

The Addison-Wesley Signature Series

A KENT BECK
SIGNATURE
BOOK

USER STORIES APPLIED

FOR AGILE SOFTWARE DEVELOPMENT

MIKE COHN
Foreword by Kent Beck



Contents

Foreword	xv
Acknowledgments	xvii
Introduction	xix
PART I: Getting Started.....	1
Chapter 1: An Overview	3
What Is a User Story?.....	4
Where Are the Details?.....	5
“How Long Does It Have to Be?”	7
The Customer Team.....	8
What Will the Process Be Like?	8
Planning Releases and Iterations.....	10
What Are Acceptance Tests?	12
Why Change?.....	13
Summary	15
Questions.....	15
Chapter 2: Writing Stories	17
Independent	17
Negotiable	18
Valuable to Purchasers or Users	20
Estimatable.....	22
Small	23
Testable	27
Summary	28
Developer Responsibilities	28
Customer Responsibilities	28
Questions.....	29

Chapter 3:	User Role Modeling	31
	User Roles	31
	Role Modeling Steps	33
	Two Additional Techniques	37
	What If I Have On-Site Users?	39
	Summary	40
	Developer Responsibilities	40
	Customer Responsibilities	41
	Questions	41
Chapter 4:	Gathering Stories	43
	Elicitation and Capture Should Be Illicit	43
	A Little Is Enough, or Is It?	44
	Techniques	45
	User Interviews	45
	Questionnaires	47
	Observation	48
	Story-Writing Workshops	49
	Summary	52
	Developer Responsibilities	53
	Customer Responsibilities	53
	Questions	53
Chapter 5:	Working with User Proxies	55
	The Users' Manager	55
	A Development Manager	57
	Salespersons	57
	Domain Experts	58
	The Marketing Group	59
	Former Users	59
	Customers	59
	Trainers and Technical Support	61
	Business or Systems Analysts	61
	What to Do When Working with a User Proxy	61
	Can You Do It Yourself?	63
	Constituting the Customer Team	63
	Summary	64
	Developer Responsibilities	65
	Customer Responsibilities	65
	Questions	66

Chapter 6:	Acceptance Testing User Stories	67
	Write Tests Before Coding	68
	The Customer Specifies the Tests	69
	Testing Is Part of the Process.	69
	How Many Tests Are Too Many?.	70
	The Framework for Integrated Test.	70
	Types of Testing	72
	Summary.	73
	Developer Responsibilities.	73
	Customer Responsibilities	73
	Questions	74
Chapter 7:	Guidelines for Good Stories	75
	Start with Goal Stories	75
	Slice the Cake	75
	Write Closed Stories	76
	Put Constraints on Cards	77
	Size the Story to the Horizon.	78
	Keep the UI Out as Long as Possible	79
	Some Things Aren't Stories	80
	Include User Roles in the Stories	80
	Write for One User	81
	Write in Active Voice	81
	Customer Writes	81
	Don't Number Story Cards	82
	Don't Forget the Purpose	82
	Summary.	82
	Questions	83
PART II:	Estimating and Planning	85
Chapter 8:	Estimating User Stories	87
	Story Points.	87
	Estimate as a Team	88
	Estimating.	88
	Triangulate	90
	Using Story Points.	91
	What If We Pair Program?	92
	Some Reminders	93
	Summary.	94



	Developer Responsibilities	94
	Customer Responsibilities	95
	Questions	95
Chapter 9:	Planning a Release	97
	When Do We Want the Release?	98
	What Would You Like in It?	98
	Prioritizing the Stories	99
	Mixed Priorities	100
	Risky Stories	101
	Prioritizing Infrastructural Needs	101
	Selecting an Iteration Length	103
	From Story Points to Expected Duration	103
	The Initial Velocity	104
	Creating the Release Plan	105
	Summary	106
	Developer Responsibilities	106
	Customer Responsibilities	107
	Questions	107
Chapter 10:	Planning an Iteration	109
	Iteration Planning Overview	109
	Discussing the Stories	110
	Disaggregating into Tasks	111
	Accepting Responsibility	113
	Estimate and Confirm	113
	Summary	115
	Developer Responsibilities	115
	Customer Responsibilities	116
	Questions	116
Chapter 11:	Measuring and Monitoring Velocity	117
	Measuring Velocity	117
	Planned and Actual Velocity	119
	Iteration Burndown Charts	121
	Burndown Charts During an Iteration	123
	Summary	126
	Developer Responsibilities	127
	Customer Responsibilities	127
	Questions	127

PART III: Frequently Discussed Topics	131
Chapter 12: What Stories Are Not	133
User Stories Aren't IEEE 830.	133
User Stories Are Not Use Cases.	137
User Stories Aren't Scenarios.	141
Summary.	143
Questions	143
Chapter 13: Why User Stories?	145
Verbal Communication.	145
User Stories Are Comprehensible	148
User Stories Are the Right Size for Planning	148
User Stories Work for Iterative Development	149
Stories Encourage Deferring Detail	150
Stories Support Opportunistic Development	151
User Stories Encourage Participatory Design.	152
Stories Build Up Tacit Knowledge.	153
Why Not Stories?	153
Summary.	154
Developer Responsibilities.	155
Customer Responsibilities	155
Questions	155
Chapter 14: A Catalog of Story Smells	157
Stories Are Too Small	157
Interdependent Stories.	157
Goldplating.	158
Too Many Details.	159
Including User Interface Detail Too Soon	159
Thinking Too Far Ahead.	160
Splitting Too Many Stories	160
Customer Has Trouble Prioritizing	161
Customer Won't Write and Prioritize the Stories.	162
Summary.	162
Developer Responsibilities.	163
Customer Responsibilities	163
Questions	163
Chapter 15: Using Stories with Scrum	165
Scrum Is Iterative and Incremental	165

The Basics of Scrum	166
The Scrum Team	166
The Product Backlog	167
The Sprint Planning Meeting	168
The Sprint Review Meeting	170
The Daily Scrum Meeting	171
Adding Stories to Scrum	173
A Case Study	174
Summary	175
Questions	176
Chapter 16: Additional Topics	177
Handling NonFunctional Requirements	177
Paper or Software?	179
User Stories and the User Interface	181
Retaining the Stories	184
Stories for Bugs	185
Summary	186
Developer Responsibilities	186
Customer Responsibilities	187
Questions	187
PART IV: An Example	189
Chapter 17: The User Roles	191
The Project	191
Identifying the Customer	191
Identifying Some Initial Roles	192
Consolidating and Narrowing	193
Role Modeling	195
Adding Personas	197
Chapter 18: The Stories	199
Stories for Teresa	199
Stories for Captain Ron	202
Stories for a Novice Sailor	203
Stories for a Non-Sailing Gift Buyer	204
Stories for a Report Viewer	204
Some Administration Stories	205
Wrapping Up	206

Chapter 19: Estimating the Stories	209
The First Story	210
Advanced Search	212
Rating and Reviewing	213
Accounts	214
Finishing the Estimates	215
All the Estimates	216
Chapter 20: The Release Plan	219
Estimating Velocity	219
Prioritizing the Stories	220
The Finished Release Plan	221
Chapter 21: The Acceptance Tests	223
The Search Tests	223
Shopping Cart Tests	224
Buying Books	225
User Accounts	226
Administration	227
Testing the Constraints	228
A Final Story	229
PART V: Appendices	231
Appendix A: An Overview of Extreme Programming	233
Roles	233
The Twelve Practices	234
XP's Values	240
The Principles of XP	241
Summary	242
Appendix B: Answers to Questions	245
Chapter 1, An Overview	245
Chapter 2, Writing Stories	246
Chapter 3, User Role Modeling	247
Chapter 4, Gathering Stories	248
Chapter 5, Working with User Proxies	249
Chapter 6, Acceptance Testing User Stories	249
Chapter 7, Guidelines for Good Stories	249
Chapter 8, Estimating User Stories	251
Chapter 9, Planning a Release	251



Chapter 10, Planning an Iteration	252
Chapter 11, Measuring and Monitoring Velocity . .	252
Chapter 12, What Stories Are Not	253
Chapter 13, Why User Stories?	254
Chapter 14, A Catalog of Story Smells	255
Chapter 15, Using Stories with Scrum	255
Chapter 16, Additional Topics	256
References	259
Index	263

Foreword

How do you decide what a software system is supposed to do? And then, how do you communicate that decision between the various people affected? This book takes on this complicated problem. The problem is difficult because each participant has different needs. Project managers want to track progress. Programmers want to implement the system. Product managers want flexibility. Testers want to measure. Users want a useful system. Creating productive conflict between these perspectives, coming to a single collective decision everyone can support, and maintaining that balance for months or years are all difficult problems.

The solution Mike Cohn explores in this book, *User Stories Applied*, is superficially the same as previous attempts to solve this problem—requirements, use cases, and scenarios. What’s so complicated? You write down what you want to do and then you do it. The proliferation of solutions suggests that the problem is not as simple as it appears. The variation comes down to what you write down and when.

User stories start the process by writing down just two pieces of information: each goal to be satisfied by the system and the rough cost of satisfying that goal. This takes just a few sentences and gives you information other approaches don't. Using the principle of the “last responsible moment,” the team defers writing down most details of the features until just before implementation.

This simple time shift has two major effects. First, the team can easily begin implementing the “juiciest” features early in the development cycle while the other features are still vague. The automated tests specifying the details of each feature ensure that early features continue to run as specified as you add new features. Second, putting a price on features early encourages prioritizing from the beginning instead of a panicked abortion of scope at the end in order to meet a delivery date.



FOREWORD

Mike's experience with user stories makes this a book full of practical advice for making user stories work for your development team. I wish you clear, confident development.

Kent Beck
Three Rivers Institute

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.

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:

- 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:

- 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

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.

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

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

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

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

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

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 conversations? 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.

- 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.

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?

- 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?

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.

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-

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

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

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.

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 *time-box*), 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.

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.

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.

- 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.

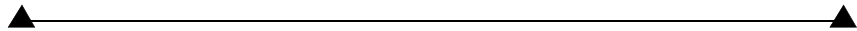
Complex stories are also common when developing new or extending known algorithms. One team in a biotech company had a story to add novel extensions

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

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.

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.

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.”

Chapter 12

What Stories Are Not

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 (IEEE 1998). This document, known as IEEE Standard 830, was last revised in 1998. The IEEE recommendations cover such topics as how to organize the requirements 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 company to pay for a job posting 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 before the job posting is placed on the site.
 - 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.

▼

A Warning Sign

One warning sign of a project going astray with a requirements specification is a ping-ponging of the specification document between the software development group and another group like Marketing or Product Management. What typically happens is the Product Management (or similar) group writes a requirements specification that is given to the developers. The developers then rewrite this document so that it conveys their interpretation of the requirements as first written by Product Management. The developers are always careful to give their document a completely different name (something like Functional Specification perhaps) to hide that it is the same document as the initial document, just written from the perspective of a different group.

Both groups know that a requirements specification for a project of any significance is too difficult to read and fully understand and impossible to write with the desired precision. So, whichever group writes the final requirements can claim ownership of the intent of the document. When the project is finished and blame is being allocated they will point to sections of the document and claim that missing features were implied. Or they will claim that expected functionality is clearly out of scope because of a sentence buried somewhere in the document.

Most of the times when I see two groups writing separate versions of essentially the same document I already know they are positioning themselves for the end-of-project blame sessions and for claiming to know the intent of the document. This type of silliness goes away with user stories. Along with the shift to conversations from documentation comes the freedom of knowing that nothing is final. Documents that look like contracts feel so final. Conversations don't feel that way. If we talk today and then learn something next month, we talk again.

▲

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 the reality on most projects.

Unfortunately, it is effectively impossible to write all of a system's requirements this way. There is a powerful and important feedback loop that occurs when users 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 changes are requested to the software described in a requirements specification, we've become accustomed to calling it 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 is put into upfront thinking about requirements, we've learned that users will have different (and better) opinions once they see the software. Second, this type of thinking reinforces the belief that software is complete when it fulfills a list of requirements, rather than when it fulfills its intended users' goals. If the scope of the user's goals changes 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 solution 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 (2000) suggests that designers "may produce a solution for only the first few requirements they encounter." For example, consider the following requirements:¹

- 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.

By this point I suppose images of an automobile are floating around 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 user told us her goals for the product:

- The product makes it easy and fast for me to mow my lawn
- I am comfortable while using the product

1. Adapted from *The Inmates Are Running the Asylum* (Cooper 1999).

By looking at the user's goals, we get a completely different view of the product and realize that the customer really wants a riding lawn mower, not an automobile. These goals are not user stories, but where IEEE 830 documents are a list of requirements, stories describe a user's goals. 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.

A final difference between user stories and IEEE 830-style requirements specifications is that with the latter the cost of each requirement is not made visible until all the requirements are written down. The typical scenario is that one or more analysts spends two or three months (often longer) writing a lengthy requirements document. This is then handed to the programmers who tell the analysts (who relay the message to the customer) that the project will take twenty-four months, rather than the six months they had hoped for. In this case, time was wasted writing the three-fourths of the document that the team won't have time to develop, and more time will be wasted as the developers, analysts and customer iterate over which functionality can be developed in time. With stories, an estimate is associated with each story right up front. The customer knows the velocity of the team and the story point cost of each story. After writing enough stories to fill all the iterations, she knows she's done.

Kent Beck explains this difference with an analogy of registering for a wedding.² When you register for a wedding you don't see the cost of each item. You just make a wish list of everything you want. That may work for weddings, but it doesn't work for software development. When a customer places an item on her project wish list, she needs to know the cost of it.

How and Why Will the Feature Be Used?

A further problem with requirements lists is that the items on the lists describe the behavior of the software, not the behavior or goals of a user. A requirements list rarely answers "But how and why will someone use this feature?"

Steve Berczuk, XP developer and author of *Software Configuration Management Patterns*, points out the importance of this question: "I can't count the number of times that I saved a whole lot of work by taking a feature list and asking a customer to create scenarios that use those features. Often the customer will realize that the feature really isn't needed, and that you should spend time building things that add value."³

2. Personal communication, November 7, 2003.

3. Steve Berczuk on extremeprogramming@yahoo.com, February 20, 2003.

User Stories Are Not Use Cases

First introduced by Ivar Jacobson (1992), use cases are today most commonly associated with the Unified Process. A use case is a generalized description of a set of interactions between the system and one or more actors, 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 (2001) are among the most commonly used. A sample is shown in Figure 12.1, which is equivalent to the user story “A Recruiter can pay for a job posting with a credit card.”

Since this is not a book on use cases we won’t fully cover all the details of the use case shown in Figure 12.1; 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 five 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 Figure 12.1. 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 five, an alternate scenario is represented by the sequence 1, 2, 2a, 2a1, 2, 3, 4, 5.

One of the most obvious differences between stories and use cases is their scope. Both are sized to deliver business value, but stories are kept smaller in scope because we place constraints on their size (such as no more than ten days of development work) so that they may be used in scheduling work. A use case almost always covers a much larger scope than a story. For example, looking at the user story “A Recruiter can pay for a job posting with a credit card” we see it is similar to the main success scenario of Figure 12.1. This leads to the observation that a user story is similar to a single scenario of a 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 Figure 12.1.

User stories and use cases also differ in the level of completeness. James Grenning has noted that the text on a story card “plus acceptance tests are basically the same thing as a use case.”⁴ By this Grenning means that the story cor-

4. James Grenning on extremeprogramming@yahoogroups.com, February 23, 2003.

Use Case Title: Pay for a job posting

Primary Actor: Recruiter

Level: Actor goal

Precondition: The job information has been entered but is not viewable.

Minimal Guarantees: None

Success Guarantees: Job is posted; recruiter's credit card is charged.

Main Success Scenario:

1. Recruiter submits credit card number, date, and authentication information.
2. System validates credit card.
3. System charges credit card full amount.
4. Job posting is made viewable to Job Seekers.
5. Recruiter 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.

2c: The card is expired:

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

3a: The card has insufficient available credit to post the ad.

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.

Figure 12.1 *A sample use case for pay for a job posting.*

responds to the use case's main success scenario, and that the story's tests correspond to the extensions of the use case.

For example, in Chapter 6, "Acceptance Testing User Stories," we saw that appropriate acceptance test cases for the story "A Recruiter can pay for a job posting with a credit card" might be:

- 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).

Looking at these acceptance tests we can see the correlation between them and the extensions of Figure 12.1.

Another important difference between use cases and stories is their longevity. Use cases are often permanent artifacts that continue to exist as long as the product is 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. While it is possible to archive story cards, many teams simply rip them up.

An additional difference is that use cases are more prone to including details of the user interface, despite admonishments to avoid this (Cockburn 2001; Adolph, Bramble, et al. 2002). There are several reasons this happens. First, use cases often lead to a large volume of paper and without another suitable place to put user interface requirements they end up in the use cases. Second, use case writers focus too early on the software implementation rather than on business goals.

Including user interface details 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 Figure 12.2, which describes the steps for composing and sending 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.

Figure 12.2 *A use case to compose and send an email 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 ini-

tiate the message instead of typing it in. Additionally, the use case of Figure 12.2 precludes the use of voice recognition as the interface to the system.

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 Figure 12.2: “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 (1999) have suggested the use of *essential use cases*. An essential use case is a use case that has been stripped of hidden assumptions about technology and implementation details. For example, Table 12.1 shows an essential use case for composing and sending an email message. What is interesting about essential use cases is that the user intentions could be directly interpreted as user stories.

Another difference is that use cases and stories are written for different purposes (Davies 2001). 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 about the users’ detailed needs.

Table 12.1 *An essential use case.*

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

Not all use cases are written by filling in a form, as shown in Figure 12.1. Some use cases are written as unstructured text. Cockburn refers to these as *use case briefs*. Use case briefs differ from user stories in two ways. First, since a use case brief must still cover the same scope as a use case, the scope of a use case brief is usually larger than the scope of a user story. That is, one use case brief will typically tell more than one story. Second, use case briefs are intended to live on for the life of a product. User stories, on the other hand, are disposed of.

Finally, use cases are generally written as the result of an analysis activity, while user stories are written as notes that can be used to initiate analysis conversations.

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 is thinking about making a career change. Since the glory days of the dot-com boom she has worked as a tester at BigTechCo. A former high school math teacher, Maria decides she'll be happier if she returns to teaching. Maria goes to the BigMoneyJobs.com website. She creates a new account with a user name and password. She then creates her resume. She wants to find a job as a math teacher anywhere in Idaho but preferably near her current job in Coeur d'Alene. Maria finds a handful of jobs that match her search criteria. The job that intrigues her most is with the North Shore School, a private high school in Boise. Maria has a friend, Jessica, in Boise whom she hopes may know someone at North Shore. Maria enters Jessica's email address and forwards the job link to her with a note asking if she knows anyone at the school. The next morning Maria gets an email from Jessica saying that she doesn't know anyone at the school, but she knows of the North Shore School and it has a wonderful reputation. Maria clicks on a button that submits her resume to North Shore.

Carroll (2000) 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 the job posting, she is considered a *secondary actor*. Unlike use cases, actors in interaction design scenarios are always people and never 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 find an appropriate job in her desired location. 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 response. Searching for a job in Idaho is an action Maria performs. The response to that action is the event of the system displaying a list of matching jobs.

The primary differences between user stories and scenarios are scope and detail. Scenarios contain much more detail and their scope usually covers multiple stories. The example scenario contains many possible stories, such as:

- a user can send information about a job to a friend via email
- a user can create a resume
- a user can submit her resume for a matching job
- a user can search for a job by geographic region

Even with their additional detail, scenarios (like stories) leave details to be worked through in a discussion. For example:

- Maria logged onto the site with a user name 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 job 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 cannot fully and perfectly 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 can be more prone to containing 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.
- Unlike IEEE 830 specifications and use cases, user stories are not artifacts of analysis activities. Rather, user stories are a support tool for analysis.
- Interaction design scenarios are much more detailed than user stories and differ in the kind of detail provided.
- 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 user stories.

Questions

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

- 12.2 What are the key differences between user stories and IEEE 830 requirements statements?
- 12.3 What are the key differences between user stories and interaction design scenarios?
- 12.4 For a non-trivial project, why is it impossible to write all the requirements at the start of the project?
- 12.5 What is the advantage to thinking about users' goals rather than on listing the attributes of the software to be built?

Chapter 13

Why User Stories?

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 emphasize verbal communication.
- User stories are comprehensible by everyone.
- User stories are the right size for planning.
- User stories work for iterative development.
- User stories encourage deferring detail.
- User stories support opportunistic design.
- User stories encourage participatory design.
- User stories build up tacit knowledge.

After having considered the advantages of user stories over alternative approaches, the chapter concludes by pointing out a few potential drawbacks to user stories.

Verbal Communication

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. We shifted focus to a shared document and away from a shared understanding.

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 the developers' interpretation of what was *written down*, which may not be exactly what they *wanted*.

Until trying it, it would seem simple enough to write down a bunch of software requirements and have a team of developers build exactly what you want. However, if we have trouble writing a lunch menu with sufficient precision, think how hard it must be to write software requirements. At lunch the other day my menu read:

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 buffaloes into promising an earlier delivery date." So, the first sentence 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 another length or must it always be 127 characters?

Writing things down does have advantages: written words help overcome the limitations of short-term memory, distractions, and interruptions. But, so many sources of confusion—whether from the imprecision of written words or from words with multiple meanings—go away if we shift the focus from writing requirements down to talking about them.

Naturally, some of the problems of our language exist with verbal as well as with written communication; but when customers, developers and users talk there is the opportunity for a short feedback loop that leads to mutual learning and understanding. With conversations 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."

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 placeholder 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, there are still two problems. First, users will refine their opinions as they learn more about the software being developed. Second, there is no guarantee that the sum of these perfect parts is a perfect whole. Tom Poppendieck has reminded me that 100 perfect left shoes does not yield a single perfect pair of shoes. A far

more valuable goal than perfect requirements is to augment *adequate stories* with *frequent conversations*.

User Stories Are Comprehensible

One of the advantages that use cases and scenarios bring us over IEEE 830-style software requirements specifications is that they are 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.

Stories take this further and are even more comprehensible than use cases or scenarios. Constantine and Lockwood (1999) 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. Because stories are terse and are always written to show customer or user value, they are always readily comprehensible by business people and developers.

Additionally, a study in the late 1970s found that people are better able to remember events if they are organized into stories (Bower, Black and Turner 1979). Even better, study participants had better recall of both stated actions as well as inferred actions. That is, not only do stories facilitate recall of stated actions, they facilitate recall of the unstated actions. The stories we write can be more terse than traditional requirements specifications or even use cases, and because they are written and discussed as stories, recall will be greater.

User Stories Are the Right Size for Planning

Additionally, user stories are the right size for planning—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 something like 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 sample requirements:

- 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.
 - 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. Prioritizing a few dozen use cases or scenarios is sometimes easy but the results are often not useful because it is rarely the case that all of the top priority items are more important than all of the second priority items. 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. Stories work well for iterative development because of how easy it is to iterate over the stories themselves.

For example, 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 progressive levels of detail, and Cockburn (2001) 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 when many people feel compelled to fill in each space on a form. Fowler (1997) refers to this as *completism*. In practice, few organizations are able to write some use cases at a summary level and some at a detailed level. User stories work well for completists because—so far—no one has proposed a template of fields to be written for each story.

Stories Encourage Deferring Detail

Stories also have the advantage that they encourage the team to defer collecting details. An initial place-holding goal-level story ("A Recruiter can post a new job opening") can be written and then replaced with more details once it becomes important to have the details.

This makes user stories perfect for time-constrained projects. A team can very quickly write a few dozen stories to give them an overall feel for the system. They can then plunge into the details on a few of the stories and can be coding much sooner than a team that feels compelled to complete an IEEE 830-style software requirements specification.

Stories Support Opportunistic Development

It is tempting to believe that we can write down all of the requirements for a system and then think our way to a solution in a top-down manner. Nearly two decades ago Parnas and Clements (1986) told us that we will never see a project work this way, because:

- Users and customers do not generally know exactly what they want.
- Even if the software developers know all the requirements, many of the details they need to develop the software become clear only as they develop the system.
- Even if all the details could be known up front, humans are incapable of comprehending that many details.
- Even if we could understand all the details, product and project changes occur.
- People make mistakes.

If we can't build software in a strictly top-down manner, then how do we build software? Guindon (1990) studied how software developers think about problems. She presented a small set of software developers with a problem in designing an elevator control system for a building. She then videotaped and observed the developers as they worked through the problem. What she found was that the developers did not follow a top-down approach at all. Rather, the developers followed an *opportunistic* approach in which they moved freely between considering the requirements to inventing and discussing usage scenarios, to designing at various levels of abstraction. As the developers perceived opportunities to benefit from shifting their thinking between, they readily did so.

Stories acknowledge and overcome the problems raised by Parnas and Clements. Through their heavy reliance on conversation and their ability to be easily written and rewritten at varying levels of detail, stories provide a solution:

- that is not reliant on users fully knowing and communicating their exact needs in advance
- that is not reliant on developers being able to fully comprehend a vast array of details
- that embraces change

In this sense, stories acknowledge that software must be developed opportunistically. Since there can be no process that proceeds in a strictly linear path from high-level requirements to code, user stories easily allow a team to shift between high- and low-levels of thinking and talking about requirements.

User Stories Encourage Participatory Design

Stories, like scenarios, are engaging. Shifting the focus from talking about the attributes of a system to stories about users' goals for using the system leads to more interesting discussions about the system. Many projects have failed because of a lack of user participation; stories are an easy way to engage users as participants in the design of their software.

In *participatory design* (Kuhn and Muller 1993; Schuler and Namioka 1993) the users of a system become a part of the team designing the behavior of 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. In participatory design, for example, users assist in the prototyping of the user interface from the beginning. They are not involved only after an initial prototype is available for viewing.

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.

Because user stories and scenarios are completely void of technical jargon, they are totally comprehensible to users and customers. While well-written use cases may avoid technical jargon, readers of use cases must typically learn how to interpret the format of the use cases. Very few first-time readers of a use case have an implicit understanding of common fields on use case forms such as extensions, preconditions and guarantees. Typical IEEE 830 documents suffer from both the inclusion of technical jargon and from the inherent difficulty of comprehending a lengthy, hierarchically-organized document.

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.

Stories Build Up Tacit Knowledge

Because of the emphasis placed on face-to-face communication, stories promote the accumulation of tacit knowledge across the team. The more often developers and customers talk to each other and among themselves, the more knowledge builds up within the team.

Why Not Stories?

Having looked at a number of reasons why stories are a preferred approach to agile requirements, let's also consider their drawbacks.

One problem with user stories is that, on a large project with many stories, it can be difficult to understand the relationships between stories. This problem can be lessened by using roles and by keeping stories at a moderate to high level until the team is ready to start developing the stories. Use cases have an inherent hierarchy that helps when working with a large number of requirements. A single use-case, through its main success scenario and extensions, can collect the equivalent of many user stories into a single entity.

A second problem with user stories is that you may need to augment them with additional documentation if requirements traceability is mandated of your development process. Fortunately, this can usually be done in a very lightweight manner. For example, on one project where we were doing subcontracted development to a much larger, ISO 9001-certified company, we were required to demonstrate traceability from requirements to tests. We achieved this in a very light manner: At the start of each iteration we produced a document that contained each story we planned to do in the iteration. As tests were developed, the test names were added to the document. During the iteration we kept the document up to date by adding or removing stories that moved into or out of the iteration. This addition to our process probably cost us an hour a month.

Finally, while stories are fantastic at enhancing tacit knowledge within a team, they may not scale well to extremely large teams. Some communication on large teams simply must be written down or the information does not get dispersed among as many on the team. However, keep in mind the very real tradeoff between a large number of people knowing a little (through written, low-bandwidth documents) and a smaller number of people knowing a lot (through high-bandwidth face-to-face conversations).

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.
- The shift toward verbal communication provides rapid feedback cycles, which leads to greater understanding.
- User stories are comprehensible by both developers and users. IEEE 830 software requirements specifications tend to be filled with too much technical or business jargon.
- User stories, which are typically smaller in scope than use cases and scenarios but larger than IEEE 830 statements, are the right size for planning. Planning, as well as programming and testing, can be completed with stories without further aggregation or disaggregation.
- 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 encourage deferring details. 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.
- Stories encourage opportunistic development, in which the team readily shifts focus between high and low levels of detail as opportunities are discovered.
- Stories enhance the level of tacit knowledge on the team.
- User stories encourage participatory, rather than empirical, design, in which users become active and valued participants in designing the behavior of the software.
- While there are many reasons to use stories, they do have some drawbacks: on large projects it can be difficult to keep hundreds or thousands of stories organized; they may need to be augmented with additional documents for traceability; and, while great at improving tacit knowledge through face-to-face communication, conversations do not scale adequately to entirely replace written documents on large projects.

Developer Responsibilities

- You are responsible for understanding why you have chosen any technique you choose. If the project team decides to write user stories, you are responsible for knowing why.
- You are responsible for knowing the advantages of other requirements techniques 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 designing what your software will do.

Questions

- 13.1 What are four good reasons for using user stories to express requirements?
- 13.2 What can be two drawbacks to using user stories?
- 13.3 What is the key difference between participatory and empirical design?
- 13.4 What is wrong with the requirements statement, “All multipage reports should be numbered”?