "SCRUM: A Pattern Language for Hyperproductive Software Development," Mike Beedle et al., *Pattern Languages of Program Design,* Harrison, N., Foote, B., Rohnert, H. (editors), Addison-Wesley, 4:637–651, 1999.

never longer than 30. To keep the meeting short, many teams require participants to stand.

The first few times you try this there will almost certainly be some on the team who decide it's OK to sit. Don't let them. Yes, you can probably keep the meeting to 15 minutes even with one or two people sitting. However, I've typically found that the one or two who sit down are really trying to make the statement that while nominally a part of the team they are also better than it. By showing that they can get away with breaking the rules (even such a little rule as this) they desire to demonstrate their importance to everyone else on the team. Make them stand.

During the daily scrum each team member answers the following three questions:

1. What did you do yesterday?
2. What will you do today?
3. What is it in your way?

It is important that the daily scrum does not come across as a grilling by the ScrumMaster. One of the goals of the meeting is to have each developer make commitments in front of his or her peers. Commitments are not made to the manager or the company but to each other.

The scrum meeting should stick as closely as possible to the three questions above and should not move off into designing parts of the system or resolving issues that were raised. Such items are noted during the meeting but are resolved afterwards. It's OK to identify a subset of the team that will meet to resolve an issue but do not resolve issues during the stand-up meeting. For example, suppose someone asks whether we should start using the recently released version 5.0 of our vendor's application server. In that case, we agree that right after the standup meeting another meeting will occur between

- the technical architect, who can assess the technical impact on using the new application server;

- the product owner, who comes from the Marketing department and will be in the best position to decide if our customers will deploy the old or new application server; and,
- a representative from the test team who can assess the impact on her group.

The ScrumMaster is on hand as a facilitator and to make sure the meeting stays focused on the three questions and moves along at a brisk pace. The product owner is present because she will ideally have work to report on just like anyone else ("I finished writing tests for the 'Add book to shopping cart' story and today I'm going to do some market research on which cards we should accept. I should be able to finish that by the end of the day").

One benefit to holding daily stand-up meetings is that they can serve as random checkpoints for senior managers or anyone else interested in the state of the project. By consistently meeting at the same time of day and extending a general "join us whenever you're interested" invitation the team may be able to get out of more onerous meetings such as monthly project reviews. However, if non-team members are invited to the daily scrums be sure to establish a rule that only people directly on the project can ask questions during the meeting. So, the Big Boss can attend and listen to her heart's content. She cannot, however, redirect the meeting by asking questions.

The daily scrum provides everyone on the team a quick daily snapshot of where things stand on the project. This makes it the perfect time for the team to reconsider current assignments. For example, suppose Randy reports that the story he's working on is taking much longer than he expected and Andrew reports that he is significantly ahead of schedule. In this case it may be appropriate for Andrew to spend the day paired with Randy and working on Randy's tasks or for Andrew to outright take on responsibility for some of Randy's tasks.

The ScrumMaster must walk a fine line during the daily scrums. She needs to keep the pace brisk but cannot let the meetings feel as though they are for her benefit alone. One question that should never be asked is “How much time do you have left on the ‘Order a book’ story?” This information is extremely important but if it is asked for during the standup meeting then the standup meeting will become all about estimates and numbers. I have the team update their estimates on a communal whiteboard or in the software we’re using if not collocated.

I have found stories beneficial to daily scrum meetings because they help make sure focus remains on customer and end-user desires. Because there is no upfront requirements or analyis phase that precedes a sprint, sprints are started with only a partial understanding of exactly what will be built. The team may know they are planning to add a search screen but they may not know what fields will be searchable, how search criteria may be combined, and so on. Stories are useful because they remind the team about the intent behind what is being developed. In the midst of a sprint the team can use the story (as well as ongoing discussions with the product owner about the story) to determine whether they have gone far enough, or perhaps too far, in programming a particular story.

Summary

- ✧ Scrum is an iterative and an incremental process.
- ✧ Scrum projects progress in a series of four-week iterations called sprints.
- ✧ A ScrumMaster fills the project manager role but is more leader and facilitator than manager.
- ✧ A typical Scrum team includes four to seven developers.
- ✧ The product backlog is a list of all desired features that have not yet been either added to the product or planned for the current sprint.
- ✧ The sprint backlog is a list of tasks the team has committed to for the current sprint.

- XP's customer role is played by the Scrum product owner.
- The product owner prioritizes the product backlog.
- At the start of the sprint the team selects what and how much work they commit to competing during the sprint.
- Brief daily scrum meetings are held. During these meetings each team member says what she completed yesterday, what she'll complete today, and identifies any obstacles in her way.
- Each sprint is responsible for producing a potentially shippable product increment.
- At the end of sprint the team demonstrates the software it created in a sprint review meeting.

Questions

16.1 Describe the differences between an incremental and an iterative process.

16.2 What is the relationship between the product backlog and the sprint backlog?

16.3 What is meant by a potentially shippable product increment?

16.4 Who is responsible for prioritizing work and for selecting the work the team will perform during a sprint?

16.5 What questions are answered by each team member at the daily scrum?

Part 4

An Example

Chapter 18

The User Roles

Identifying user roles is not a required step in writing good user stories. But it can be accomplished quickly and helps most project teams identify stories they would otherwise miss.

Over the course of the next five chapters we will undertake a small, hypothetical project. In this chapter we'll start by identifying the key user roles. In subsequent chapters we'll move on to writing stories, estimating the stories, planning a release, and writing acceptance tests for the stories in the release.

The Project

Our company has been selling sailing supplies through a print catalog for thirty years. Our catalogs feature items such as Global Positioning Systems, clocks, weather equipment, navigation and plotting equipment, life rafts, inflatable vests, charts, maps, and books. So far our Internet presence has been a simple one page site directing people to call a toll-free number to request a catalog.

Our boss has decided we should finally get with the times and start selling things over the Internet. However, rather than start by selling some of our bigger ticket items he wants us to start by selling just books. Some items in our catalog cost over $10,000 and until we know our site works well and doesn't lose orders, we don't want to take any risk with expensive items. But if we find that our customers like being able to order online, and if we do a good job on the site, we'll expand and sell the rest of our products on the site.

Oh, and the last thing the boss says is that the site needs to be live in thirty days.

Finding a Customer

We know we need a customer to help us identify and write stories. The customers for this product are the sailors who buy the books and they are all outside the company. So, we need an internal customer who can act as a proxy for the real end-user customers. For this the boss designates Lori, who is the Vice President of Sales and Marketing.

In an initial meeting with Lori she provides more background on the system she needs. She wants a "typical bookstore / eCommerce site." She wants customers to be able to search for books in a variety of ways (we don't push for clarification at this point), she wants users to be able to maintain lists of books they'll buy later, she wants users to rate and review books they buy, and check on the status of an order. We've seen plenty of sites like this so we tell Lori we're ready to start.

Identifying Some Initial Roles

The first thing we do is gather some of the developers together with Lori in a room with a large table. They write the following user role cards, which are placed as shown in Figure 18.1:

- Hardcore Sailor

- Novice Sailor
- New Sailor
- Gift-Buyer
- Non-Sailing Spouse
- Administrator
- Sales Vice President
- Charter Captain
- Sailing School
- Instructor

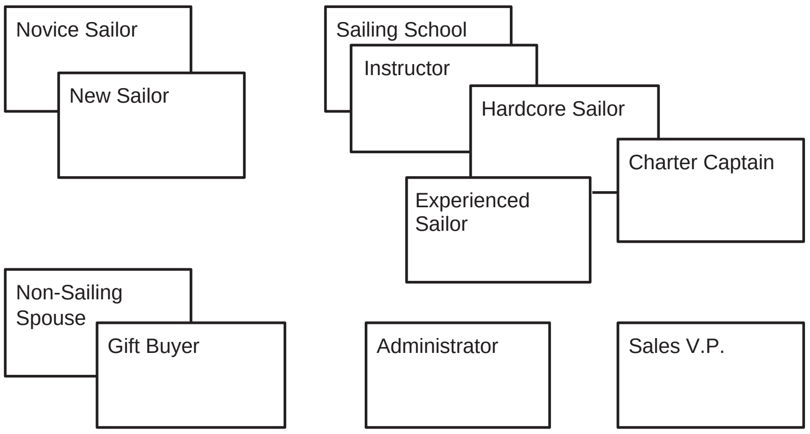


FIGURE 18.1 Positioning of the user role cards.

Consolidating and Narrowing

After the user role names are written on cards we need to remove duplicates or near duplicates, consider whether any roles should be merged, and come up with a refined list of user roles that the project will start with. The easiest way to start is by trying to remove any card that is placed entirely on top of another card, indicating that its author thought they were duplicates.

In this case, the New Sailor card has been placed on top of the Novice Sailor card. The authors of those cards explain what they intended by them and anyone else adds any remarks they wish. It turns out that there is a distinction between a Novice Sailor and a New Sailor. A New Sailor is someone who is new to the sport of sailing; perhaps she is currently taking lessons or has sailed a few times. The author of the Novice Sailor card actually meant that role to represent someone who may have been sailing for years but just not often enough to have become good at it. The group decides that although these roles appear to be slightly different they are not sufficiently different that it is worth having two roles for them. They are merged together in a single role, Novice Sailor, and the New Sailor card is torn up and thrown away.

Next, the group considers the overlapping Sailing School and Instructor cards. The author of the Instructor card explains that the role represents sailors who teach sailing classes. Instructors, she argues, frequently buy copies of books for their students or perhaps assemble lists of books that students are required to read. The author of the Sailing School role card indicates that this is partially what she intended for that card. However, she thought that typical use would be by an administrator at the school rather than by the sailing instructor himself. Lori the customer clears this up for us by telling us that even if it is an administrator, the administrator will have many of the same characteristics of an Instructor. Since an Instructor is more clearly a single person than is Sailing School the Sailing School card is torn up.

The Hardcore Sailor card has been postioned so that it partially overlaps Instructor, Experienced Sailor and even the Charter Captain role cards. The group discusses these roles next and learns that the Hardcore Sailor role was written to capture the type of sailor who typically knows exactly which books she wants. The Hardcore Sailor knows, for example, the name of the best book on navigation. Her search patterns will therefore be quite different from those of a less knowledgeable sailor, even an Experienced Sailor. The Experienced Sailor role represents people very familiar with the products the site will offer but who may not automatically recall the names of the best books.

After discussing Charter Captain the team decides that role is essentially the same as Hardcore Sailor and the card is torn up.

At this point, the team has decided to keep Novice Sailor, Instructor, Hardcore Sailor, and Experienced Sailor. They've disposed of New Sailor, Charter Captain, and Sailing School. And they have yet to consider Gift-Buyer, Non-Sailing Spouse, Administrator, and Sales Vice President. The authors of these remaining role cards explain their intent.

The Gift Buyer role represents someone who is not a sailor but who is buying a gift for someone who is. The author of the Non-Sailing Spouse role indicates that this was the intent behind that card. After some discussion about the two role cards the team decides to tear both up and replace them with a consolidated card for the Non-Sailing Gift Buyer role.

The author of the Administrator role explains that this role represents the work that will be done by one or more administrators who will need to load data into the system and otherwise keep the system running. This is the first role the team talks about that represents someone who is not buying items from the site. After discussing it, they decide that the role is important and Lori indicates that she will have some stories about how the system will be maintained and how new items will be added to the site's inventory.

The Sales Vice President role is discussed next. This is another non-purchasing role. However, the CEO has mandated that the new system be watched very closely to see how it is affecting sales. The team considers leaving the role out because they don't think there will be many stories specifically for the role. In the end they decide to include the role but to rename it to the more generic Report Viewer.

At this point the team has settled on the roles shown in Figure 18.2

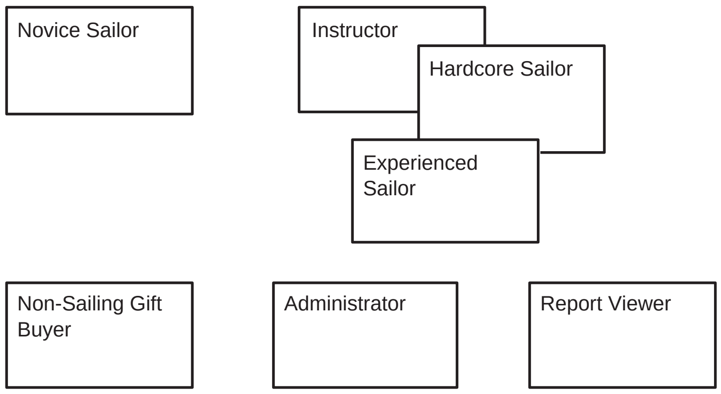


FIGURE 18.2 The roles after consolidation and initial discussion.

Role Modeling

Next, the team considers each role and adds detail to the role cards. The details will vary based on the domain and the type of software but good, general factors to consider include:

- The frequency with which the user will use the software
- The user's level of expertise with the domain
- The user's general level of proficiency with computers and software
- The user's level of proficiency with the software being developed
- The user's general goal for using the software. Some users are after convenience, others favor a rich experience, and so on

The group discusses these issues for each role card. They update the user role cards as follows:

Novice Sailor: Experienced web shopper. Expected to make 6 purchases during first 3 months of sailing. Is sometimes referred to a specific title; other times needs help selecting the right book. Wants more help in selecting appropriate books (good content written at the appropriate level) than she gets in a physical bookstore.

Instructor: Expected to use the website frequently, often once a week. Through the company's telephone sales group, an Instructor frequently places similar orders (e.g., 20 copies of the same book). Proficient with the website but usually somewhat nervous around computers. Interested in getting the best prices. Not interested in reviews or other "frills."

Hard Core Sailor: Generally inexperienced with computers. A large purchaser of items from the company's catalog but not a large purchaser of books. Buys lots of additional equipment from us. Usually knows exactly what he's looking for. Doesn't want to feel stupid while trying to use the site.

Experienced Sailor: Proficient with computers. Expected to order once or twice per quarter, perhaps more often during the summer. Knowledgeable about sailing but usually only for local regions. Very interested in what other sailors say are the best products and the best places to sail.

Non-Sailing Gift Buyer: Usually proficient with computers (or would not chose to purchase a gift online). Not a sailor and will have passing familiarity at best with sailing terms. Will usually be after a very specific book but may be looking for a book on a given topic.

Administrator: Highly proficient with computers. Has at least a passing familiarity with sailing. Accesses the backend of the system daily as part of her job. Interested in quickly learning the software but will want advanced user shortcuts later.

Report Viewer: Moderate proficiency with computers, especially business programs such as spreadsheets and word processors. Interested in highly detailed data on how the system is working, what visitors are buying or not buying, and how they are navigating or searching the site. Very willing to trade speed for power and depth.

The User Role Map

After the team has finished modeling each of the roles they draw the user role map, which is shown in Figure 18.3. A new role, Sailor, is introduced and used to show that Novice Sailor, Experienced Sailor, and Hardcore Sailor all share much in common. Since most Instructors will be Hardcore Sailors themselves a resembles arrow is drawn between those roles.

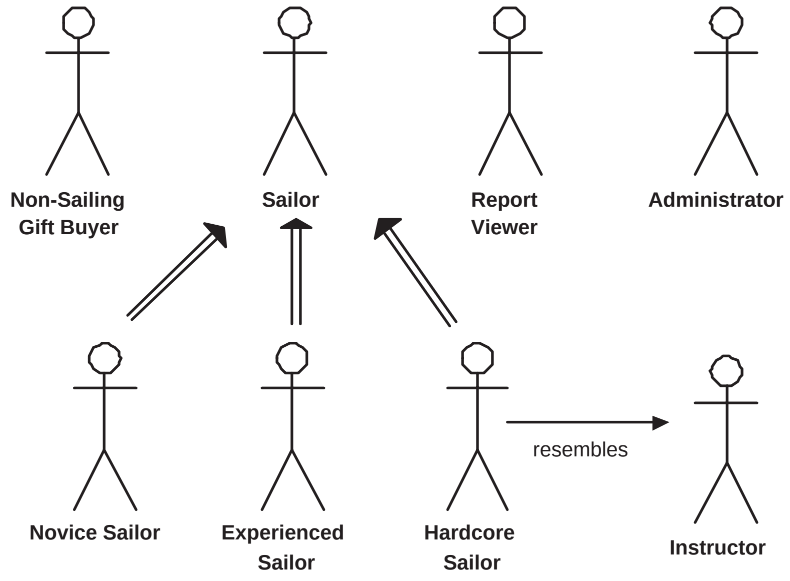


FIGURE 18.3 The User Role Map.

Adding a Persona

It is sometimes worth spending a few minutes adding a persona to the work done thus far. The team asks Lori which of these user roles must absolutely be satisfied with the new website in order for it to succeed. She says that Hardcore Sailors are important because of their longevity as customers. But, even though they sail frequently they are not large purchasers of books. On the other hand, the large population of Experienced Sailors is important and that group buys a significant number of books. Lori adds that perhaps the most important role to satisfy is the Instructor. Instructors can be responsible for the sale of hundreds of books throughout the year. In fact, she continues, with the new website she'd like to investigate ways of eventually offering financial incentives to Instructors who refer their students to the site.

With this information in mind the team decides to develop two personas. The first persona is Teresa. Teresa has been sailing for four years. She is the CEO of a publicly-traded biotech company and is completely comfortable ordering online. Teresa sails mainly during the summer so she'll only use the site during spring or summer, when preparing for a cruise. She's very busy and is interested in using our site to save time and to find books she hasn't seen before. Teresa is married to Tom, who doesn't sail himself but has accompanied Teresa on on two cruises through the Mediterranean.

The second persona is Captain Ron. Captain Ron has been sailing for 40 years and runs a sailing school out of San Diego. He retired from teaching high school five years ago and has been a sailing instructor ever since. He's been a loyal catalog customer for ten years. He's still a little intimidated by the computer in his office but he's intrigued enough by ordering on the web that we expect him to give it a try.

Chapter 19

The Stories

By considering each of the user roles the customer and developers generate a list of stories.

To generate the initial list of stories, the team decides to convene a story writing workshop where they will dedicate an hour or two to writing as many stories as they can. During a story writing workshop one approach is to just write stories in any order regardless of which role or persona may act in the story. An alternative approach is to start with a specific user role or persona and write all the stories the team can think of before moving onto the next role or persona. The results should be the same with either approach. In this case the team discusses it and chooses to work through each role and persona.

Stories for Teresa

The team decides to start with Teresa, a persona identified in the previous chapter, since the customer, Lori, has said that it is critical that the new website satisfy Teresa. The team knows Teresa is looking for speed and convenience. She's a true power user and will not mind a little extra complexity as long as the complexity helps her find what she's looking for more quickly. The first story they write is

> *Users can search for books by author, title, or ISBN number.*

The developers have some questions about this story. For example, can the user search by author, title and ISBN at the same time or does Lori want them to search by only one criterion at a time? They let these questions sit for now and focus on getting more preliminary stories written down.

Next, Lori says that after searching for a book a user can see detailed information about a book. She gives a few examples of the type of information she means and then writes

> *Users can view detailed information on a book. For example, number of pages, publication date, and a brief description.*

There's probably more she'll want besides these three details but the developers can ask her later when they're ready to code this story.

As a typical eCommerce site, the team knows that users will need a "shopping cart" and that users will buy the books in their shopping carts. So they write

> *Users can put book into a "shopping cart" and buy them when they're done shopping.*

Lori the customer also says that a user will need the chance to delete books from the shopping cart before the order is processed. This leads to this story:,

> A user can remove books from her cart before completing an order.

To actually process an order with a credit card the system will need to know the credit card to charge and some address information. This leads to a new story:

> To buy a book the user enters her address, the shipping address and credit card information.

Lori reminds the developers that since Teresa has only been sailing for four years she won't always know exactly what book she wants. For Teresa the site should include features for customers to rate and review items. This leads the developers to write

> Users can rate and review books.

Since Teresa wants to be able to order as quickly as possible, the team decides that the system needs to remember shipping and billing information. Some of the site's customers, for example the Non-Sailing Gift Buyer role, may not buy very often so these customers may not want to create a reusable account. Similarly, any extra steps the first time may scare off Captain Ron who is always a little tentative with new websites. So, the team decides that a user can buy books with or without an account and writes these two stories:

> A user can establish an account that remembers shipping and billing information.

> Users can edit their account information (credit card, shipping address, billing address, etc.)

The team also knows that Teresa wants to put items on a wish list of items she wants but is not ready to buy today. She'll either buy them for herself later or she'll tell her husband, Tom, and he'll buy items from her wish list. So, Lori writes these two stories:

> Users can put books into a "wish list" that is visible to other site visitors.

> A shopper can place an item from a wish list (even someone else's) into his or her shopping cart.

For that last story we want to make sure that whoever programs it knows that a user can select items from her own or someone else's wish list. We make sure to note that on the card (parenthetically in this case).

Because speed will be important to Teresa, Lori also identifies a performance constraint relating to how long it takes to order a book. She writes:

> Repeat customers must be able to find one book and complete an order in less than 90 seconds.
> (Constraint)

In this case Lori has chosen to focus on the time it takes a repeat customer to search for a book and complete her order. This is a good performance requirement because it captures all aspects of the user's experience with the site. Blazing fast database queries and middleware don't count for much if the user interface is so confusing to navigate that it takes a user three minutes to get to the search screen. This story reflects this better than would a story like "Searches must complete in two seconds." Naturally, Lori can add more performance constraints but it is usually sufficient to pick a few broad ones like this story.

Stories for Captain Ron

It becomes clear that the team is running out of stories for Teresa, the Experienced Sailor. So, they agree to switch focus to Captain Ron, who runs a sailing school and is a little more tentative with computers than was Teresa. When Captain Ron comes to the site he typically knows exactly what he's looking for. This leads Lori to write the following stories:

> A user can view a history of all of his past orders.

Users can easily re-purchase items when viewing past orders.

These stories will allow Captain Ron to look back at his old orders and repurchase items from those orders. However, Lori points out that Captain Ron may also want to purchase an item that he looked at recently, even if he hasn't previously purchased it. She writes:

The site always tells a shopper what the last 3 (?) items she viewed are and provides links back to them. (This works even between sessions.)

Stories for a Novice Sailor

Next the team moves on to consider the Novice Sailor role. The needs of a Novice Sailor largely overlap those of the Teresa and Captain Ron. But Lori decides it would be helpful if a Novice Sailor could se a list of our recommendations. Here the Novice Sailor could find the books we recommend on a variety of topics. She writes

Users can see what books we recommend on a variety of topics.

Stories for a Non-Sailing Gift Buyer

Switching to the Non-Sailing Gift Buyer role, the team discusses how it will need to be easy for a shopper to find the wish list of another person. They start to discuss various design solutions and what fields will be used for searching until they remember that design discussions should be saved for later. Instead of designing the feature in this meeting they write:

Users, especially Non-Sailing Gift Buyers, can easily find the wish lists of other users.

Lori also knows that the system needs to support gift cards and wrapping. She writes these two stories:

Users can choose to have items gift wrapped.

Users can choose to enclose a gift card and can write their own message for the card.

Stories for a Report Viewer

Lori says that the system will need to generate reports on purchase and traffic patterns and so on. She hasn't yet thought about the reports in detail so the developers write a simple placeholding story that will remind them that there are reports to develop. They'll decide on the report contents later. For now they write:

A Report Viewer can see reports of daily purchases broken down by book category, traffic, best– and worst–selling books, and so on.

Thinking about reports reminds Lori that the reports are highly sensitive. Naturally, they won't be available from the main site that consumers see. But, she says that only certain people within the company can have access to the reports. This could mean that if you can access one report you can access them all, or it could mean some users can access only some reports. The developers don't ask Lori about that now and instead write:

Only properly authenticated users can view reports.

To make reports meaningful Lori says that the database used by the website must be the same database used by our current telephone-based system. This leads to this constraint:

Orders made on the website have to end up in the same order database as telephone orders.
(Constraint)

Some Administration Stories

At this point attention shifts to the Administrator user role. The team thinks instantly of these two stories:

An administrator can add new books to the site.

An administrator needs to approve or reject reviews before they're available on the site.

The story about adding new books, reminds them that administrators need to delete books and also edit books in case incorrect information was used when the book was added. So they write:

An administrator can delete a book.

An administrator can edit the information about an existing book.

Wrapping Up

The developers can tell Lori is starting to run out of stories. So far each story has come instantly to mind but now she's having to think about whether there are any others. Because the project will be done using an incremental and iterative development process it's not important that she think of every story right up front. But because she wants a preliminary estimate of how long the system will take to build the team does want to think of as much as they can without spending an inordinate amount of time. If Lori comes up with a new story once we've started she'll have the opportunity to move it into the release if she moves out the same approximate amount of work.

The developers ask Lori if there are any other stories she feels have been forgotten thus far. She writes this story:

> A user can check the status of her recent orders. If an order hasn't shipped, she can add or remove books, change the shipping method, the delivery address, and the credit card.

Lori also reminds the developers that scalability needs are not tremendous but that the site does need to handle at least 50 concurrent users. They write this as a constraint story:

> The system must support peak usage of up to 50 concurrent users.
> (Constraint)

Chapter 20

Estimating the Stories

After the stories have been written each is estimated by the developers.

The story writing workshop resulted in 27 stoires, which are summarized in Table 20.1. The next goal is to create a release plan that will show Lori the customer what the developers expect to accomplish and whether the site can be operation within the boss's thirty day deadline. Because there is probably more work than can be completed in those thirty days the developer will need to work closely with Lori to prioritize stories.

TABLE 20.1 The initial collection of stories.

Number	Story
1	Users can search for books by author, title, or ISBN number.
2	Users can view detailed information on a book. For example, number of pages, publication date, and a brief description.
3	Users can put book into a "shopping cart" and buy them when they're done shopping.
4	A user can remove books from her cart before completing an order.

TABLE 20.1 The initial collection of stories.

Number	Story
5	To buy a book the user enters her address, the shipping address and credit card information.
6	Users can rate and review books.
7	A user can establish an account that remembers shipping and billing information.
8	Users can edit their account information (credit card, shipping address, billing address, etc.)
9	Users can put books into a "wish list" that is visible to other site visitors.
10	A shopper can place an item from a wish list (even someone else's) into his or her shopping cart.
11	Repeat customers must be able to find one book and complete an order in less than 90 seconds.
12	A user can view a history of all of his past orders.
13	Users can easily re-purchase items when viewing past orders.
14	The site always tells a shopper what the last 3 (?) items she viewed are and provides links back to them. (This works even between sessions.)
15	Users can see what books we recommend on a variety of topics.
16	Users, especially Non-Sailing Gift Buyers, can easily find the wish lists of other users.
17	Users can choose to have items gift wrapped.
18	Users can choose to enclose a gift card and can write their own message for the card.
19	A Report Viewer can see reports of daily purchases broken down by book category, traffic, best– and worst–selling books, and so on.
20	Only properly authenticated users can view reports.
21	Orders made on the website have to end up in the same order database as telephone orders.
22	An administrator can add new books to the site.
23	An administrator needs to approve or reject reviews before they're available on the site.
24	An administrator can delete a book.

TABLE 20.1 The initial collection of stories.

Number	Story
25	An administrator can edit the information about an existing book.
26	A user can check the status of her recent orders. If an order hasn't shipped, she can add or remove books, change the shipping method, the delivery address, and the credit card.
27	The system must support peak usage of up to 50 concurrent users.

In order to create the release plan an estimate is needed for each story. As we learned in Chapter 8, "Principles of Estimating," the developers are going to estimate each story in story points that represent ideal time expended by an experienced senior programmer. They will give estimates that they are 50% and 90% confident in. The 50% estimate will be an estimate they feel is aggressive but that can be met about half the time. The 90% estimate is the amount of time they think it will take if things don't go smoothly and if Lori answers any as–yet–unanswered questions with the more difficult to implement choice.

The First Story

It isn't necessary to start with User Story 1 ("Users can search for books by author, title, or ISBN number.") but in this case the first story is a good one to start estimating with. When Lori first wrote this story the developers weren't sure if Lori meant that a user could search for all of these fields at the same time or whether a user could search only one at a time. Since Lori's answer could have a big impact on the estimate it's worth asking her.

Naturally Lori says she wants both. She wants a basic search mode where the value in one field searches both author and title. She then wants an advanced search screen where any or all of these fields can be used in combination. Even with both search modes the story isn't that big; but, there's such an easy division between the modes that the developers tear up Story 1 and replace it with Stories 1a and 1b.

Story 1a: Users can do a basic simple search that searches for a word or phrase in both the author and title fields.

Story 1b: Users can search for books by entering values in any combination of author, title, and ISBN.

The developers have chosen to use a wideband Delphi estimation approach and to prepare the estimates there are three programmers available—Rafe, Jay, and Maria. Each programmer takes a couple of index cards that they'll use to write down estimates. They talk about Story 1a, clarify a few details on it, and then each programmer writes his or her 50% estimate on an index card. When everyone is done, each programmer holds his or her card up so everyone can see it. They've written:

Rafe: 1 day
Jay: ½ day
Maria: 2 days

These three developers discuss their estimates. Maria explains why she thinks this will take two ideal days by an experienced senior programmer. She talks about how they'll need to select a search engine, incorporate it, and only then be able to write the screens to fulfill the story. Jay says that he's already familiar with all likely searching options and is pretty confident about the direction they should go, which is why his estimate is so much lower.

Everyone is asked to write down a new estimate. When they're down they again show their cards. This time the cards say:

Rafe: 1 day
Jay: 1 day
Maria: 1 day

That was pretty easy. Jay decided to move his estimate up and Maria was convinced that they could do Story 1a faster than she originally thought. They now have a one ideal day (or one story point) estimate to use as the 50% estimate. What about an estimate that the developers feel 90% confident in? The same three developers talk about it and decide that even if they have problems or make a mistake in choosing a search engine this story can't take more than the two days Maria thought initially. They start writing estimates down as shown in Table 20.2.

TABLE 20.2 Starting to write down the estimates.

Number	Story	50% Estimate	90% Estimate
1a	Users can do a basic simple search that searches for a word or phrase in both the author and title fields.	1	2

Advanced Search

On to story 1b, the advanced search. The programmers again write their estimates on index cards and turn them over at the same time showing:

Rafe: 2 days

Jay: 1 days

Maria: 2 days

Rafesays the advanced search will take a little longer than the basic search because there's more to search on. Jay agrees but says that since the basic search will have already been coded it won't take long to add the advanced search features. However, Maria points out that the stories are independent and we don't know that Story 1a will be done before Story 1b. Lori the customer could decide that the advanced search is more critical and she may choose to have it done first.

After another round or two of writing estimates on cards everyone agrees that while there is a bit more work on the advanced search than the basic search it isn't much and they should again use estimates of 1 day (50%) and 2 days (90%).

The next few stories are pretty straightforward to estimate and none need to be split. The developers arrive at the estimates shown in Table 20.3.

TABLE 20.3 Building up the list of estimates.

Number	Story	50% Estimate	90% Estimate
1a	Users can do a basic simple search that searches for a word or phrase in both the author and title fields.	1	2
1b	Users can search for books by entering values in any combination of author, title, and ISBN.	1	2
2	Users can view detailed information on a book. For example, number of pages, publication date, and a brief description.	1	3
3	Users can put books into a "shopping cart" and buy them when they're done shopping.	1	1
4	A user can remove books from his cart before completing an order.	1/2	1
5	To buy a book the user enters his address, the shipping address and credit card info.	2	5

Rating and reviewing

Story 6 ("Users can rate and review books.") is a bit harder. Before writing down estimates and showing them to each other the developers talk about this story. The rating part doesn't seem hard but the reviews seem more complicated. They'll need a screen for users to enter a review and maybe to preview it. Will reviews just be plain text or can the reviewer type in HTML? Can users only review books they bought from us?

Because reviews are so much more involved than just rating books we decide to split the story and create Stories 6a and 6b.

Story 6a: Users can rate books from 1 (bad) to 5 (good). The book does not have to be one the user bought from us.

Story 6b: Users can write reviews of books. They can preview the review before submitting it. The book does not have to be one the user bought from us.

They estimate story 6a as between one day (50%) and three days (90%). Story 6b is estimated as three days (50%) and five days (90%).

While they're thinking about rating and reviewing books, there is also story 23 ("An administrator needs to approve or reject reviews before they're available on the site.") to consider. This could be really simple or it could be more involved and require the administrator to identify the reason she rejects a review or possibly email the reviewer. We don't think Lori will want anything complicated and our discussion leads to estimates of one day (50%) and two days (90%).

Accounts

Story 7 ("A user can establish an account that remembers shipping and billing information.") seems straightforward and the developers estimate it as between two and four days.

Next, they start to estimate story 8: "Users can edit their account information (credit card, shipping address, billing address, etc.)." This story is not very large but it is easily split. Splitting a story like this one is frequently a good idea because it allows more flexibility during release planning and it allows the customer to prioritize work at a much finer level. In our case, for example, Lori may think it is critical for users to edit their credit cards but she may be willing to wait a few iterations for the ability for users to change addresses. The developers split story 8 and come up with stories 8a and 8b:

Story 8a: A user can edit the credit card information stored in her account.

Story 8b: A user can edit the shipping and billing addresses stored in her account.

Neither of these stories seems difficult so our Delphi estimation process results in estimates of one-half day (50%) to a full day (90%) for each.

Finishing the Estimates

This same process is repeated for each of the remaining stories. Only a few of the remaining stories are worth specific mention. First, story 16 ("Users, especially Non-Sailing Gift Buyers, can easily find the wish lists of other users.") is vague. When asked about her intentions regarding how users could search for a wish list, Lori provides enough detail that the developers rewrite the story as

Story 16: Users, especially Non-Sailing, Gift Buyers, can search for a wish list based on its owner's name and state.

Next, story 26 is split into two stories: one for checking the status of recent orders and a second for changing orders that haven't yet shipped. The new stories are

Story 26a: A user can check the status of her recent orders.

Story 26b: If an order hasn't shipped, a user can add or remove books, change the shipping method, the delivery address, and the credit card.

Finally, stories 11 ("Repeat customers must be able to find one book and complete an order in less than 90 seconds.") and 21 ("Orders made on the website have to end up in the same order database as telephone orders.") are constraints. As constraints they influence other stories but do not require any specific coding themselves.

All The Estimates

Table 20.4 shows all of the estimates.

TABLE 20.4 The complete list of stories and estimates.

Number	Story	50% Estimate	90% Estimate
1a	Users can do a basic simple search that searches for a word or phrase in both the author and title fields.	1	2
1b	Users can search for books by entering values in any combination of author, title, and ISBN.	1	2
2	Users can view detailed information on a book. For example, number of pages, publication date, and a brief description.	1	3
3	Users can put books into a "shopping cart" and buy them when they're done shopping.	1	1
4	A user can remove books from his cart before completing an order.	1/2	1
5	To buy a book the user enters his address, the shipping address and credit card info.	2	5
6a	Users can rate books from 1 (bad) to 5 (good). The book does not have to be one the user bought from us.	1	3
6b	Users can write reviews of books. She can preview the review before submitting it. The book does not have to be one the user bought from us.	3	5
7	A user can establish an account that remembers shipping and billing information.	1	2
8a	A user can edit the credit card information stored in her account.	1/2	1
8b	A user can edit the shipping and billing addresses stored in her account.	1/2	1

TABLE 20.4 The complete list of stories and estimates.

Number	Story	50% Estimate	90% Estimate
9	Users can put books into a "wish list" that is visible to other site visitors.	1	2
10	A shopper can place an item from a wish list (even someone else's) into his or her shopping cart.	1/2	1
11	Repeat customers must be able to find one book and complete an order in less than 90 seconds.	0	0
12	A user can view a history of all of his past orders.	1	1
13	Users can easily re-purchase items when viewing past orders.	1/2	1/2
14	The site always tells a shopper what the last 3 (?) items she viewed are and provides links back to them. (This works even between sessions.)	1/2	2
15	Users can see what books we recommend on a variety of topics.	2	7
16	Users, especially Non-Sailing, Gift Buyers, can search for a wish list based on its owner's name and state.	1	2
17	Users can choose to have items gift wrapped.	1/2	1/2
18	Users can choose to enclose a gift card and can write their own message for the card.	1/2	1/2
19	A Report Viewer can see reports of daily purchases broken down by book category, traffic, best- and worst-selling books, and so on.	5	10
20	Only properly authenticated users can view reports.	1/2	2

TABLE 20.4 The complete list of stories and estimates.

Number	Story	50% Estimate	90% Estimate
21	Orders made on the website have to end up in the same order database as telephone orders.	0	0
22	An administrator can add new books to the site.	1/2	2
23	An administrator needs to approve or reject reviews before they're available on the site.	1	2
24	An administrator can delete a book.	1/2	1
25	An administrator can edit the information about an existing book.	1/2	2
26a	A user can check the status of her recent orders.	1/2	1
26b	If an order hasn't shipped, a user can add or remove books, change the shipping method, the delivery address, and the credit card.	1	2
27	The system must support peak usage of up to 50 concurrent users.	2	5

Chapter 21

The Release Plan

The Release Plan describes the stories the developers believe they can complete in the timeframe desired by the customer.

The customer prioritizes the stories. The main factor in determining the priority of a story is the value it will deliver to the business. However, the customer should also consider the estimate for the story. Occasionally, highly desired stories become less desirable when their cost (the estimates) are considered.

So, in this example, Lori the customer prioritizes the stories. Since the stories are written on notecards she sorts them into three piles based on their importance to the targeted go-live date in thirty days: Must Have, Nice to Have, and Can Wait. Within each pile Lori further prioritizes the stories into the order she'd like to see them implemented. Her Must Have stories are shown in Table 21.1

TABLE 21.1 The must-have stories for the initial release in thirty days.

Number	Story	50% Estimate	90% Estimate
1a	Users can do a basic simple search that searches for a word or phrase in both the author and title fields.	1	2
3	Users can put books into a "shopping cart" and buy them when they're done shopping.	1	1
5	To buy a book the user enters his address, the shipping address and credit card info.	2	5
4	A user can remove books from his cart before completing an order.	1/2	1
7	A user can establish an account that remembers shipping and billing information.	1	2
21	Orders made on the website have to end up in the same order database as telephone orders.	0	0
22	An administrator can add new books to the site.	1/2	2
24	An administrator can delete a book.	1/2	1
25	An administrator can edit the information about an existing book.	1/2	2

Estimating the Total Effort

Once the stories are prioritized the developers look at the estimates for the must–have stories and calculate a buffered estimate. Looking at the must-have stories in Table 21.1 the developers create Table 21.2.

As you'll recall from Chapter 11, "Planning a Release," the best way to size the project buffer is to square the difference between the 90% and 50% estimate for each story and then sum those values and take the square root of that total. The final column of Table 21.2

TABLE 21.2 Calculating a buffered schedule for the must-have stories only.

Story	50% Estimate	90% Estimate	$(90\%—50\%)^2$
1a	1	2	1
3	1	1	0
5	2	5	9
4	0.5	1	0.25
7	1	2	1
21	0	0	0
22	0.5	2	2.25
24	0.5	1	0.25
25	0.5	2	2.25
Sum	**7**	**16**	**16**

shows the squared differences and then the sum of those squares in the final row. From this table the team sees that the project will take seven ideal days if all stories are completed within the optimistic 50% estimates. Alternatively, the project could take sixteen ideal days if all stories take as long as their 90% estimates. To create a buffered schedule the team adds the sum of the 50% estimates to the square root of the sum of the squared differences (the sixteen in the final column). This gives them a buffered schedule of 7+4=11 ideal days (or story points).

Estimating Velocity

To determine if they can complete eleven ideal days worth of work in the thirty days they have, the team must estimate their velocity over those first thirty days. Since this project is the first of its kind for the company the developers cannot rely on previous velocity measures. Since the schedule is so short for this project the developers and Lori the customer choose an iteration length of one week. There will be two developers assigned to the project, Maria and Rafe. Maria and Rafe estimate that for each calendar day they will each spend about one half of an ideal day on this project. Each has additional obligations in the company that cannot be put entirely on hold. They estimate their productivity to be as shown in Table 21.3.

TABLE 21.3 Productiivity of the two programmers in ideal days per calendar day.

Developer	Iteration 1	Iteration 2	Iteration 3	Thereafter
Maria	0.5	0.5	0.5	0.6
Rafe	0.4	0.4	0.4	0.5
Total	**0.9**	**0.9**	**0.9**	**1.1**

Table 21.3 means that the combination of Maria and Rafe will complete about 0.9 ideal days of work for each calendar day they have available. With an iteration length of one week (five working days) this means they will complete 0.9x5=4.5 story points per week. By the fourth week each expects to become a little more productive and collectively they will complete 1.1x5=5.5 story points that week.

So, in the four weeks they have available for the project they estimate they can complete 4.5+4.5+4.5+5.5=19 story points.

This is actually great news because the buffered schedule says they need to complete eleven story points to deliver the bare minimum amount of functionality to Lori.

Adding More Work

Since the developers seem well able to complete the minimum amount of work Lori requested, they ask her which additional stories she'd like to include. Lori pulls the stories shown in Table 21.4 from her pile of Nice To Have stories.

To determine whether they are likely to complete these additional stories, the developers recalculate the buffer as shown in Table 21.5. The first line of this table is the sum line from Table 21.2. The next four lines are the new stories Lori the customer selected. From

TABLE 21.4 The stories it would be nice to have in thirty days.

Number	Story	50% Estimate	90% Estimate
1b	Users can search for books by entering values in any combination of author, title, and ISBN.	1	2
8a	A user can edit the credit card information stored in her account.	1/2	1
8b	A user can edit the shipping and billing addresses stored in her account.	1/2	1
15	Users can see what books we recommend on a variety of topics.	2	7

Table 21.5 the developers calculate a buffer size of 6.5 days (the square root of 42.5). They round that up to seven days and add it to the eleven day sum of the 50% estimates. This results in a buffered estimate of 18 ideal days (or story points). They'd previously estimated they could complete 19 story points during the four weeks. They should have just enough time to complete the additional stories.

TABLE 21.5 Adding some nice-to-have stories to the release plan.

Story	50% Estimate	90% Estimate	$(90\%—50\%)^2$
All Must-Have Stories	7	16	16
1b	1	2	1
8a	0.5	1	0.25
8b	0.5	1	0.25
15	2	7	25
Sum	**11**	**27**	**42.5**

At this point the developers and Lori could choose to add another tiny story to the release plan or they could consider themselves done since the amount of work planned and the estimated velocity are so close. Either approach will likely result in delivery of the same end product. The team will need to measure and monitor their velocity as they complete the initial iterations. They should then adjust the release (either to possibly include more or less) at the completion of those iterations.

Chapter 22

The Acceptance Tests

Writing the acceptance tests before the programmers start coding helps identify unstated assumptions and missing stories.

The acceptance tests for a story are used to determine whether the story is completed to the point where the customer can accept that part of the software as complete. This means that ideally the customer writes the acceptance tests; after all, only the customer can define when something is acceptable to her.

On this project, however, Lori (like many customers on other projects) is a little uncomfortable having to think through all of the tests that define whether a story is fully implemented or not. She enlists the help of the developers in identifying her acceptance tests. An additional benefit of this is that, beyond generating a list of acceptance tests, it leads to further conversations between Lori and the developers, Rafe and Maria.

The Search Tests

Stories 1a and 1b cover search features that Lori prioritized into the first release.

Story 1a: Users can do a basic simple search that searches for a word or phrase in both the author and title fields.

Story 1b: Users can search for books by entering values in any combination of author, title, and ISBN.

The tests for Story 1a are as follows:

- Search for a word that is known to be part of a title but unlikely to be an author; e.g., "navigation"
- Search for a word that is likely to be an author but unlikely to be a title; e.g., "John"
- Search for a word that is unlikely to be in either; e.g., "wookie"

The tests for Story 1b are as follows:

- Use values for author and title that match at least one book.
- Use values for author and title that match no books.
- Try an ISBN search.

Shopping Cart Tests

Stories 3 and 4 cover the use of the shopping cart.

Story 3: Users can put books into a "shopping cart" and buy them when they're done shopping.

Story 4: A user can remove books from his cart before completing an order.

Lori and the developers talk about these stories and realize they have a few open questions: Can users put out–of–stock books into shopping carts? What about books that are not yet in print? Additionally, the team realizes that story 4 covers changing the quantity of an item to 0 there is no explicit story for increasing the quantity of an item. They could write this as a separate story but decide instead to rephrase Story 4:

Story 4: A user can adjust the quantity of any item in her cart. Setting the quantity to 0 removes the item from the cart.

The tests for Story 3 are

- Put an out–of–stock book into the cart. Verify that the user is told that the book will be shipped when it becomes available.
- Put a book that hasn't been published yet into the cart. Verify that the user is told that the book will ship when available.
- Put a book that is in stock into the cart.
- Put a second copy of a book into the cart. Verify that the count goes up.

The tests for Story 4 are

- Change the quantity of a book from one to ten.
- Change the quantity from ten to one.
- Remove a book by changing the quantity to zero.

Buying Books

Story 5 covers the actual purchasing of books. In discussing this story, the developers clarify a few aspects of it with Lori the customer. Lori wants users to be able to enter separate shipping and billing addresses or indicate that the addresses are the same. The site will accept only Visa and MasterCard.

Story 5: *To buy a book the user enters his address, the shipping address and credit card info.*

The tests for Story 5 are:

- Enter a billing address and indicate that the shipping address is the same.
- Enter separate billing and shipping addresses.
- Test with a valid Visa card.
- Test with a valid MasterCard
- Test with a valid American Express card (fail)
- Test with an expired Visa
- Test with a MasterCard that is over its credit limit
- Test with a Visa that is missing digits

User Accounts

Story 7 covers the creation of user accounts. Tests for Story 7 are:

Story 7: A user can establish an account that remembers shipping and billing information.

- Users can order without creating an account.
- Create an account then recall it and see if the information has been saved.

Stories 8a and 8b allow users to modify the information stored in their accounts.

Story 8a: A user can edit the credit card information stored in her account.

Story 8b: A user can edit the shipping and billing addresses stored in her account.

Tests for Story 8a are

- Edit the card number to make it an invalid number. Verify that the user is warned.
- Edit the expiration date to one in the past. Verify that the change isn't saved.
- Change the credit card number to a new valid number and make sure the change is saved.

- Change the expiration date to a date in the future and make sure the change is saved.

Test for Story 8b are

- Change various parts of the shipping address and verify that the changes are saved.
- Change various parts of the billing address and verify that the changes are saved.

Administration

Stories 22, 24 and 25 cover the administration needs of the site.

Story 22: *An administrator can add new books to the site.*

Tests for Story 22 are

- Test that an administrator can add a book to the site.
- Test that a non-administrator cannot add a book.
- Test that a book can only be added if required data is present.

Story 24: *An administrator can delete a book.*

Test for Story 24 are

- Verify that an administrator can delete a book.
- Verify that a non-administrator cannot delete a book.

- Delete a book and then verify that outstanding orders for the book will still ship.

Story 25: *An administrator can edit the information about an existing book.*

When the developers and Lori discuss Story 25 they discuss how to handle unshipped orders that include a book for which the price changes. This becomes one of the tests for Story 25

- Verify that items like name, author, number of pages, etc. can be changed.
- Verify that price can be changed but that price changes do not affect previously placed (but unbilled and unshipped orders).

Additional Stories

There are only two stories remaining that need tests written. The first is Story 15.

Story 15: *Users can see what books we recommend on a variety of topics.*

Lori and the developers talk about Story 15 and decide it will be a simple static page that includes a list of recommendations for a variety of topics. They write these tests:

- Select a topic (for exampe, navigation or cruising) and view the recommendations for that topic. Make sure they make sense.

✧ Click on an item in the list to verify that the browser goes to a page of information on that book.

Next, Story 25 is a constraint story that specifies the database to be used by this application. The only test for it is to examine the database and verify that an order submitted from the website gets stored in the database:

Story 25: *Orders made on the website have to end up in the same order database as telephone orders.*

✧ Place an order. Open up the telephone order entry database and verify that the order was stored in that database.