Chapter 1

An Overview

Software requirements is a communication problem. Those who want the new software (either to use or to sell) must communicate with those who will build the new software. To succeed, a project relies on information from the heads of very different people: on one side are customers and users and sometimes analysts, domain experts and others who view the software from a business or organizational perspective; on the other side is the technical team.

If either side dominates these communications, the project loses. When the business side dominates, it mandates functionality and dates with little concern that the developers can meet both objectives, or whether the developers understand exactly what is needed. When the developers dominate the communications, technical jargon replaces the language of the business and the developers lose the opportunity to learn what is needed by listening.

What we need is a way to work together so that neither side dominates and so that the emotionally-fraught and political issue of resource allocation becomes a shared problem. Projects fail when the problem of resource allocation falls entirely on one side. If the developers shoulder the problem (usually in the form of being told "I don't care how you do it but do it all by June") they may trade quality for additional features, may only partially implement a feature, or may solely make any of a number of decisions in which the customers and users should participate. When customers and users shoulder the burden of resource allocation, we usually see a lengthy series of discussions at the start of a project during which features are progressively removed from the project. Then, when the software is eventually delivered, it has even less functionality than the reduced set that was identified.

By now we've learned that we cannot perfectly predict a software development project. As users see early versions of the software, they come up with new ideas and their opinions change. Because of the intangibility of software, most developers have a notoriously difficult time estimating how long things will take. Because of these and other factors we cannot lay out a perfect PERT chart showing everything that must be done on a project.

From the Library of www.wowebook.com

So, what do we do?

We make decisions based on the information we have at hand. And we do it often. Rather than making one all-encompassing set of decisions at the outset of a project, we spread the decision-making across the duration of the project. To do this we make sure we have a process that gets us information as early and often as possible. And this is where user stories come in.

What Is a User Story?

A user story describes functionality that will be valuable to either a user or purchaser of a system or software. User stories are composed of three aspects:

- a written description of the story used for planning and as a reminder
- conversations about the story that serve to flesh out the details of the story
- tests that convey and document details and that can be used to determine when a story is complete

Because user story descriptions are traditionally hand-written on paper note cards, Ron Jeffries has named these three aspects with the wonderful alliteration of Card, Conversation, and Confirmation (Jeffries 2001). The Card may be the most visible manifestation of a user story, but it is not the most important. Rachel Davies (2001) has said that cards "*represent* customer requirements rather than *document* them." This is the perfect way to think about user stories: While the card may contain the text of the story, the details are worked out in the Conversation and recorded in the Confirmation.

As an example user story see Story Card 1.1, which is a story card from the hypothetical BigMoneyJobs job posting and search website.

> A user can post her resume to the website.

Story Card 1.1 An initial user story written on a note card.

For consistency, many of the examples throughout the rest of this book will be for the BigMoneyJobs website. Other sample stories for BigMoneyJobs might include:

From the Library of www.wowebook.com

- A user can search for jobs.
- A company can post new job openings.
- A user can limit who can see her resume.

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

- 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 for BigMoneyJobs because its users would not care which programming language was used. However, if this were an application programming interface, then the user of that system (herself a programmer) could very well have written that "the software will be written in C++."

The second story is not a good user story in this case because the users of this system do not care about the technical details of how the application connects to the database.

Perhaps you've read these stories and are screaming "But wait— using a connection pool is a requirement in my system!" If so, hold on, the key is that stories should be written so that the customer can value them. There are ways to express stories like these in ways that are valuable to a customer. We'll see examples of doing that in Chapter 2, "Writing Stories."

Where Are the Details?

It's one thing to say "A user can search for jobs." It's another thing to be able to start coding and testing with only that as guidance. Where are the details? What about all the unanswered questions like:

- What values can users search on? State? City? Job title? Keywords?
- Does the user have to be a member of the site?
- Can search parameters be saved?
- What information is displayed for matching jobs?

Many of these details can be expressed as additional stories. In fact, it is better to have more stories than to have stories that are too large. For example, the entire BigMoneyJobs site is probably described by these two stories:

From the Library of www.wowebook.com

- A user can search for a job.
- A company can post job openings.

Clearly these two stories are too large to be of much use. Chapter 2, "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 between half a day and perhaps two weeks by one or a pair of programmers. Liberally interpreted, the two stories above could easily cover the majority of the BigMoneyJobs site so each will likely 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 "A user can search for a job" could be split into these stories:

- A user can search for jobs by attributes like location, salary range, job title, company name, and the date the job was posted.
- A user can view information about each job that is matched by a search.
- A user can view detailed information about a company that has posted a job.

However, we do not continue splitting stories until we have a story that covers every last detail. For example, the story "A user can view information about each job that is matched by a search" is a very reasonable and realistic story. We do not need to further divide it into:

- A user can view a job description.
- A user can view a job's salary range.
- A user can view the location of a job.

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

4.6) A user can view information about each job that is matched by a search.
  4.6.1) A user can view the job description.
  4.6.2) A user can view a job's salary range.
  4.6.3) A user can view the location of a job.

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

From the Library of www.wowebook.com

a discussion, as shown in Story Card 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. As we'll see, agreements are documented by tests that demonstrate that a story has been developed correctly.

> Users can view information about each job that is matched by a search.
>
> Marco says show description, salary, and location.

Story Card 1.2 A story card with a note.

"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 capture these expectations there. The expectations are written as reminders about how to test the story as shown in Story Card 1.3. If you're using an electronic system it probably has a place you can enter the acceptance test reminders.

> Try it with an empty job description.
> Try it with a really long job description.
> Try it with a missing salary.
> Try it with a six-digit salary.

Story Card 1.3 The back of a story card holds reminders about how to test the story.

From the Library of www.wowebook.com

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.

The Customer Team

On an ideal project we would have a single person who prioritizes work for developers, omnisciently answers their questions, will use the software when it's finished, and writes all of the stories. This is almost always too much to hope for, so we establish a customer team. The customer team includes those who ensure that the software will meet the needs of its intended users. This means the customer team may include testers, a product manager, real users, and interaction designers.

What Will the Process Be Like?

A project that is using stories will have a different feel and rhythm than you may be used to. Using a traditional waterfall-oriented process leads to a cycle of write all the requirements, analyze the requirements, design a solution, code the solution, and then finally test it. Very often during this type of process, customers and users are involved at the beginning to write requirements and at the end to accept the software, but user and customer involvement may almost entirely disappear between requirements and acceptance. By now, we've learned that this doesn't work.

The first thing you'll notice on a story-driven project is that customers and users remain involved throughout the duration of the project. They are not expected (or allowed!) to disappear during the middle of the project. This is true whether the team will be using Extreme Programming (XP; see Appendix A, "An Overview of Extreme Programming," for more information), an agile version of the Unified Process, an agile process like Scrum (see Chapter 15, "Using Stories with Scrum"), or a home-grown, story-driven agile process.

The customers and intended users of the new software should plan on taking a very active role in writing the user stories, especially if using XP. The story writing process is best started by considering the types of users of the intended

From the Library of www.wowebook.com

system. For example, if you are building a travel reservation website, you may have user types such as frequent fliers, vacation planners, and so on. The customer team should include representatives of as many of these user types as practical. But when it can't, user role modeling can help. (For more on this topic see Chapter 3, "User Role Modeling.")

Why Does the Customer Write the Stories?

The customer team, rather than the developers, writes the user stories for two primary reasons. First, each story must be written in the language of the business, not in technical jargon, so that the customer team can prioritize the stories for inclusion into iterations and releases. Second, as the primary product visionaries, the customer team is in the best position to describe the behavior of the product.

A project's initial stories are often written in a story writing workshop, but stories can be written at any time throughout the project. During the story writing workshop, everyone brainstorms as many stories as possible. Armed with a starting set of stories, the developers estimate the size of each.

Collaboratively, the customer team and developers select an iteration length, from perhaps one to four weeks. The same iteration length will be used for the duration of the project. By the end of each iteration the developers will be responsible for delivering fully usable code for some subset of the application. The customer team remains highly involved during the iteration, talking with the developers about the stories being developed during that iteration. During the iteration the customer team also specifies tests and works with the developers to automate and run tests. Additionally, the customer team makes sure the project is constantly moving toward delivery of the desired product.

Once an iteration length has been selected, the developers will estimate how much work they'll be able to do per iteration. We call this *velocity.* The team's first estimate of velocity will be wrong because there's no way to know velocity in advance. However, we can use the initial estimate to make a rough sketch, or release plan, of what work will happen in each iteration and how many iterations will be needed.

To plan a release, we sort stories into various piles with each pile representing an iteration. Each pile will contain some number of stories, the estimates for which add up to no more than the estimated velocity. The highest-priority stories go into the first pile. When that pile is full, the next highest-priority stories go into a second pile (representing the second iteration). This continues until

From the Library of www.wowebook.com

you've either made so many piles that you're out of time for the project or until the piles represent a desirable new release of the product. (For more on these topics see Chapter 9, "Planning a Release," and Chapter 10, "Planning an Iteration.")

Prior to the start of each iteration the customer team can make mid-course corrections to the plan. As iterations are completed, we learn the development team's actual velocity and can work with it instead of the estimated velocity. This means that each pile of stories may need to be adjusted by adding or removing stories. Also, some stories will turn out to be far easier than anticipated, which means the team will sometimes want to be given an additional story to do in that iteration. But some stories will be harder than anticipated, which means that some work will need to be moved into later iterations or out of the release altogether.

Planning Releases and Iterations

A release is made up of one or more iterations. Release planning refers to determining a balance between a projected timeline and a desired set of functionality. Iteration planning refers to selecting stories for inclusion in this iteration. The customer team and the developers are both involved in release and iteration planning.

To plan a release, the customer team starts by prioritizing the stories. While prioritizing they will want to consider:

- 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.

The developers have different priorities for many of the stories. They may suggest that the priority of a story be changed based on its technical risk or because it is complementary to another story. The customer team listens to their opinions but then prioritizes stories in the manner that maximizes the value delivered to the organization.

Stories cannot be prioritized without considering their costs. My priority for a vacation spot last summer was Tahiti until I considered its cost. At that point

From the Library of www.wowebook.com

other locations moved up in priority. Factored into the prioritization is the cost of each story. The cost of a story is the estimate given to it by the developers. Each story is assigned an estimate in *story points*, which indicates the size and complexity of the story relative to other stories. So, a story estimated at four story points is expected to take twice as long as a story estimated at two story points.

The release plan is built by assigning stories to the iterations in the release. The developers state their expected velocity, which is the number of story points they think they will complete per iteration. The customer then allocates stories to iterations, making sure that the number of story points assigned to any one iteration does not exceed the expected team velocity.

As an example, suppose that Table 1.1 lists all the stories in your project and they are sorted in order of descending priority. The team estimates a velocity of thirteen story points per iteration. Stories would be allocated to iterations as shown in Table 1.2.

Table 1.1 *Sample stories and their costs.*

Story	Story Points
Story A	3
Story B	5
Story C	5
Story D	3
Story E	1
Story F	8
Story G	5
Story H	5
Story I	5
Story J	2

Because the team expects a velocity of thirteen, no iteration can be planned to have more than thirteen story points in it. This means that the second and third iterations are planned to have only twelve story points. Don't worry about it—estimation is rarely precise enough for this difference to matter, and if the developers go faster than planned they'll ask for another small story or two. Notice that for the third iteration the customer team has actually chosen to

From the Library of www.wowebook.com

include Story J over the higher priority Story I. This is because Story I, at five story points, is actually too large to include in the third iteration.

Table 1.2 *A release plan for the stories of Table 1.1.*

Iteration	Stories	Story Points
Iteration 1	A, B, C	13
Iteration 2	D, E, F	12
Iteration 3	G, H, J	12
Iteration 4	I	5

An alternative to temporarily skipping a large story and putting a smaller one in its place in an iteration is to split the large story into two stories. Suppose that the five-point Story I could have been split into Story Y (three points) and Story Z (two points). Story Y contains the most important parts of the old Story I and can now fit in the third iteration, as shown in Table 1.3. For advice on how and when to split stories see Chapter 2, "Writing Stories," and Chapter 7, "Guidelines for Good Stories."

Table 1.3 *Splitting a story to create a better release plan.*

Iteration	Stories	Story Points
Iteration 1	A, B, C	13
Iteration 2	D, E, F	12
Iteration 3	G, H, Y	13
Iteration 4	J, Z	4

What Are Acceptance Tests?

Acceptance testing is the process of verifying that stories were developed such that each works exactly the way the customer team expected it to work. Once an iteration begins, the developers start coding and the customer team starts specifying tests. Depending on the technical proficiency of customer team members, this may mean anything from writing tests on the back of the story card to putting the tests into an automated testing tool. A dedicated and skilled tester should be included on the customer team for the more technical of these tasks.

Tests should be written as early in an iteration as possible (or even slightly before the iteration if you're comfortable taking a slight guess at what will be in

From the Library of www.wowebook.com

the upcoming iteration). Writing tests early is extremely helpful because more of the customer team's assumptions and expectations are communicated earlier to the developers. For example, suppose you write the story "A user can pay for the items in her shopping cart with a credit card." You then write these simple tests on the back of that story card:

- Test with Visa, MasterCard and American Express (pass).
- Test with Diner's Club (fail).
- Test with a Visa debit card (pass).
- Test with good, bad and missing card ID numbers from the back of the card.
- Test with expired cards.
- Test with different purchase amounts (including one over the card's limit).

These tests capture the expectations that the system will handle Visa, MasterCard and American Express and will not allow purchases with other cards. By giving these tests to the programmer early, the customer team has not only stated their expectations, they may also have reminded the programmer of a situation she had otherwise forgotten. For example, she may have forgotten to consider expired cards. Noting it as a test on the back of the card before she starts programming will save her time. For more on writing acceptance tests for stories see Chapter 6, "Acceptance Testing User Stories."

Why Change?

At this point you may be asking why change? Why write story cards and hold all these conversatons? Why not just continue to write requirements documents or use cases? User stories offer a number of advantages over alternative approaches. More details are provided in Chapter 13, "Why User Stories?", but some of the reasons are:

- User stories emphasize verbal rather than written communication.
- User stories are comprehensible by both you and the developers.
- User stories are the right size for planning.
- User stories work for iterative development.

From the Library of www.wowebook.com

- User stories encourage deferring detail until you have the best understanding you are going to have about what you really need.

Because user stories shift emphasis toward talking and away from writing, important decisions are not captured in documents that are unlikely to be read. Instead, important aspects about stories are captured in automated acceptance tests and run frequently. Additionally, we avoid obtuse written documents with statements like:

> The system must store an address and business phone number or mobile phone number.

What does that mean? It could mean that the system must store one of these:

> (Address and business phone) or mobile phone
> Address and (business phone or mobile phone)

Because user stories are free of technical jargon (remember, the customer team writes them), they are comprehensible by both the developers as well as the customer team.

Each user story represents a discrete piece of functionality; that is, something a user would be likely to do in a single setting. This makes user stories appropriate as a planning tool. You can assess the value of shifting stories between releases far better than you can assess the impact of leaving out one or more "The system shall…" statements.

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 successively refine those areas until the product is satisfactory. With each iteration the software is improved through the addition of greater detail. Stories work well for iterative development because it is also possible to iterate over the stories. For a feature that you want eventually but that isn't important right now, you can first write a large story (an epic). When you're ready to add that story into the system you can refine it by ripping up the epic and replacing it with smaller stories that will be easier to work with.

It is this ability to iterate over a story set that allows stories to encourage the deferring of detail. Because we can write a placeholder epic today, there is no need to write stories about parts of a system until close to when those parts will be developed. Deferring detail is important because it allows us to not spend time thinking about a new feature until we are positive the feature is needed. Stories discourage us from pretending we can know and write everything in advance. Instead they encourage a process whereby software is iteratively refined based on discussions between the customer team and the developers.

From the Library of www.wowebook.com

Summary

- A story card contains a short description of user- or customer-valued functionality.
- A story card is the visible part of a story, but the important parts are the conversations between the customer and developers about the story.
- The customer team includes those who ensure that the software will meet the needs of its intended users. This may include testers, a product manager, real users, and interaction designers.
- The customer team writes the story cards because they are in the best position to express the desired features and because they must later be able to work out story details with the developers and to prioritize the stories.
- Stories are prioritized based on their value to the organization.
- Releases and iterations are planned by placing stories into iterations.
- Velocity is the amount of work the developers can complete in an iteration.
- The sum of the estimates of the stories placed in an iteration cannot exceed the velocity the developers forecast for that iteration.
- If a story won't fit in an iteration, you can split the story into two or more smaller stories.
- Acceptance tests validate that a story has been developed with the functionality the customer team had in mind when they wrote the story.
- User stories are worth using because they emphasize verbal communication, can be understood equally by you and the developers, can be used for planning iterations, work well within an iterative development process, and because they encourage the deferring of detail.

Questions

1.1 What are the three parts of a user story?

1.2 Who is on the customer team?

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

From the Library of www.wowebook.com

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 country from a drop-down list.

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.

1.4 What advantages do requirements conversations have over requirements documents?

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

From the Library of www.wowebook.com

Chapter 2

Writing Stories

In this chapter we turn our attention to writing the stories. To create good stories we focus on six attributes. A good story is:

- Independent
- Negotiable
- Valuable to users or customers
- Estimatable
- Small
- Testable

Bill Wake, author of *Extreme Programming Explored* and *Refactoring Workbook*, has suggested the acronym INVEST for these six attributes (Wake 2003a).

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 we are working on the BigMoneyJobs website and need to write stories for how companies can pay for the job openings they post to our site. We could write these stories:

1. A company can pay for a job posting with a Visa card.
2. A company can pay for a job posting with a MasterCard.

From the Library of www.wowebook.com

3. A company can pay for a job posting with an American Express card.

Suppose the developers estimate that 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—which story should be given the three day estimate?

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 company can pay for a job posting 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. If the estimates for these stories had been longer, then an alternative split would be:

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

If you don't want to combine the stories and can't find a good way to split them, you can always take the simple approach of putting two estimates on the card: one estimate if the story is done before the other story, a lower estimate if it is done after.

Negotiable

Stories are negotiable. They are not written contracts or requirements that the software must implement. Story cards are short descriptions of functionality, the details of which are to be negotiated in a conversation between the customer and the development team. Because story cards are reminders to have a conversation rather than fully detailed requirements themselves, they 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 card, as shown in Story Card 2.1. The challenge comes in learning to include just enough detail.

Story Card 2.1 works well because it provides the right amount of information to the developer and customer who will talk about the story. When a devel-

From the Library of www.wowebook.com

A company can pay for a job posting with a credit card.

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

Story Card 2.1 A story card with notes providing additional detail.

oper starts to code this story, she will be reminded that a decision has already been made 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 help a developer and the customer to resume a conversation where it left off previously. Ideally, the conversation can be resumed this easily regardless of whether it is the same developer and customer who resume the conversation. 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 Story Card 2.2. 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 use").

A company can pay for a job posting 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 two digits of the card number. The system can store a card number for future use. Collect the expiration month and date of the card.

Story Card 2.2 A story card with too much detail.

Working with stories like Story Card 2.2 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 company can pay for a job posting with a credit card" they will not forget that their discussion is somewhat abstract. There are too many missing details for

From the Library of www.wowebook.com

them to mistakenly view their discussion as definitive or their estimate as accurate. However, when as much detail is added as in Story Card 2.2, discussions about the story are much more likely to feel concrete and real. This can lead to the mistaken belief that the story cards reflect all the details and that there's no further need to discuss the story with the customer.

If we think about the story card as a reminder for the developer and customer to have a conversation, then it is useful to think of the story card 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. Story Card 2.3 and Story Card 2.4 show how the excess detail of Story Card 2.2 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 notes about open questions while the back of the card contains details about the story in the form of tests that will prove whether or not it works as expected.

> A company can pay for a job posting with a credit card.
>
> Note: Will we accept Discover cards?
> Note for UI: Don't have a field for card type (it can be derived from first two digits on the card).

Story Card 2.3 The revised front of a story card with only the story and questions to be discussed.

Valuable to Purchasers 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 *purchaser* (someone who purchases the software), suppose a development team is building software that will be deployed across a

From the Library of www.wowebook.com

> 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 over $100 and under $100.

Story Card 2.4 Details that imply test cases are separated from the story itself. Here they are shown on the back of the story card.

large user base, perhaps 5,000 computers in a single company. The purchaser 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 purchasers might.

Similarly, stories like the following might be valued by purchasers contemplating buying the product but would not be 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 you want to avoid are stories that are only valued by developers. For example, avoid stories like these:

- All connections to the database are 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. This will allow the customer to intelligently prioritize these stories into the development schedule. 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.

From the Library of www.wowebook.com

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.

The best way to ensure that each story is valuable to the customer or users is to have the customer write the stories. Customers are often uncomfortable with this initially—probably because developers have trained them to think of everything they write as something that can be held against them later. ("Well, the requirements document didn't say that…") Most customers begin writing stories themselves once they become comfortable with the concept that story cards are reminders to talk later rather than formal commitments or descriptions of specific functionality.

Estimatable

It is important for developers to be able to estimate (or at least take a guess at) the size of a story or the amount of time it will take to turn a story into working code. There are three common reasons why a story may not be estimatable:

1. Developers lack domain knowledge.
2. Developers lack technical knowledge.
3. The story is too big.

First, the developers may lack domain knowledge. If the developers do not understand a story as it is written, they should discuss it with the customer who wrote the story. Again, it's not necessary to understand all the details about a story, but the developers need to have a general understanding of the story.

Second, a story may not be estimatable because the developers do not understand the technology involved. For example, on one Java project we were asked to provide a CORBA interface into the system. No one on the team had done that so there was no way to estimate the task. The solution in this case is to 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. During the spike the developers learn just enough that they can estimate the task. The spike itself is always given a defined maximum amount of time (called a *timebox*), which allows us to estimate the spike. In this way an unestimatable story turns into two stories: a quick spike to gather information and then a story to do the real work.

From the Library of www.wowebook.com

Finally, the developers may not be able to estimate a story if it is too big. For example, for the BigMoneyJobs website, the story "A Job Seeker can find a job" is too large. In order to estimate it the developers will need to disaggregate it into smaller, constituent stories.

A Lack of Domain Knowledge

As an example of needing more domain knowledge, we were building a website for long-term medical care of chronic conditions. The customer (a highly qualified nurse) wrote a story saying "New users are given a diabetic screening." The developers weren't sure what that meant and it could have run the gamut from a simple web questionnaire to actually sending something to new users for an at-home physical screening, as was done for the company's product for asthma patients. The developers got together with the customer and found out that she was thinking of a simple web form with a handful of questions.

Even though they are too big to estimate reliably, it is sometimes useful to write epics such as "A Job Seeker can find a job" because they serve as placeholders or reminders about big parts of a system that need to be discussed. If you are making a conscious decision to temporarily gloss over large parts of a system, then consider writing an epic or two that cover those parts. The epic can be assigned a large, pulled–from–thin–air estimate.

Small

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. Epics are difficult to work with because they frequently contain multiple stories. For example, in a travel reservation system, "A user can plan a vacation" is an epic. Planning a vacation is important functionality for a travel reservation system but there are many tasks involved in doing so. The epic should be split into smaller stories. The ultimate determination of whether a story is appropriately sized is based on the team, its capabilities, and the technologies in use.

From the Library of www.wowebook.com

Splitting Stories

Epics typically fall into one of two categories:

- The compound story
- The complex story

A compound story is an epic that comprises multiple shorter stories. For example, the BigMoneyJobs system may include the story "A user can post her resume." During the initial planning of the system this story may be appropriate. But when the developers talk to the customer, they find out that "post her resume" actually means:

- that a resume can include education, prior jobs, salary history, publications, presentations, community service, and an objective
- that users can mark resumes as inactive
- that users can have multiple resumes
- that users can edit resumes
- that users can delete resumes

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. For example, depending on the technologies in use and the size and skill of the team, stories like these will generally be too small:

- A Job Seeker can enter a date for each community service entry on a resume.
- A Job Seeker can edit the date for each community service entry on a resume.
- A Job Seeker can enter a date range for each prior job on a resume.
- A Job Seeker can edit the date range for each prior job on a resume.

Generally, a better solution is to group the smaller stories as follows:

- A user can create resumes, which include education, prior jobs, salary history, publications, presentations, community service, and an objective.
- A user can edit a resume.
- A user can delete a resume.

From the Library of www.wowebook.com

- A user can have multiple resumes.
- A user can activate and inactivate resumes.

There are normally many ways to disaggregate a compound story. The preceding disaggregation is along the lines of create, edit, and delete, which is commonly used. This works well if the create story is small enough that it can be left as one story. An alternative is to disaggregate along the boundaries of the data. To do this, think of each component of a resume as being added and edited individually. This leads to a completely different disaggregation:

- A user can add and edit education information.
- A user can add and edit job history information.
- A user can add and edit salary history information.
- A user can add and edit publications.
- A user can add and edit presentations.
- A user can add and edit community service.
- A user can add and edit an objective.

And so on.

Unlike the compound story, the complex story is a user story that is inherently large and cannot easily be disaggregated 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 "A company can pay for a job posting with a credit card" but none of the developers has ever done credit card processing before. They may choose to split the stories like this:

- Investigate credit card processing over the web.
- A user can pay with a credit card.

In this case the first story will send one or more developers on a spike. When complex stories are split in this way, always define 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

From the Library of www.wowebook.com

to a standard statistical approach called 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.

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.

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 has only the complex story to prioritize ("Add novel extensions to standard expectation maximization") and an estimate for the story, 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 has 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 and user interface changes are common examples 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

From the Library of www.wowebook.com

and the entire collection is treated as a single story. If you've chosen to use paper note cards, you can do this by stapling them together with a cover card.

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 nonfunctional requirements, which are requirements about the software but not directly about its functionality. For example, consider these nonfunctional stories:

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

As written, these stories are not testable. Whenever possible, tests should be automated. This means strive for 99% automation, not 10%. You can almost always automate more than you think you can. When a product is developed incrementally, things can change very quickly and code that worked yesterday can stop working today. You want automated tests that will find this as soon as possible.

There is a very small subset of tests that cannot realistically be automated. For example, a user story that says "A novice user is able to complete common workflows without training" can be tested but cannot be automated. 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 "a user never has 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." And—even better—an automated test can be written to verify this.

From the Library of www.wowebook.com

Summary

- 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.
- The details of a story are negotiated between the user and the developers.
- Stories should be written so that their value to users or the customer is clear. The best way to achieve this is to have the customer write the stories.
- 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.
- 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.
- Stories need to be testable.

Developer Responsibilities

- You are responsible for helping the customer 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 ask for a story about the use of a technology or a piece of infrastructure, you are responsible for instead describing the need in terms of its value to users or the customer.

Customer Responsibilities

- You are responsible for writing 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.

From the Library of www.wowebook.com

Questions

2.1 For the following stories, indicate if it is a good story or not. If not, why?

a A user can quickly master the system.

b A user can edit the address on a resume.

c A user can add, edit and delete multiple resumes.

d The system can calculate saddlepoint approximations for distributions of quadratic forms in normal variables.

e All runtime errors are logged in a consistent manner.

2.2 Break this epic up into appropriately sized component stories: “A user can make and change automated job search agents.”

From the Library of www.wowebook.com

This page intentionally left blank

From the Library of www.wowebook.com

Chapter 3

User Role Modeling

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 (Constantine and Lockwood 1999) and interaction design (Cooper 1999) 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.

User Roles[1]

Suppose we are building the BigMoneyJobs job posting and search site. This type of site will have many different types of users. When we talk about *user* stories, who is the user we're talking about? Are we talking about Ashish who has a job but always keeps an eye out for a better one? Are we talking about Laura, a new college graduate looking for her first professional job? Are we talking about Allan, who has decided he'll take any job that lets him move to Maui and windsurf every afternoon? Or are we talking about Scott, who doesn't hate his job but has realized it's time to move on? Perhaps we're talking about Kindra who was laid off six months ago and was looking for a great job but will now take anything in the northeastern United States.

Or should we think of the user as coming from one of the companies posting the jobs? Perhaps the user is Mario, who works in human resources and posts

1. 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 at their website at www.foruse.com or in *Software for Use* (1999).

From the Library of www.wowebook.com

new job openings. Perhaps the user is Delaney, who also works in human resources but is responsible for reviewing resumes. Or perhaps the user is Savannah, who works as an independent recruiter and is looking for both good jobs and good people.

Clearly we cannot write stories from a single perspective and have those stories reflect the experiences, backgrounds and goals of each of these users. Ashish, an accountant, may look at the site once a month just to keep his options open. Allan, a waiter, may want to create a filter to notify him any time any job on Maui gets posted but he won't be able to do that unless we make it easy. Kindra may spend hours each day looking for a job, broadening her search as time goes by. If Mario and Delaney work for a large company with many positions to fill, they may spend four or more hours a day on the site.

While each user comes to your software with a different background and with different goals, 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, we could look at the users in the preceding example and group them into roles as shown in Table 3.1 into roles this way.

Table 3.1 *One possible list of roles for the BigMoneyJobs project.*

Role	Who
Job Seeker	Scott
First Timer	Laura
Layoff Victim	Kindra
Geographic Searcher	Allan
Monitor	Ashish
Job Poster	Mario, Savannah
Resume Reader	Delaney, Savannah

Naturally, there will be some overlap between different user roles. The Job Seeker, First Timer, Layoff Victim, Geographic Searcher, and Monitor roles will all use the job search features of the site. They may use them in different ways and at different frequencies, but much of how they use the system will be similar. The Resume Reader and Job Poster roles will probably overlap as well since these roles are both pursuing the same goal of finding good candidates.

Table 3.1 does not show the only possible way to group users of BigMoneyJobs into roles. For example, we could choose to include roles like Part-Timer,

From the Library of www.wowebook.com

Full-Timer and Contractor. In the rest of this chapter we'll look at how to come up with a list of roles and how to refine that list so that it is useful.

Role Modeling Steps

We will use the following steps to identify and select a useful set of user roles:

- brainstorm an initial set of user roles
- organize the initial set
- consolidate roles
- refine the roles

Each of these steps is discussed in the following sections.

Brainstorming an Initial Set of User 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 to which they can tape or pin cards. 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 is 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. Since this is 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.

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 need to last longer than fifteen minutes.

From the Library of www.wowebook.com

A User Role Is One User

When brainstorming a project's roles, stick to identifying roles that represent a single user. For example, for the BigMoneyJobs project it may be tempting to write stories such as "A company can post a job opening." However, since a company as a whole cannot use the software, the story will be better if it refers to a role that represents an individual.

Organizing the Initial Set

Once the group has finished identify roles, it's time to organize them. To do this, cards are moved around on the table or wall so that their positions indicate the relationships between the roles. Overlapping roles are placed so that their cards overlap. If the roles overlap a little, overlap the cards a little. If the roles overlap entirely, overlap the cards entirely. An example is shown in Figure 3.1.

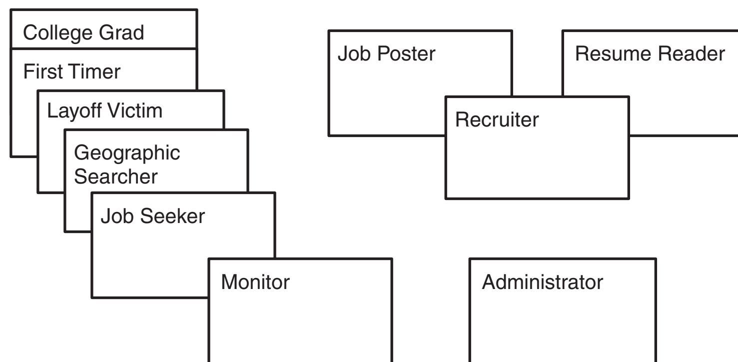

Figure 3.1 *Organizing the user role cards on a table.*

Figure 3.1 shows that the College Grad and First Timer, as those roles were intended by their card writers, overlap signficantly. There's less but similar overlap among the other cards representing people who will use the site to search for jobs. The Monitor role card is shown with only a slight overlap because that role refers to someone who is relatively happy in her current job but likes to keep her eyes open.

To the right of the job-seeking roles in Figure 3.1 are the Job Poster, Recruiter, and Resume Reader role cards. The Recruiter role is shown overlapping both Job Poster and Resume Reader because a recruiter will both post ads

From the Library of www.wowebook.com

and read resumes. An Administrator role is shown also. This role represents users internal to BigMoneyJobs who will support the system.

System Roles

As much as you can, stick with user roles that define people, as opposed to other systems. If you think it will help, then identify an occasional non-human user role. However, the purpose of identifying user roles is to make sure that we think really hard about the users that we absolutely, positively must satisfy with the new system. We don't need user roles for every conceivable user of the system, but we need roles for the ones who can make or break the success of the project. Since other systems are rarely purchasers of our system, they can rarely make or break the success of our system. Naturally, there can be exceptions to this and if you feel that adding a non-human user role helps you think about your system, then add it.

Consolidating Roles

After the roles have been grouped, 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 consolidated into a single role (perhaps taking its name from the two initial roles), or one of the initial role cards can be ripped up.

In Figure 3.1 the College Grad and First Timer roles are shown as heavily overlapping. The group decides to rip up the College Grad card since any stories for that user role would likely be identical to stories for a First Timer. Even though First Timer, Layoff Victim, Geographic Searcher and Job Seeker have significant overlap, the group decides that each represents a constituency that will be important to satisfy and the roles will have important but subtly different goals for how they use the BigMoneyJobs website.

When they look at the right side of Figure 3.1, the group decides that it is not worth distinguishing between a Job Poster and a Resume Reader. They decide that a Recruiter covers these two roles adequately and those cards are ripped up. However, the group decides that there are differences between an Internal Recruiter (working for a specific company) and an External Recruiter (matching candidates to jobs at any company). They create new cards for Internal Recruiter and External Recruiter, and consider these as specialized versions of the Recruiter role.

From the Library of www.wowebook.com

In addition to consolidating overlapping roles, the group should also rip up any role cards for roles that are unimportant to the success of the system. For example, the Monitor role card represents someone who is just keeping an eye on the job market. A Monitor may not switch jobs for three years. BigMoney-Jobs can probably do quite well without paying attention to that user role. They decide they will be better off focusing on the roles that will be important to the success of the company, such as Job Seeker and the Recruiter roles.

After the team has consolidated the cards, they are arranged on the table or wall to show relationships between the roles. Figure 3.2 shows one of many possible layouts for the BigMoneyJobs role cards. Here a generic role, such as Job Seeker or Recruiter, is positioned above specialized versions of that role. Alternatively, cards can be stacked or positioned in any other way that the group desires in order to show whatever relationships they think are important.

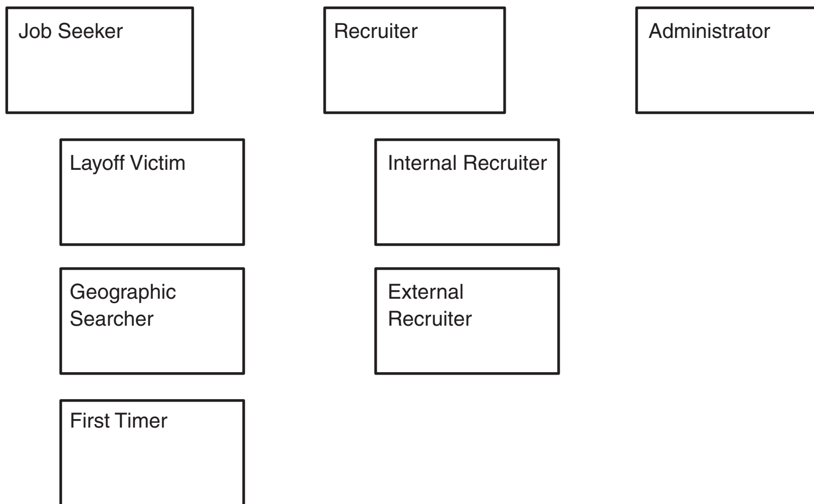

Figure 3.2 *The consolidated role cards.*

Refining the Roles

Once we've consolidated roles and have a basic understanding for how the roles relate to each other, it is possible to model those roles by defining attributes of each 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:

From the Library of www.wowebook.com

- 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, for the BigMoneyJobs website you may want to consider whether the user role will be looking for a part-time or full-time job.

As you identify interesting attributes for a role, write notes on the role card. When finished, you can hang the role cards in a common area used by the team so they can be used as reminders. A sample user role card is shown in Figure 3.3.

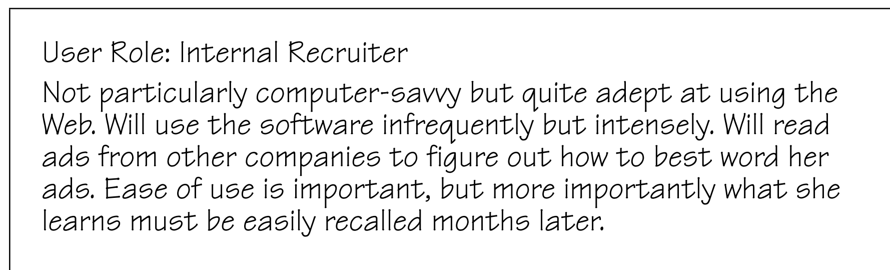

Figure 3.3 *A sample user role card.*

Two Additional Techniques

We could stop right now if we want to. By now a team might have spent an hour—almost certainly no more than that—and they will have put more thought into the users of their software than probably 99% of all software teams. Most teams should, in fact, stop at this point. However, there are two additional techniques that are worth pointing out because they may be helpful in thinking about users on some projects. Only use these techniques if you can anticipate a direct, tangible benefit to the project.

From the Library of www.wowebook.com

Personas

Identifying user roles is a great leap forward, but for some of the more important user roles, it might be worth going one step further and creating a *persona* for the role. A persona is an imaginary representation of a user role. Earlier in this chapter we met Mario who is responsible for posting new job openings for his company. 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, Mario may be described as follows:

> Mario works as a recruiter in the Personnel department of SpeedyNetworks, a manufacturer of high-end networking components. He's worked for SpeedyNetworks six years. Mario has a flex-time arrangement and works from home every Friday. Mario is very strong with computers and considers himself a power user of just about all the products he uses. Mario's wife, Kim, is finishing her Ph.D. in chemistry at Stanford University. Because SpeedyNetworks has been growing almost consistently, Mario is always looking for good engineers.

If you choose to create personas for your project, be careful that enough market and demographic research has been done that the personas chosen truly represent the product's target audience.

This persona description gives us a good introduction to Mario. However, nothing speaks as loudly as a picture, so you should also find a picture of Mario 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 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 may, however, think about writing a persona definition for one or two of the primary user roles. If the system you are building is such that it is absolutely vital that the product satisfy one or two user roles, then those user roles are candidates for expansion into personas.

Stories become much more expressive when put in terms of a user role or persona. After you have identified user roles and possibly a persona or two, you can begin to speak in terms of roles and personas instead of the more generic "the user." Rather than writing stories like "A user can restrict job searches to specific geographic regions" you can write "A Geographic Searcher can restrict

From the Library of www.wowebook.com

his job searches to a specific geographic region." Hopefully writing a story this way reminds the team about Allan who is looking for any job on Maui. Writing some stories with user role or persona names does not mean that other roles cannot perform those stories; rather, it means that there is some benefit in thinking about a specific user role or persona when discussing or coding the story.

Extreme Characters

Djajadiningrat and co-authors (2000) have proposed a second technique you might want to think about: the use of extreme characters when considering the design of a new system. They describe an example of designing a Personal Digital Assistant (PDA) handheld computer. They advise that instead of designing solely for a typical sharp-dressed, BMW-driving management consultant, the system designers should consider users with exaggerated personalities. Specifically, the authors 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. At a minimum, you can have a few minutes of fun thinking about how the Pope might use your software and it just may lead to an insight or two.

What If I Have On-Site Users?

The user role modeling techniques described in this chapter are still useful even if you have real, live users in your building. Working with real users will strongly improve your likelihood of delivering the desired software. However, even with real users there is no guarantee that you have the right users or the right mix of users.

From the Library of www.wowebook.com

To decrease the likelihood of failing to satisfy important users, you should do some simple role modeling on projects even when you have available internal users.

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.
- 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 them 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.

From the Library of www.wowebook.com

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.
- 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.

Questions

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

3.2 Consolidate the roles you came up with in the previous question and show how you would lay out the role cards. Explain your answer.

3.3 Write persona descriptions for the one most important user role.

From the Library of www.wowebook.com

Chapter 7

Guidelines for Good Stories

At this point, with a good foundation of what stories are, how to trawl for and write them, how to identify key user roles, and the role of acceptance testing, we turn our attention to some additional guidelines for writing good stories.

Start with Goal Stories

On a large project, especially one with many user roles, it is sometimes difficult to even know where to begin in identifying stories. What I've found works best is to consider each user role and identify the goals that user has for interacting with our software. For example, consider the Job Seeker role in the BigMoney-Jobs example. She really has one top priority goal: find a job. But we may consider that goal to comprise the following goals:

- search for jobs she's interested in (based on skill, salary, location, and so on)
- automate the search process so she doesn't have to search manually each time
- make her resume available so that companies may search for her
- easily apply for any jobs she likes

These goals (which really are high-level stories themselves) can then be used to generate additional stories as needed.

Slice the Cake

When faced with a large story, there are normally many ways of breaking it into smaller pieces. The first inclination of many developers is to split the story

From the Library of www.wowebook.com

along technical lines. For example, suppose the team has decided that the story "A Job Seeker can post a resume" is simply too large to fit in the current iteration and must be split. The developers may want to split it along technical boundaries, such as:

- A Job Seeker can fill out a resume form.
- Information on a resume form is written to the database.

In this case, one story would be done in the current iteration while the other story would be deferred until (presumably) the next iteration. The problem with this is that neither story on its own is very useful to users. The first story says that job seekers can fill out a form but that the data is not saved. Not only is this not useful, it would actually waste users' time. The second story says that the data collected on the form will be written to the database. Without a story to present the form to users, the second story is not useful.

A far better approach is to write the replacement stories such that each provides some level of end–to–end functionality. Bill Wake (2003a) refers to this as "slicing the cake." Each story must have a little from each layer. This leads to splitting "A Job Seeker can post a resume" like this:

- A Job Seeker can submit a resume that includes only basic information such as name, address, education history.
- A Job Seeker can submit a resume that includes all information an employer may want to see.

Stories that represent a full slice of cake are to be preferred over those that do not. There are two reasons for this. First, exercising each layer of an application's architecture reduces the risk of finding last minute problems in one of the layers. Second, although not ideal, an application could conceivably be released for use with only partial functionality as long as the functionality that is included in the release slices all the way through the system.

Write Closed Stories

Soren Lauesen (2002) introduces the idea of closure for tasks in his compendium of requirements techniques. His ideas are equally applicable to user stories. A closed story is one that finishes with the achievement of a meaningful goal and that allows the user to feel she has accomplished something.

From the Library of www.wowebook.com

For example, suppose the BigMoneyJobs website project includes the story "A recruiter can manage the ads she has placed." This is not a closed story: Managing the ads she's placed is not something that is ever completely done. Instead, it is an ongoing activity. This story can be better constructed as a set of closed stories, such as:

- A recruiter can review resumes from applicants to one of her ads.
- A recruiter can change the expiration date of an ad.
- A recruiter can delete an application that is not a good match for a job.

And so on. Each of these closed stories is a part of the original story that was not closed. After completing one of these closed stories, a user is likely to feel a sense of accomplishment.

The desire to write closed stories has to be tempered against competing needs. Remember that stories also need to be small enough to be estimatable and small enough to be conveniently scheduled into a single iteration. But stories must also be large enough that you avoid capturing details about them any earlier than necessary.

Put Constraints on Cards

Newkirk and Martin (2001) recommend a practice I've found useful. They introduce the practice of annotating a story card with "Constraint" for any story that must be obeyed rather than directly implemented. An example can be seen in Story Card 7.1.

> The system must support peak usage of up to 50 concurrent users.
>
> Constraint

Story Card 7.1 An example of a constraint story card.

Other examples of constraints are:

- Do not make it hard to internationalize the software if needed later.
- The new system must use our existing order database.

From the Library of www.wowebook.com

- The software must run on all versions of Windows.
- The system will achieve uptime of 99.999%.
- The software will be easy to use.

Even though constraint cards do not get estimated and scheduled into iterations like normal cards, they are still useful. Minimally, constraint cards can be taped to the wall where they act as reminders. Even better, acceptance tests can be written to ensure the constraint is not violated. For example, it would not be difficult to write a test for Story Card 7.1. Ideally the team would write this test during one of the first iterations when there's little chance of it being violated. The team would then continue running the test as part of each subsequent iteration. Whenever possible (and it usually is), write automated tests to ensure that constraints are being met.

For more on constraints as a way of specifying nonfunctional requirements see Chapter 16, "Additional Topics."

Size the Story to the Horizon

You want to focus your attention on the areas that most need it. Usually, this means paying more attention to things happening in the near future than to things happening further out. With stories, you do this by writing stories at different levels based on the implementation horizon of the stories. This means, for example, that stories for the next few iterations would be written at sizes that can be planned into those iterations, while more distant stories could be much larger and less precise. For example, suppose at the highest level we've determined that the BigMoneyJobs website will include four stories:

- A Job Seeker can post a resume.
- A Job Seeker can search job openings.
- A Recruiter can post a job opening.
- A Recruiter can search resumes.

The customer has decided that the first iterations will focus on allowing users to post resumes. Only after the bulk of the resume posting functionality has been added will attention be turned to searching for jobs, posting job openings, and searching resumes. This means that the project team and customer will start having the conversations about "A Job Seeker can post a resume."

From the Library of www.wowebook.com

That story will be expanded as details are discovered through those conversations; the other three high-level stories will be left alone. A possible list of stories then becomes:

- A Job Seeker can add a new resume to the site.
- A Job Seeker can edit a resume that is already on the site.
- A Job Seeker can remove her resume from the site.
- A Job Seeker can mark a resume as inactive.
- A Job Seeker can mark a resume as hidden from certain employers.
- A Job Seeker can see how many times her resume has been viewed.
- … and so on about posting resumes…
- A Job Seeker can search job openings.
- A Recruiter can post job openings.
- A Recruiter can search resumes.

In writing your stories, take advantage of the flexibility of stories to be useful at various levels.

Keep the UI Out as Long as Possible

One of the problems that has plagued every approach to software requirements has been mixing requirements with solution specification. That is, in stating a requirement, a solution is also either explicitly stated or implied. Most commonly this happens with aspects of the user interface. You want to keep the user interface out of your stories as long as possible. For example, consider Story Card 7.2, which is a story from a real system. If this story is to be developed early in the life of the project, it includes too many user interface details. Readers of this story are told about the print dialog, printer lists, and at least four ways of searching.

Eventually it will become inevitable for user interface details to slip into stories. This happens as the software becomes more and more complete, and stories shift away from being entirely new functionality to being modifications or extensions of existing functionality.

For example, consider the story "A user can select dates from a date widget on the search screen." This story may represent three days of work regardless of

From the Library of www.wowebook.com

whether it's done at the start or end of the project. However, it is not a story you'd expect to have at the start of the project before the user interface has even been considered.

> Print dialog allows the user to edit the printer list. The user can add or remove printers from the printer list. The user can add printers either by auto-search or manually specifying the printer DNS name or IP address. An advanced search option also allows the user to restrict his search within specified IP addresses and subnet range.

Story Card 7.2 A card with too much user interface detail.

Some Things Aren't Stories

While user stories are a very flexible format that works well for describing much of the functionality of many systems, they are not appropriate for everything. If you need to express some requirements in a form other than user stories, then do so. For example, user interface guidelines are often described in documents with lots of screen captures. Similarly, apart from any user stories, you may want to document and agree upon an interface between important systems, especially if one is being developed by an external vendor.

If you find that some aspect of a system could benefit from expression in a different format, then use that format.

Include User Roles in the Stories

If the project team has identified user roles, they should make use of them in writing the stories. So instead of writing "A user can post her resume" they write "A Job Seeker can post her resume." The difference is minor but writing stories in this way keeps the user in the forefront of the developer's mind. Instead of thinking of bland, faceless, interchangeable users, the developer will begin thinking of real, tangible users whom she needs to satisfy with the software.

From the Library of www.wowebook.com

Connextra, one of the early adopters of Extreme Programming, incorporated roles into their stories by using a short template. Each story was written in the following format:

I as a (role) want (function) so that (business value)

You may want to experiment with this template or with one of your own. A template like this can help distinguish important from frivolous stories.

Write for One User

Stories are generally most readable when written for a single user. For many stories, writing for one or many users will not make a difference. However, for some stories the difference can be significant. For example, consider the story "Job Seekers can remove resumes from the site." This could be interpreted to mean that one Job Seeker can remove her own resume and possibly the resumes of others.

Normally, this type of issue will become clear when you think about a story with a single user in mind. For example, the story above could be written as "A Job Seeker can remove resumes." When written this way, the problem of one Job Seeker removing resumes of others becomes more apparent, though, and the story can be further improved to "A Job Seeker can remove her own resumes."

Write in Active Voice

User stories are easier to read and understand when written in active voice. For example, rather than saying "A resume can be posted by a Job Seeker" say "A Job Seeker can post a resume."

Customer Writes

Ideally the customer writes the stories. On many projects the developers help out, either by doing the actual writing during an initial story writing workshop or by suggesting new stories to the customer. But, responsibility for writing stories resides with the customer and cannot be passed to the developers.

From the Library of www.wowebook.com

Additionally, because the customer is responsible for prioritizing the stories that will go into each iteration, it is vital that the customer understand each story. The best way to do this is to write them.

Don't Number Story Cards

The first time we use story cards many of us are tempted to number them. The usual reasoning is that this will help keep track of individual cards or add some level of traceability to stories. For example, when we discover that the story on card 13 is too large we rip up card 13 and replace it with cards 13.1, 13.2, and 13.3. However, numbering story cards adds pointless overhead to the process and leads us into abstract discussions about features that need to be tangible. I'd rather talk about "the story to add user groups" than "story 13." I especially don't want to talk about "story 13.1."

If you feel compelled to number story cards, instead try adding a short title to the card and use the title as shorthand for the rest of the story text.

Don't Forget the Purpose

Don't forget that the main purpose of a story card is to act as a reminder to discuss the feature. Keep these reminders brief. Add the detail you need to remember where to resume a conversation, but do not replace the conversation by adding more detail to the story card.

Summary

- To identify stories, start by considering the goals of each user role in using the system.
- When splitting a story, try to come up with stories that cut through all layers of the application.
- Try to write stories that are of a size where the user feels justified in taking a coffee break after completing the story.
- Augment stories with other requirements gathering or documenting techniques as necessary for the project's domain and environment.

From the Library of www.wowebook.com

- Create constraint cards and either tape them to a shared wall or write tests to ensure the constraints are not violated.
- Write smaller stories for functionality the team will implement soon, and write broad, high-level stories for functonality further into the future.
- Keep the user interface out of the stories for as long as possible.
- When practical, include the user role when writing the story.
- Write stories in active voice. For example, say "A Job Seeker can post a resume" rather than "A resume can be posted by a Job Seeker."
- Write stories for a single user. Instead of "Job Seekers can remove resumes" write "A Job Seeker can remove her own resumes."
- Have the customer, rather than a developer, write the stories.
- Keep user stories short, and don't forget their purpose as reminders to hold conversations.
- Don't number story cards.

Questions

7.1 Assume the story "A Job Seeker can search for open jobs" is too large to fit into one iteration. How would you split it?

7.2 Which of these stories is appropriately sized and can be considered a closed story?

a A user can save her preferences.

b A user can change the default credit card used for purchases.

c A user can log on to the system.

7.3 What simple changes could improve the story "Users can post their resumes"?

7.4 How would you test the constraint "The software will be easy to use"?

From the Library of www.wowebook.com