```
}
// still more code omitted...
```

If you were familiar with this pattern, you may have been tempted to dive right in and start writing code. Write the tests first, take small steps, and your code will be more robust.

A LOOK BACK

It is difficult to test first all of the time. Even devoted XPers will try to just knock some code out once in a while. For the most part, they end up regretting it. Ron Jeffries is kind enough to own up to this publicly in *Extreme Programming Installed*, but everyone does it.

What happens when you notice yourself coding without tests is important. It's like quietly meditating while trying not to let your mind wander. At some point your mind wanders. The strategy is to notice your mind wandering and return to meditating. What often happens is that you will berate yourself for letting your mind-wander and now you are on a path further from your goal. When you notice yourself not testing, return to testing. Programming as part of a vigilant pair will help keep you from wandering. A partner need not be a testing master to observe that you've started coding without tests and to suggest that you form your test before proceeding.

Do not test everything. Only test code that can break. In the running example you wrote code that ended up testing accessors. This was because the methods existed before they became accessors for a variable. If you had started with a variable `xxx` and methods `getXxx()` and `setXxx()` there would be little need to write tests to make certain the accessors were working properly.

The key to test-first programming is to take very small steps where nothing could go wrong. Fix what does go wrong. Move on. If you've followed the example in this chapter, you've already begun to feel good when the tests all pass. Maybe you've even come to see compiler messages and unit test messages as helping you fix your code.

EXERCISES

1. Just before moving on to test the GUI it was suggested that you change the return type of `needsToChange()` to void and introduce an accessor method `getChangeNeeded()`. This would require refactoring the tests. It's probably a bit cleaner because `needsToChange()` would then do what its name suggests and `getChangeNeeded()` would be used to find the status of the result. Complete that refactoring.
2. You've seen tests that verify that live or dead cell that gain between one and four live neighbors behave properly. Add a test to show that a live cell with two live neighbors dies when one of its neighbors dies. Create a method called `mournNewDepartedNeighbor()`.
3. Each cell should have eight neighbors that it can update when it changes state. Add this facility. Create the tests first.

Chapter 11

User Stories—

Exploring with the Customer

You're sitting in a theatre engrossed in an off-Broadway musical that follows programmers as they pair together on an eXtreme Programming (XP) project. In the climactic moment, the actor playing the client comes in to make an important announcement. Unfortunately, you can't hear it because the cell phone belonging to the man in front of you starts to ring. The play ends and you realize you'll never know whether the client was happy with the deliverables or not. You fume because there's nothing much you can do. You know you'll end up complaining to your friends about how rude people in theatres can be. On the way home, your mind replays the scene. In your imagination, you file a complaint with the cell-phone task force.

The task force immediately dispatches a sketch artist named Stanley, who asks you to describe the man who disturbed you. You say, "He was about forty years old, he had light brown hair, and he was follicly challenged." Stanley looks puzzled until you clarify: "balding." (In your daydream, you're pretty witty.) Stanley asks you to describe the shape of the criminal's head. You remember that it was long and thin. Stanley asks, "What about the nose?" Now it's your turn to look puzzled. Stanley flips to a page in his book where a dozen basic nose styles are displayed. You point and say, "Like that one, only there was a bend in the middle." Stanley sketches a bit and asks you if that's right. You say kind of, but the nostrils were a bit bigger. This session could go on quite a while, but you are jolted awake as you find yourself back home.

This dynamic between the aggrieved theatre goer and the sketch artist is the same as that between the client and the development team when user stories are first described. The client needs the help of the development team to turn the image of what the application should look like into a realistic sketch. The developers know what many of the options are, but they can't know the option that matches what the client sees. An experienced client may be able to write user stories alone, but most will need help. The client describes the application they want to the developer, and the developer asks questions and provides options to help the client with the description.

Just as the mouth that the sketch artist draws can not possibly be correct down to the last detail, the user stories that the client ends up with aren't legally binding agreements. The sketch allows the victim to say, "Yes, that looks like the man that answered his cell phone." The user stories allow the client to say, "Yes, that is what I want the application to do." The sketch artist shouldn't say to the victim, "No, the mouth looked more like this." In the same way, the development team shouldn't say to the client, "No, you really want the application to behave like this."

In this chapter, you'll listen in on a conversation between a developer and a client trying to come up with an initial set of user stories. This conversation is based on a real project with a real client for an XP software engineering class. In this case, the client was Dwight Olson, the chair of the Mathematics and Computer Science department at John Carroll University. The user stories mentioned in this chapter were actually used by the development team as they worked on this project. Because real conversations aren't as directed as examples need to be, this fictionalized account combines two meetings held early in the semester. You've been cast in the role of the developer. To give you an in-depth view of this process, you'll see all of the initial stories that came out of these two meetings.

BEFORE THE FIRST MEETING

Your role in the first meeting is similar to that of a reporter. You are interested in the story that the customer has to tell, but you need to help the customer get it into a form that makes sense to your readers. As with the reporter, you as a developer will learn more if you do a little research first. Think back to interviews you've heard or read in which the interviewer asks a simple question that could have been easily looked up ahead of time. You don't want to hear an interviewer ask, "Where did you grow up?" or "When were you born?" when this information is readily available. It is better to ask, "As a teenager in Seattle in the late sixties, what influence did Jimi Hendrix have on your music?" There are, of course, interviewers who go too far in the other direction. They will ask questions that are intended to show their knowledge rather than to elicit information from their subject. Neither extreme is helpful. While meeting with the client, you will need to have enough knowledge to be able to ask relevant questions and to make helpful suggestions while letting the client tell the story.

There are some things you should probably know about Dwight and this project. Each semester he has to figure out what courses his department will offer and who will teach them. He'd like a software application that automates the part of the process that amounts to just copying information from past semesters and facilitates filling in the schedule with the new information. He has a vision of how this software should look. By the way, this isn't the first time he has agreed to be a client for the software engineering class. Twice before, he has worked with a group trying to build this scheduler application. Once the application was not completed and the other time the final version was too limited and too difficult to use.

Dwight is actually interested in working with an XP group. He's heard about the differences between the traditional methodology that he has experienced on past efforts and XP. He understands that more will be expected of him and he is willing to be

available throughout the semester. This is an important fact to keep in mind. You don't have to get a complete and detailed picture in the first meeting. You will be able to go back to Dwight and clarify areas you don't completely understand. On the other hand, he'll be able to contact you and change his mind about features that he was sure that he needed.

WRITING YOUR FIRST USER STORY

It's a week into the semester; you're still working on your spike to try to figure out what XP is all about. You don't really start this project for another week, but you'd like to get a look at what you'll be doing for the next twelve weeks or so. The beginning of the semester is a busy time for a department chair but Dwight welcomes you into his office, clears some papers off a table, and invites you to sit down. The two of you chat a bit about how the semester is going. You may be anxious to get down to business, but you remember that XP is all about communication and relationships. You also remind yourself that he is being nice enough to give you a lot of time this semester despite his previous experience of getting nothing usable in return. You absentmindedly play with your stack of blank index cards while enjoying the conversation.

Taking Stock of Artifacts

Dwight pulls a folder from his top drawer and says, "Here's what I'm looking for." He takes a couple of sheets out of the folder and shows you schedules from past semesters. On one sheet, the courses are in order by course number and section. The course title, faculty member, and class-meeting times are included. On another sheet is a list of faculty members with course numbers and meeting times. For Dwight, there is an obvious metaphor for this software you are creating. Your computer might use folders and files so that a filing cabinet is a metaphor for how you store documents on your computer. Here Dwight is suggesting how he thinks of your project. It will help you and your team to use the language that fits with his view. The sheets of paper that he is showing you are the artifacts that he associates with the course-scheduling process.

"Each semester," he explains, "I have to look at the courses we need to offer and schedule the faculty members who'll be teaching them. Take first semester Calculus as an example. Looking at the number of incoming freshmen, we'll have to offer a certain number of sections. We'll need to spread them out during the day so that students can fit them into their schedules along with their required courses for other majors."

Perhaps it occurs to you that the course schedule is more or less the same from year to year. You ask Dwight, "Are you really creating the schedule from scratch each semester?"

He thinks a moment and answers, "Well, the fall and spring semester are different. We run more Calc 1 classes in the fall and just a few in the spring. With Calc 2, it's just the opposite; more classes run in the spring than in the fall. But from year to year the schedule is basically the same."

"So," you suggest, "couldn't you start a semester's schedule by modifying the schedule you had for the same semester the previous year?"

"That would work for the one hundred and two hundred level courses. But some of the upper level and graduate courses rotate on a two-year schedule. For those courses, it might be better if we started a new schedule document from the schedule for the same semester two years before," he replies.

Write a User Story

You've heard a clear description of a required feature for the scheduler. You write the following description on an index card.

Create Document

When creating a document for a new semester. The classes, sections, times, and Faculty members assigned to teach the classes from the same semester two years before are used as the starting point.

You could record these user stories using a word processor or spreadsheet. Index cards are less permanent. However, you can easily pass an index card to Dwight and he can make changes. If an idea comes to him while you're writing a card, he can take a blank one and write down his own user story.

You read the user story back to Dwight and ask him if that's what he'd like. He thinks a moment and says that it is mostly right. He doesn't think he wants the faculty members included in the new schedule. While you're making the change, you replace the word "classes" with "courses." It shouldn't matter to Dwight, but there will be confusion if classes could refer to both Java classes and to the classes taught by the department. Even though you are making small changes, you read him the revised user story. A recurring theme of XP is short feedback loops.

Create Document

When creating a document for a new semester. The courses, sections, and times from the same semester two years before are used as the starting point.

He nods and says, "That sounds good." You hand him the index card. It's only fair: he showed you his artifacts so you show him yours. You need to explain user stories to him.

"All I've done," you begin, "is take what I think you said and create what we call a 'user story.' It has a short title, 'Create Document,' that helps us identify it. The description on the card then serves as a common understanding of this feature. I may have to ask more questions later when we start coding it up, but for now I think I understand what you're saying. There is one other requirement for a user story—it has to be testable. There has to be a way for you to test whether or not we've actually given you what you want. In this case, it's pretty easy. You create a new document and you should see the information from two years ago."

Remember that this is as new a process for Dwight as it is for you. It might help both of you if you explain to him how these user stories will be used. "Once we have a bunch of user stories," you continue, "the team will estimate how long we expect each one to take and how much time we have available. We'll ask you to pick the ones you want first. You get to decide what's most important to you at any time. You might decide this story isn't important to you this week and next week decide it's the one you must have next. You can even decide that you no longer want the application to work the way you described. You have total control of the stories. If you'd like, you can write them."

Dwight smiles and says, "That's okay. You can keep writing them."

STORY GATHERING

Now that you have your first story, you can ask questions to help the client come up with more stories. Try to let the client lead. You may lead the client to many stories but they won't necessarily be the client's priorities. Ask open-ended questions. Don't ask, "Would you like the background to be blue?" Instead ask, "What would you like the panel to look like?"

Find a Starting Point

Dwight has told you what he wants to happen when a new document is created. In this case you can follow the process from the beginning. At other times you may want to begin at the end and work backwards. In this case, you ask, "What would you like to happen when you first start up the application?"

"I don't know."

"Do you want a screen to pop up with various choices?" you prompt.

"No," he answers, "most of the time I'll just be working on an existing schedule. I think I'd like the program to just bring up the last schedule I was working on."

You write your second user story and read it back to him.

Starting Application

The application begins by bringing up the last document the user was working with.

Dwight thinks that user story sounds pretty good. You point out that that means there has to be a way of saving the document.

Following a Path

The story you wrote for starting up the application has led to a story on saving the document. The danger in following this path is that you haven't even decided what goes in the document. The danger in not following this path is that you'll forget the features

that are popping into your mind right now. Continue brainstorming. Write the story for saving the document now.

Saving Document

User saves the document when they want.

Again, Dwight is happy with this story. He adds that he wants to make sure he's given a chance to save his work if he accidentally closes the application. You change the story like this.

Saving Document

User saves the document when they want. User is prompted to save when he closes the application.

Dwight frowns and says, "No, I don't think those two belong together. Go back to the original story for Saving Document and create a new story for Closing Document." You discard the expanded Saving Document story and create the following separate story.

Closing Document

If user closes Application, they are prompted to save.

Dwight is now happy with the separate user stories. Looking back, you've actually covered a lot of ground pretty quickly. He knows what he wants to happen when the application starts and when the application quits. He also has described when he can save his work and how he can start up a new document.

Going Too Far

You're on a roll. You've suggested some solid user stories that have the right amount of granularity. At this time, it might be nice to experiment with the schedule, perhaps try a few things out and decide which looks best. You create the following user story.

Experimental Changes

If the user wants to play around with the schedule without erasing the current state, they can. At any point, they can choose to revert to the earlier version, to accept these changes as the current version, or to create another experimental version.

You proudly read it to Dwight. He politely explains that he doesn't really see the need for it. You agree that he probably doesn't need it right away but suggest that you

put it in a pile of ideas to be implemented later. Dwight says, "No, I don't really think I would ever use that feature."

What went wrong? It was all going so nicely. Experimental Changes describes a very nice feature. It might be one that you would find very useful. Usually these moments happen when the development team is working and the client is nowhere in sight. Someone on the team thinks of a cool feature to add to the program and takes a few days to add it. The client doesn't want the feature and will never use it. In XP, the client is always allowed to make these decisions for himself. So really nothing went wrong. You had an idea and the client made the decision that he wouldn't use this feature. Sometimes it's hard not to take these rejections personally. Dwight didn't say the idea wasn't good; he said it wasn't for him.

GETTING BACK ON TRACK

Although there is nothing wrong in suggesting features and certainly nothing wrong with the client rejecting your suggestions, you will have more success at this point in the project if you can let the client suggest the features that are important to him. One way to get back on track is to return to the beginning. You've written the story called Starting Application. Pick up there and ask the client, "When the application starts up and loads the last schedule you worked with, what do you want to see on the screen?"

Dwight picks up the first piece of paper he showed you earlier and says, "Something formatted like this one." You write this as a story and show it to him.

Scheduling View

The scheduling view lets the user see the current state of the schedule by course number. This view shows course number and name, section number, credit hours, meeting times, and instructor's name.

Dwight likes this story. It captures what he's looking for. It also suggests other user stories.

Exploring Time Slots

You ask Dwight what he means by meeting times and how he wants to enter them.

"That's an interesting question," Dwight says. "The answer is pretty complicated. There are set time slots for classes that meet Monday, Wednesday, and Friday (classes are 50 minutes long and meet on the hour) and set times for classes that meet on Tuesday and Thursday (classes are 75 minutes long and meet at 9:30, 11:00, 12:30, etc.). Some of our classes meet four days each week so the time may be different on some days than others. For example, if a class meets four days a week (MTWF or MWThF) at nine or noon, the Tuesday and Thursday class times are automatically adjusted. The nine o'clock class moves to 8:30 and the noon classes move to 12:30. Some of the time slots are only available to three credit-hour classes. Also, I can create special time slots late in the afternoon or in the evening for graduate courses."

"As for entering them," Dwight continues, "I'd like to be able to click on a button and have all of the possible times listed so I can just select the one I want. Also, I'd like to be able to look at all of the classes scheduled at a particular time to see if I have conflicts."

"Hold on," you say, scribbling wildly, "here are the stories I think you're describing.

Time Slots

A time slot consists of the days the class meets, together with the time. The application should be aware of standard time slots and allow the addition of nonstandard slots for the current semester.

Drop Down List of Blocks

When choosing a time slot a drop down list should help with user selection.

Credits Determine Blocks

A Three-hour class can only be offered in set slots.

Four Day Class Adjustment

In a class that meets four times a week at 9 a.m. or at noon the Tuesday/Thursday times are adjusted. The 9 a.m. class moves to 8:30 a.m. and the 12:00 class moves to 12:30.

Viewing the Time Slot Schedule

The user should be able to view all classes currently scheduled for a given time slot.

Dwight looks the list over and says, "There are also slots that aren't available for certain classes. I can't schedule a freshman class at certain times because of the freshman seminar, for example." You write this.

Section Rules

The user can define rules so that courses can not be scheduled during certain times.

Dwight looks at it and says, "Put that in the pile of stories to be done later. I'd like it eventually, but it doesn't need to be one of the first things you do." You agree and look at where you are. You now have a good sense of how the time slots work in the scheduler, but you need to know more about the other entries.

Courses and Sections

Dwight points out that you haven't really dealt with courses and sections yet. The first semester course in Calculus may be offered at different times or by different instructors. These are called sections. Dwight says, "Even though this document starts with the corresponding semester from two years earlier, I'm going to need to make adjustments. I'll need to be able to change the list of courses being offered. I'll have to decide how many sections of each course we're offering. Then I'll need to be able to choose a time slot for the section."

You present him with the following cards.

Which Courses

The user should be able to add courses to or delete courses from the schedule.

Assign Number of Sections

For each course, the user can select a certain number of sections.

Schedule Section

A user should be able to schedule a section by selecting a time slot.

This last story seems close to the drop-down list of blocks story, but that one was about how a time slot is selected and this one is about being able to assign time slots for sections.

Then Dwight says, "You know what I really want? I want this program to figure out the section numbers automatically. For each course, we assign section numbers with this complicated routine. For day classes we start with the number 50 and number the courses in order early to late on Monday, Wednesday, and Friday followed by the courses early to late on Tuesday and Thursday. There's a whole system. Then we start numbering at 1 for the evening courses starting with the Monday and Wednesday classes and then the Tuesday and Thursday classes."

You sketch the following card.

Number Sections

The application should be able to generate section numbers for each course. The sections are numbered as follows. Day classes start with 50 in order MWF early to late then TTh early to late then MTWF early to late then MTWTh early to late then MWThF early to late. Night classes start with 1 in order MW early to late then TTh early to late.

"That's it," he says. "I'd also like to be able to see when a course is offered during the week. I'd also like to look over the whole week for all the courses" He accepts these stories.

View Course Distribution

A user should be able to see when in the week a given course is offered to make sure that it is well distributed.

Viewing Entire Schedule

The user should be able to view the entire week and see what classes are offered in each time slot.

Working with Faculty

Based on Dwight's description, you come up with these stories to specify how faculty will be treated in this program.

Faculty List

User can add or subtract faculty members or list them as inactive.

Teaching Preferences

User can view list of faculty appropriate for teaching a given course.

Teaching Assignments

User can assign an instructor to a given section.

View Faculty Schedule

A user should be able to select a view showing a faculty member's schedule by week and by class.

Teaching Assignment Errors

The user is notified if an instructor has a scheduling conflict when it occurs.

WRAPPING UP WITH THE CLIENT

You've gotten a lot accomplished in this meeting. You don't have the entire application specified down to the last detail. You couldn't possibly do so with any degree of accuracy. As the client sees what you're building, his understanding of what he will be getting will adjust and the requirements and, therefore, the user stories will change. You need to close the meeting by making sure the client understands where he is in the process and what's coming next.

Stopping the Client

Take the stack of user stories and pass it to Dwight. Ask him to look through them and see if anything jumps out that is missing or needs to be changed. Be careful how you phrase the question. You don't want to encourage the client to keep finding one more story. If the client interprets your question as saying that you're looking for more stories, he may try to come up with some, thinking that he's helping you. If you sense this happening, you need to explain that you're just making sure that there's no really important feature that he just hasn't thought of yet.

You look at the stack and realize that you already have plenty to work with. You can probably already see that it won't all get done this semester. After looking at the existing stack, Dwight may come up with a story that is more important than many of the existing stories. You can help take the pressure off of him by reminding him that new stories can be added at any time during the life of the project. Just as your job at the beginning of the meeting was to encourage your client to start creating stories, now your job is to help the client stop.

Ordering the Stories

Now suggest that the client put the stories roughly in order of importance. Not all stories will be equally important. If the client has difficulty you can remind him of the two stories that were already ordered: one he decided would never need doing and the other could wait. Maybe the first ordering can be in rough piles labeled "right away," "next," "later in the semester," "might not be delivered," and "not this semester." It's easier to start with this rough sort. These categories are tied to the fact that you will be abandoning this project at the end of the semester. Ordinarily a project continues until the client says "stop."

Next, ask him to go back to the "right away" and "next" stacks and put those stacks in order of importance. You need to stress that nothing is written in stone. At any point in the project Dwight can reorder the stories. There may be a cost in reordering, but he has the right to decide what is worth that cost. As he orders the two piles, he may pull stories from "next" and put them ahead of "right away." This is great. The client is handling the user stories and getting a feel for how he can control the project. He is reconsidering priorities in a way that will help your team later when you need quick answers from him. Let the client know that before you write any code you will be back to allow him to pick the stories he wants you to work on. Once the cards are in order, you can explain the planning game.

Preparing the Client for Reality

One of the keys to the relationship between an XP client and a development team is that the client's expectations are based on reality as soon into the project as possible. At this point, however, Dwight has no clear picture of the process. You realize this when he hands you the stories from the "right away" and "next" stacks and asks, "Can I have these by next month when I start working on next semester's schedule?"

"I don't know," you say with confidence.

"Well," he presses, "how much do you think you might be able to give me by then?"

"I don't know yet," you answer again, "I don't have any guesses of how much work is involved and how much work we can do."

Dwight looks a bit puzzled. After all, you've been asking him to write concrete stories and to order those stories. He's just asking you how long they might take.

You explain, "I'll take these cards back to the team and we'll estimate the cards that were in the 'right away' and 'next' piles. We might estimate more of them if that will help. Then we'll estimate how much time we can spend in the first two weeks. The estimates will be in units that probably stand for some amount of time, but we can't be sure what yet."

"Let me get this straight," Dwight responds, "You may take the Create Document story and say it takes five units but you aren't saying whether the units are hours or days or whatever. I'm not sure that I see the point of estimating if the units can mean anything."

"Okay," you say, " Let's stick with your example and let's say we also estimated Closing Document at one unit. If it actually takes us about three hours to finish Closing Document, how long should it take us to do Create Document?""

Dwight smiles. "Then Create Document should take around five times as long or about fifteen hours."

"Right," you agree, "and as you complete more stories you'll have a better and better idea of what a unit corresponds to. Actually, you don't care."

"What do you mean, 'you don't care'?" he asks.

"Well at the same time, each member of the team estimates how many units he can accomplish this week. If I estimate that I can do twenty units this week and end up only doing Create Document then I've accomplished five units of work this week. Even if I do Create Document and most of Closing Document I've only completed five units of work this week. This makes next week easy. I sign up for five units of work again and see how I'll do."

"So that's why you don't care about units," Dwight says. "Whether a unit corresponds to an hour or a day, you don't care. You know how many units you can accomplish in an iteration and you know about how long it would take you to finish the user stories."

"Right," you continue, " and our estimates will improve during this project because we'll have more experience and much more data."

"Okay," Dwight says, "I get how you are using units. What happens after you estimate all of the stories and tell me how many units you can do this week?"

"Next, we'll bring the stories back to you. We might tell you that we think a story is too big and ask you to split it into smaller stories."

"How much smaller?"

"Well, one of the keys to XP is getting quick feedback so we don't want a story that we think will take too long to finish. These stories look pretty good, though."

"So," he says, "you tell me what the stories cost me in terms of units and how many units I get to spend."

"Right," you say, "we'll tell you how many units you can pick for the first iteration and you can decide which stories you'd like us to work on first. If you get twenty units of work from us, you pick out twenty units worth of stories."

"So," Dwight asks enthusiastically, "when I pick out these twenty units worth of stories, that's what I can expect at the end of the iteration?"

"Maybe," you reply.

"What do you mean, 'maybe'?" he asks. "You tell me to choose twenty for the first iteration, I choose twenty, and now you say that maybe I'll get them and maybe I won't?"

"Well, remember," you caution, "we aren't very good at this yet. We don't really know how much we can accomplish in a week and we really aren't very good at estimating stories yet."

"So what can you do this first iteration," Dwight asks.

"Well, our next step will help us estimate this even better. After you choose the stories for the first iteration, we then go break them down into what we call tasks. These are the programming jobs it takes for us to program your story."

"That's like me telling you I need to run out for an hour to buy office supplies but then I break it down to driving there, shopping, and driving back."

"Right, and you know it takes you about twenty-five minutes to drive there, twenty-five minutes back, and you usually spend fifteen minutes in the store. So now you have a better idea of how long it might take. You now are guessing that it will take you an hour and five minutes."

"But," says Dwight, "it's still a guess."

"That's right," you say, "but it's based on more detail. Say you get caught in traffic on the way there and it takes you forty minutes to get there. You could call me and say that you're running late. You still have the fifteen minutes of shopping and the twenty-five minute drive back so you can now give me a better estimate. You don't have to wait until you get back to realize that you're going to take longer."

He thinks a minute and then says, "so once you break down the story into tasks do you give me new estimates based on the total of the tasks."

"Maybe," you say. "If the new totals mean that we can't get everything done that we estimated, we'll come back to you and let you choose which of the reestimated stories you want us to work on during the first iteration."

"And then you go work on the twenty units of stories."

"Right, but if we get caught in traffic we'll be back."

"Why?"

"In your shopping example, what if it's more important that you get back in an hour than that you get your office supplies? After twenty minutes of driving in traffic you may realize that you're not going to make it to the store so you change your plan and turn around and come back here."

"So," says Dwight, "if you find a story took longer than you think then you'll know that you can't complete all the work we agreed on."

"Right. But we don't get to choose what's most important out of the remaining stories. This means that we have to come back to you and ask you to choose. Maybe what we now know is that we only get sixteen units done during each iteration and we've completed eight. We'll ask you to choose eight units worth of stories out of the remaining twelve."

"So," Dwight says, "it sounds like I will be making a lot of decisions."

"You will, and they won't all be eliminating stories. The second iteration often goes better than the first so we may come back to you and tell you that you can pick additional stories."

You and Dwight have covered a lot at the first meeting. You established a rapport. You listened as he described his project. You helped him turn his description into user stories that will help both the team and the client understand what is being built. You helped him prioritize the stories, and you prepared him for the future. These are your goals for this first meeting. You wrap up the meeting and take your stack of user stories back to your team.

CLIENT VARIATIONS

Although Dwight was a real client, he was almost too good to be true. Your clients may have a more difficult time with the process. They may not understand what they are agreeing to. At the very least, your client must have a realistic view of what they are asking for.

Much of your interaction with the client will involve your explaining to them what is coming next and coaching them on what you need. You have to explain what a user story is and you need to provide examples. If a client provides you with a bad or inappropriate user story you need to be able to help him or her see why it may need improvement.

Consider a user story that specifies that the application must be extremely easy to use. How would you test this story? How would you know when it is completed? What are the steps along the way? Be careful how you discuss this with your client. He or she may be very proud of this story. Ask what is meant by easy to use. Maybe you can suggest menu options or buttons that will bring up the last schedule being worked on.

Another weak user story may specify that you use a javax.swing.JTable to present the schedule information. This may be a great idea, but the client does not get to make technical decisions. You may say that you'll consider that as an option, but then ask which features of the JTable are important to the client. The client gets to make many decisions in this process, but technology is generally not one of them. An exception is that the client may want to specify that your application be able to read in information generated from a spreadsheet program that they like to use. Then this is a legitimate user story.

Generally, your client will try to cooperate with you in this setting. They have agreed to help on a project and are interested in their software and your education. The biggest requirements of the client in an academic XP project are availability and the ability to make decisions. You need to be able to contact the client when problems arise. If they are off campus for two months then this won't be possible. When you contact them you are often asking them to choose among alternatives. They need to make quick decisions so that the project can move on.

Chapter 12

The Planning Game—

Negotiating the Future

After testing first, the planning game may be the hardest practice in XP. It wouldn't be so hard if we were in the habit of telling the truth in estimating software practices. Of course, developers will say that they would be more honest in their estimates if management would treat these estimates in a reasonable way. So, developers learn to pad their estimates and management learns to demand that the estimates be cut down. Meanwhile, the sales staff is presenting their own version of the schedule and requirements to the customers. The weary customer just wants the truth. Customers need to build their own business plans around the delivery of your software. Like the Tom Cruise character in the movie *A Few Good Men,* customers repeatedly ask for the truth and tell you that they feel they are entitled to it. Often the reaction of the development team echoes the response of the Jack Nicholson character in the movie, "You can't handle the truth!"

In XP you will tell the truth. You will tell the truth early and often. Even a disbelieving customer will come to trust you quickly because you are backing the truth up with reality. Every two weeks the customer gets to see what you've produced. After a couple of iterations your estimates get better, and the customer comes to trust that you mean it when you say that your best guess is that you'll get a certain amount done. There aren't many rules in the planning game. Here's the first.

Rule 1: The developers will be truthful in their estimates and the customers will believe these estimates.

This sounds like business as usual. After all, back in the Chapter 1 you were told about projects that promised a delivery date. The project team then set out to do the analysis, and then the design, and then the programming. Some time into the programming cycle, the developers realized they weren't going to make the delivery date. Usually a little bit before the deadline a quiet announcement is made that the delivery date will be

missed. In his book *Slack*, Tom DeMarco argues that the fault is with the schedule and not with the workers. He writes, "A bad schedule is one that sets a date that is subsequently missed. . . . If the date is missed, the schedule was wrong. It doesn't matter why the date was missed. The purpose of the schedule was planning, not goal-setting. Work that is not performed according to a plan invalidates the plan."

This statement seems to fly in the face of how missed deadlines are dealt with. Isn't it the fault of the developers? It has to be someone's fault. DeMarco continues, "The missed schedule indicts the planners, not the workers. Even if the workers are utterly incompetent, a plan that takes careful note of their inadequacies can help to minimize the damage. A plan that takes no account of realities is not just useless but dangerous." In the planning game you will never plan too far ahead because the standard errors, in the technical statistics sense, are initially so large that these estimates are useless. You will estimate what you can get done in the next two weeks based on what you know of your abilities in the past during this length of time. Just as your estimates will be based on more data points as you progress, the customers will tend to have a better sense of what to expect in the next iteration as more and more iterations pass. This is summarized as the second rule of the planning game.

> *Rule 2:* The developers will refine their estimates and the customers will refine their expectations based on the actual achievements in each iteration.

Many popular games can serve as a metaphor for the planning game. Consider any card game in which some cards are revealed and then discarded; Bridge, Blackjack, Poker, and Gin Rummy are some obvious examples. Before the game begins, the chance of any given card being an ace is one in thirteen. At a later point, if a player has carefully watched the discards and knows that all the aces have already been played, he or she knows the probability of any future card being an ace is zero. By paying close attention to what has happened in the past, the player can better assess the current possible outcomes.

This is your strategy for success during the planning game. Carefully observe what's happened in past iterations and use this knowledge to predict the outcome of this iteration. A further analogy to the planning game can be made using the board game Monopoly. In Monopoly, players take turns rolling a pair of dice and moving around a board that has properties and special spaces. If you land on unowned property, you can purchase it from the bank with money you've accumulated. If you land on property owned by another player, you have to pay them rent. You can buy and trade property with other players in an effort to control all properties that share a given characteristic (i.e., same color, other railroads, or both utilities). You may have a strategy, but this strategy must be adjusted as others land on and buy property that you wanted. Each time that it is your turn, you must reassess and make your plans based on all that you know about the current state of the game. If you land on a property that costs $280 and you only have $220 then you can't buy the property without selling or mortgaging some of your other assets. At each point you have a fixed amount of resources and have to judge what you can do with them.

This is true in XP's planning game as well. In each iteration there are a fixed amount of resources. You will have a better idea of the true value of these resources as

the game goes on. As in Blackjack, you will make better guesses as to the value of the hidden cards later in the deck.

What's interesting about the planning game is that the developers get to estimate their assets and the costs of potential activities. The client, however, decides how the developer spends these assets. It's as if the developers and client are part of this team that is taking their turn once per iteration. The developers, however, possess the information that the client needs to steer the development process. If the developers notice something during the iteration that needs to be shared with the client, then they should do so immediately. The sort of things that the developers need to share include informing the client that they are working more slowly than expected, that they are working more quickly than expected, or that a certain user story was underestimated or overestimated.

There's one last rule that also is related to game playing. The best game players are constantly updating their view of the game and reconsidering the options available to them. In some games (for example, the card game Set or the word game Boggle), as soon as you see a potential move, you are allowed to make it. Simply stated, Rule 3 says that you don't have to wait until it is your turn to take your turn.

> *Rule 3:* During the iteration the developers will update the client as to the progress of the iteration. The client will use this information to quickly refine what is required in the current iteration.

Notice that at each point the client makes decisions based on the information that the developers provide. There must be trust in both directions. That isn't to say that clients should close their eyes and fall backwards, trusting that the developers are there to catch them. The clients can start out by trusting the developers a little. The developers then show this trust is well founded and also that they trust the client just a little as the client begins to make decisions. It may not begin smoothly, but XP has built in checks so that the discussion can proceed in a practical and not a personal way. Consider the following interaction.

The client says, "Oh, you say that costs four units and that costs three and that costs three and that costs two. Hmmm. How much do I have to spend? Six units? Okay, I think I want the one that costs four and one of the ones that costs three." It's not that the client can't add. Traditional clients always ask for more because they suspect the developers will always deliver less.

There's no reason to fight. The developer smiles and says, "You can spend six units."

The client says, "Oh, that's right. I would like the two 3's please."

And that's the planning game. Except for the part where the developer comes back and says, "Oops, one of those threes is a four." Or the developer might say, "We've found some extra time, you can choose two more units."

> *Note:* This chapter, like all others in Part III, is a tutorial. Rather than just read it through, you should stop when directed and participate in the activity. It is unlikely that you will come up with the same answers as those provided in subsequent sections, but these answers will have more meaning for you if you play along at home.

THE BEGINNING USER STORIES

Check out the User Stories tutorial in Chapter 11 for a look at how user stories are discovered and developed. In this chapter, we'll use actual client user stories from a tic-tac-toe game. In this case, Steve wanted a tic-tac-toe game designed for his daughter to play on their home computer. Steve came in with the following eighteen user stories. Read them carefully and think about whether you have any questions about these stories.

You will be tempted to use spreadsheets and other software solutions for managing your stories, tasks, and other aspects of the planning game. Traditionally, these are kept on 3 × 5 index cards. It's not that the developers and pioneers of this methodology couldn't write applications to manage this process. They found that, for colocated teams, index cards work best. They can be passed around, reorganized, and torn up with ease. They are inexpensive, easy to use, and several people can create new cards at the same time.

Default Pieces (1)

Default representations for the pieces are a graphical "X" and a graphical "O."

Board Setup (2)

The board is a 3 × 3 grid. Each square on the grid is initially empty.

Winning the Game (3)

A player wins the game when three of their markers form a line either horizontally (row), vertically (column), or diagonally.

Tie Games (4)

If nine pieces have been played and there is no winner, the game ends in a tie.

Player's Pieces (5)

Tic-Tac-Toe is played by two players. One player is assigned the marker "X," the other is assigned the marker "O."

Standings (6)

A running tally of the number of wins for each player during a session will be kept and displayed.

Displaying a Move (7)

Once played, a visual representation of the piece will be displayed on the screen.

Highlighting the Win (8)

When a player wins, the winning three-in-a-row is highlighted by a line drawn through the three markers.

Taking Turns (9)

Players alternate placing their markers in empty squares. Once placed, a marker can neither be moved nor removed.

Making a Move (10)

Players select the square to place their marker by clicking on the square with the mouse.

Number of Games (11)

A running tally of the number of games played during a session will be kept and displayed.

Display Results and Replay (12)

At the end of a game, the result will be displayed and an offer of another game will be presented.

Alternate Markers (13)

Players can select alternate representations for their markers from a list or supply their own.

Illegal Moves (14)

Attempting to select an illegal move results in no action taken.

Computer First (15)

The player may select whether the computer plays first or second.

Computer Ability (16)

The computer played has two levels of ability: low—makes random plays; high—never loses.

Computer Opponent (17)

Player may choose to play against another human or the computer.

Quitting **(18)**

The session can be ended by selecting the quit button.

Stop: Before going on, take time to think through these user stories. If you were meeting with a client, what questions might you have about these stories? What clarifications or changes might you be looking for?

REFINING THE USER STORIES

The first step in the planning game is to make sure that the user stories are clear and testable. Here are the actual concerns that the developers had for Steve. In this case there were actually two developers, Kathy and Jason. To make it easier to follow this part of the conversation, we'll combine the developers into the single persona of Kathy.

Kathy starts by saying, "I'm not sure I understand the difference between user stories (9), Taking Turns, and (10), Making a Move. It sounds as if they're both about moving by placing markers in empty squares."

Steve responds, "I was thinking that Taking Turns was more of a description of what constitutes a move whereas Making a Move describes the physical action of making the move, but I can see where there's a lot of overlap."

"In fact," Kathy quickly adds, "I think we can throw story (7), Displaying a Move, into this combination as well. If you're placing a marker in a square then this requires a visual representation of the piece being placed in the square."

"Again," says Steve a bit defensively, "that's probably true but at the time I thought of them separately."

"Should we keep them separate?" Kathy asks.

"No, we can combine them," Steve agrees. He jots down the following user story.

Make Move **(19)**

Players, alternating turns, use a mouse to select an empty square in which to place their marker. Once placed, a marker can't be moved or removed.

"What do you think?" Steve asks.

"It works for me. Do you think you need to say something about the visual representation of the marker?" Kathy answers.

"I don't think so," says Steve. Remember, the user story is more of a record of their common understanding than it is a binding agreement. At this point, both Steve and Kathy agree that this story stood for the three stories it was replacing.

Kathy continues to look at the cards. Something seems to be missing. She says, "I don't mean to create more work for my team, but do you want to allow the players to enter their names?"

Steve agrees. He adds the following user story.

Player Names **(20)**

The players will be asked to enter their names at the beginning of the session.

Kathy nods and then asks, "What about stories (1), Default Pieces, and (5), Player's Pieces? They both talk about the players' markers being an 'X' and an 'O'."

Again Steve agrees. He thinks for a minute and then suggests replacing them with this user story.

Choose Marker **(21)**

Tic-Tac-Toe is played by two players. At the beginning of a session, one player chooses either "X" or "O" as his or her marker, the other player is assigned the remaining one.

"That's better," said Kathy. "What about number (14), Illegal Moves? Don't you think it would be better if the user was told that a move was illegal? If no action is taken, the user might be confused."

At first Steve thinks that this would just complicate the game and then he sees Kathy's point that a user could be confused without further feedback. He replaces Illegal Moves with this refinement.

Illegal Moves **(22)**

Attempting to select an illegal move results in a message indicating that the move is not allowed.

"I think we're going to need to change (8), Highlighting the Win, as well," says Kathy.

"Why?" asks Steve.

"This is a little more subtle," Kathy replies. "If you look at the story, you have us making the line through the three markers when a player wins."

"That's what I want," says Steve.

"Well yes," Kathy answers, "but really as soon as three markers of the same kind are in a row you can draw the line."

"That's true," says Steve, "but I don't really see the difference."

Kathy answers, "The problem lies in the acceptance testing. If you want to test that the user story has been fulfilled, in the second version you just need to see what happens if three of the same markers are in a row or not. The acceptance test there can avoid the issue of whether the program 'understands' that the game has been won. In the first version you have to tie the logic of your acceptance test to whichever object is in charge of determining a win. The second way is cleaner and can be more easily tested."

Steve asks, "Isn't this coding? It seems like the decision is yours."

Kathy says, "It's somewhere in between. I'm making a coding argument for why you may want to change how you've described your story. Not only that, but it will

allow the acceptance tests that you write to more accurately reflect the status of the development."

"How do you figure that?" asks Steve.

"Look at it this way, if the acceptance test for user story (8), Highlighting the Win, depends on knowing when the game has been won, then it cannot possibly pass until story (3), Winning the Game, also passes. So if (8) is implemented before (3), its acceptance test cannot pass until (3) is implemented, at which point both acceptance tests will pass. It's a question of granularity, the line should bedroll when there are three consecutive markers, testing for a win is a separate issue."

Steve agrees and writes this replacement story for Highlighting the Win.

Highlighting a Three-in-a-Line **(23)**

Whenever three markers of the same type are in a line, draw a line through those markers.

Steve quickly adds, "You know, we should change story (3), Winning the Game, to point out that after someone wins, the game is all over."

Recognizing a Win **(24)**

A player wins the game when three of their markers form a line either horizontally (row), vertically (column) or diagonally, and the game ends.

At this point Steve and Kathy agreed that they had enough stories to begin. They were:

Board Setup **(2)**

The board is a 3 × 3 grid. Each square on the grid is initially empty.

Winning the Game **(3)**

A player wins the game when three of their markers form a line either horizontally (row), vertically (column), or diagonally.

Tie Games **(4)**

If nine pieces have been played and there is no winner, the game ends in a tie.

Standings **(6)**

A running tally of the number of wins for each player during a session will be kept and displayed.

Number of Games **(11)**

A running tally of the number of games played during a session will be kept and displayed.

Display Results and Replay **(12)**

At the end of a game, the result will be displayed and an offer of another game will be presented.

Alternate Markers **(13)**

Players can select alternate representations for their markers from a list or supply their own.

Computer First **(15)**

The player may select whether the computer plays first or second.

Computer Ability **(16)**

The computer played has two levels of ability: low—makes random plays; high—never loses.

Computer Opponent **(17)**

Player may choose to play against another human or the computer.

Quitting **(18)**

The session can be ended by selecting the quit button.

Make Move **(19)**

Players, alternating turns, use a mouse to select an empty square in which to place their marker. Once placed, a marker can't be moved or removed.

Player Names **(20)**

The players will be asked to enter their names at the beginning of the session.

Choose Marker **(21)**

Tic-Tac-Toe is played by two players. At the beginning of a session, one player chooses either "X" or "O" as their marker, the other player is assigned the remaining one.

Illegal Moves **(22)**

Attempting to select an illegal move results in a message indicating that the move is not allowed.

Highlighting a Three-in-a-Line **(23)**

Whenever three markers of the same type are in a line, draw a line through those markers.

Recognizing a Win **(24)**

A player wins the game when three of their markers form a line either horizontally (row), vertically (column) or diagonally, and the game ends.

Stop: Before going on, pretend you are the client. Choose the stories that might be most important to you. Technically, this activity is not part of the current developer-centered tutorial. It does, however, help you understand the client's next step.

SELECTING USER STORIES FOR THE FIRST ITERATION

The client now has a set of user stories to prioritize. Just as in the game of Monopoly, the priorities can change every time it's "their turn." Using yet another game as an analogy, think about the game of Scrabble. Here you draw tiles with individual letters on them and alternate trying to form words from your tiles and those that have already been placed on the board. On your turn you have to reassess your options. You have additional letters that you didn't have to choose from on your last turn, and your opponent has both added and removed possibilities from the board by placing their last word. Throughout the planning game, the client will reorder the user stories based on what the developers have accomplished so far, on what the client's current most important need is, and on what the developer estimates to be the time available and the time required by the remaining tasks.

The Client Provides a Basic Ordering of Stories

To start the process, the client will broadly prioritize the user stories. This is to keep the developers from having to estimate every story at this point. In the current tic-tac-toe example there are few enough user stories that it would be possible to estimate them all, but let's just estimate the ones that the client rates at the top. In this case, Steve organized the user stories like this. The story numbers appear in parenthesis to make it easier for you to locate the original descriptions.

Top six stories (immediate and most important):

- Board Setup (2),
- Alternate Markers (13),

- Make Move (19),
- Choose Marker (21),
- Highlighting a Three-in-a-Line (23), and
- Recognizing a Win (24).

Next four (important but can wait):

- Tie Games (4),
- Display Results and Replay (12),
- Player Names (20), and
- Illegal Moves (22).

Next three:

- Standings (6),
- Number of Games (11), and
- Quitting (18).

Next three:

- Computer First (15),
- Computer Ability (16), and
- Computer Opponent (17).

Steve was only asked to pick out his top six. He did and then asked if he could then suggest the next four that he wanted the developers to tackle. When they said yes, he continued to organize the remaining stories in relative levels of importance. Although this wasn't necessary, it did provide possibly useful information.

Stop: Take the top six stories and estimate how many units you think each one will take. Use two units for user story (2), Board Setup, and estimate the others in comparison to it.

The Developers Provide Rough Estimates of the Stories

The developers now take the top stories that the client chose and estimate how long these will take. These first estimates are going to be wrong. In a traditional software engineering approach you would base the remainder of the project on these estimates. In XP you'll refine these estimates before the end of the day and only depend on those revised estimates for, at most, a week.

The most important thing to remember while estimating is that you are approximating the relative difficulty of the tasks. In this example, the developers initially used units that correspond to a half-hour knowing that their notion of how long a unit will take will change during the course of the project. The developers are on short iteration cycles and so have committed twenty units to the first iteration. They figure that user story (2), Board Setup, will take about an hour and so the estimate is two units.

In looking at (21), Choose Marker, they figure it will take a bit longer. There's not much more to do, but they estimate it at three units. With story (13), Alternate Markers, something interesting happens. The developers initially estimated it at eleven units. Then they estimated user story (19), Make Move, at eight units.

They have more confidence in this estimate of the Make Move user story (19). They can't express why, but they both are pretty certain it will take around eight units. In this light they catch each other reconsidering the Alternate Markers story (13). Here they both agree that if Make Move (19) is estimated at eight units then Alternate Markers (13) should require more than eleven units. On the other hand, they argue that work on one of the stories may reduce the time required by the other. They then remember that they are to estimate the stories in isolation because the client gets to choose the order and may ignore their advice and choose an order different from the one preferred by developers. They go with their gut feeling and revise the estimate of the Alternate Markers user story (13) to fourteen units.

They present the following results to the client.

User story	Units
Board Setup (2)	2
Alternate Markers (13)	14
Make Move (19)	8
Choose Marker (21)	3
Highlighting a Three-in-a-Line (23)	6
Recognizing a Win (24)	6

Stop: For kicks, from this set of stories, which requires thirty-nine units of work, select twenty units' worth that you want. Again, this is a client activity so you are doing this just for fun.

The Client Chooses the Current Iteration

Now the client chooses as many units as they would like the developers to work on in the first iteration. The upper limit is the developers' allotted amount. In this case, the developers have said they can accomplish twenty units. The client isn't bound to only choose from this list. The client can ask the developers to estimate another story.

Also, take a look at the estimate for (13), Alternate Markers. It's more than half of the iteration's worth. It may be a good idea to break it down into smaller stories. For now, Steve is happy leaving it as it is. He selects the following stories for the first iteration:

Board Setup **(2)**

The board is a 3 × 3 grid. Each square on the grid is initially empty.

Make Move **(19)**

Players, alternating turns, use a mouse to select an empty square in which to place their marker. Once placed, a marker can't be moved or removed.

Choose Marker **(21)**

Tic-Tac-Toe is played by two players. At the beginning of a session, 1 player chooses either "X" or "O" as their marker, the other player is assigned the remaining one.

Recognizing a Win **(24)**

A player wins the game when three of their markers form a line either horizontally (row), vertically (column) or diagonally, and the game ends.

This is a total of nineteen units but Steve and Kathy agree that there is enough uncertainty in these estimates that this difference won't matter much.

Stop: Break each of these four stories into tasks. These tasks are development units where the user stories are client functionality units.

A CLOSER LOOK AT THE ESTIMATES

In some ways the estimates on both sides are still very rough. The client looked at a pile of stories and chose the ones they wanted first. Then the developers looked at those stories and indicated how long they think each one will take without fully considering what's involved. Then the client chose the stories for the first iteration based on client priorities together with developer estimates of the user stories and of their available time.

Now the developers are going to take a closer look at what is involved in coding each story. Once they've broken each story down into tasks, individuals will estimate and take responsibility for the tasks. The developers can then go back to the client with the revised estimates if the client needs to make further decisions. For example, if in the process of breaking the stories into tasks, the developers realize they have forty-three units of work instead of twenty, then the client will have to pick the most important twenty units worth.

The Developers Break the Stories into Tasks

The next step in the process is for the developers to meet and discuss how the four stories selected by the client can be broken down into tasks. These tasks are the developers' domain because this is how the developers will achieve the functionality specified in the user story. For example, here's the breakdown for story (2), Board Setup.

Board Setup **(2)**

The board is a 3 × 3 grid. Each square on the grid is initially empty.

Task 2-A: Board-GUI Task

Create a GUI that displays a board consisting of nine empty squares arranged in a 3 × 3 grid.

Task 2-B: Board-Model Task
Create an internal representation of a tic-tac-toe board that serves as a model of the game.

The developers don't need to share these tasks with the client. The idea is that now a pair can work on a specific task and start coding. They can decide how the task will be tested and start writing unit tests followed by the code that makes the unit tests pass. Here are the tasks for user story (19), Make Move.

Make Move **(19)**

Players, alternating turns, use a mouse to select an empty square in which to place their marker. Once placed, a marker can't be moved or removed.
Task 19-A: Determine Marker Task
Determine which is current marker based on current player.
Task 19-B: Place Marker Task
Respond to mouse click in empty square by placing current marker in square, and update internal representation.
Task 19-C: Ignore Bad Clicks Task
Ensure that mouse click in nonempty square has no effect.
Task 19-D: Determine Current Player Task
Determine current player by alternating between players starting with the specified starting player.

The first task (19-A) brought up an issue that the developers decided needed to be addressed in user story (21), Choose Marker. The developers weren't sure when the players should choose their markers. They asked Steve if he wanted to allow for the players to choose their markers before every game and he answered that no, choosing once per session would be fine. He added this information to user story (21), Choose Marker, and it was renumbered as (25) so that the changes could continue to be tracked. Here's (25), Choose Marker, along with the tasks.

Choose Marker **(25)**

Tic-Tac-Toe is played by two players. At the beginning of a session, one player chooses either "X" or "O" as their marker, the other player is assigned the remaining one.

Task 25-A: Choose Marker GUI Task
Create a GUI to allow player to choose X or O as their marker.
Task 25-B: Record Marker Choice Task
Record choice of marker for both players.

This leaves user story (24), Recognizing a Win. This user story took the developers more time to break down than the other stories. You can see from the tasks that it

appears that some design decisions were made here. There were considerable arguments about this fact. After trying different options it was agreed that it was better to halt the discussion and proceed with coding. Take a look at the way in which you broke down Recognizing a Win. Your tasks are probably different than those given below. You may have checked the whole board each time for winning configurations. You may have had row objects that represented rows, columns, and diagonals. You only needed to check completed row objects to see if they contained three of the same markers. You may have only kept track of those row objects that only contained one type of marker. As soon as a row contains one of each type of marker you no longer need to check to see if it's a winner. The point is that there are many ways to break down user story (24), Recognizing a Win.

Here's Jason and Kathy's solution.

Recognizing a Win **(24)**

A player wins the game when three of their markers form a line either horizontally (row), vertically (column) or diagonally, and the game ends.
Task 24-A: Checking Potential Wins Task
Determine if a given line contains three of the same marker.
Task 24-B: Finding Potential Wins Task
Determine which lines to check (i.e., those lines containing the last played marker).
Task 24-C: Recording a Win Task
Record a win for last player if a row returned from task 24-B is determined in task 24-A to contain three of their markers.

Task 24-B clearly stands out as suggesting a particular solution. Sometimes that is the case. eXtreme Programming doesn't forbid any design. It encourages you to design a little, test a little, and code a little. In this case the design decision may be changed later.

Stop: Look at the tasks and estimate how long each one will take using the units you used to estimate the entire stories before.

The Developers Estimate the Tasks

The developer team for this project is small. It's up to Kathy and Jason, the other developer, to estimate and claim responsibility for the tasks estimated in the previous section. Remember, Jason has been here all along; he's just been left out of the description. Each developer has ten units that they can spend in this estimation process. The developers bid on how long they think a task would take them with a partner. The low bid wins. You bid on stories until your allotment for the iteration is gone. You can not bid on more work than you can do. In a way, this simplified situation is a bit comical because when Kathy bids on a task, Jason will be her partner and when Jason bids on a task Kathy will be his partner. In other words, the same two people will be working on all of the tasks in this case. Nevertheless, it's a good idea to practice bidding and estimating tasks so they begin this phase of the planning game.

Jason begins by bidding one unit each on tasks 2-A and 2-B, the Board-GUI task and the Board-Model task. Kathy tells him that the tasks are his. She counters by bidding two units on 25-A and one unit on 25-B, the Choose Marker GUI task and the Record Marker Choice task. Jason tells her the stories are hers and he reconsiders his bid on 2-B, the Board-Model task. He says that he probably should change it to two units. Kathy tells him that he can if he wants. He decides not to. It turns out, as is often the case, that his instincts are correct. His bid is an underbid. If you worry that a bid is too low, have the confidence to change your bid. Although you are currently making a guess, you should make the most accurate guess you can.

Although you don't have to bid on tasks one story at a time, you do want to make sure that all of the tasks in a story are taken. The developers next looked at story (19), Make Move. Jason bids five units on 19-B and three on 19-C, the Place Marker task and the Ignore Bad Clicks task. Kathy thinks these estimates are very high. She counterbids four units on 19-B, Place Marker task and one on 19-C, Ignore Bad Clicks task. Jason thinks the bid on Place Marker task is okay but that the bid on 19-C, Ignore Bad Clicks task is quite low. He asked that Kathy reconsider her bid. She does, and decides she is fine. In this case, again, she may have wanted to reconsider the direction of the discussion. The person she would be pairing with was trying to indicate that he didn't think he could help finish this task in such a short time. The bidding in the planning game is more similar to two partners bidding in a game of Bridge than it is to two combatants bidding against each other for a chance at a job. The bidding is done openly and cooperatively.

Kathy also bids two units each on tasks 19-A and 19-D, Determine Marker task and Determine Current Player task. Jason counters with a bid of one unit on Determine Marker task. Kathy is overextended by one unit, but they agree to ignore this until after the bidding is done and then they'll redistribute the tasks if necessary. Kathy decides to let Jason have the Determine Marker task and Jason agrees that the Determine Current Player task is hers.

This leaves user story (24), Recognizing a Win. Kathy is still uncomfortable with the way the tasks have been written. Jason bids three units each on tasks 24-A and 24-B and one unit on 24-C, the Checking Potential Wins task, the Finding Potential Wins task, and the Recording a Win Task, respectively. Kathy considers the bids and tells Jason that the tasks are his. Here's a summary of the bidding.

	TASK #	TITLE	ESTIMATE (units)
Jason	2-A	Board-GUI	1
	2-B	Board-Model	1
	24-A	Checking Potential Wins	3
	24-B	Finding Potential Wins	3
	24-C	Recording a Win	1
	19-A	Determine Marker	1
Total			10
Kathy	19-B	Place Marker	4
	19-C	Ignore Bad Clicks	1
	19-D	Determine Current Player	2
	25-A	Choose Marker GUI	2
	25-B	Record Marker Choice	1
Total			10

You may notice that this totals to twenty units, although the client only selected nineteen units of stories. Because there was a small fudge factor built in, Jason and Kathy don't have to go back to the client and ask Steve to reorder his priorities. Story (24), Recognizing a Win, is now worth seven units and not six. If the task estimates had forced the total over twenty units, Jason and Kathy would have had to negotiate further with Steve. In this case, they are ready to begin the iteration.

REALITY CHECK

Halfway through the iteration, Kathy and Jason take a look at what they've accomplished. It took a little longer to set up their environment than they expected. Jason's estimate for 2-B, Board-Model task was too low. Kathy's estimate for 25-A, Choose Marker GUI, was a little low also. At the end of the first week they have completed all of stories (2), Board Setup, and (25), Choose Marker, as well as task 19-A, Determine Marker task. This leaves them with tasks 19-B, Place Marker task, 19-C, Ignore Bad Clicks task, 19-D, Determine Current Player task, and all of user story (24), Recognizing a Win. In terms of units, fourteen units remain to be done.

Jason smiles and says, "That's not so bad. There were problems starting up and some things took longer than they should. These next stories should go more quickly. We'll make it."

Kathy stops him and says, "No, we need to tell Steve."

Jason disagrees. "Steve doesn't need to know," he says. "By the time the week's over we'll be back on track, or pretty close. He'll never know the difference."

Kathy reminds Jason that telling the truth is fundamental to the process. "Maybe," she says, "Steve will agree that it's no big deal, but it is his call to make."

They agree to tell Steve. He's happy that two of the stories have been completed, but he's not happy to be eliminating more user stories so early in the project. He asks the developers if they need to reestimate their tasks given what they now know. They think about it and agree that their remaining estimates are still pretty accurate. The problem is that they aren't accomplishing nearly enough work. In the first week of the iteration they accomplished six units worth of work. Steve looks at the two stories left to complete. There are seven units remaining to be done in each.

Jason reminds Kathy that really he should have estimated one of the stories higher and that they could do seven units worth of stories this week. Kathy agrees with him. Although this seems plausible, it is dangerous to reassess what you should have estimated after the fact. Perhaps if they'd checked with Steve earlier they wouldn't have spent the one unit's worth of work on 19-A, Determine Marker task, and it would now be available. Steve considers his alternatives and decides that he'd like to see the team complete user story (19), Make Move.

Moving from One Iteration to Another

It's hard to know when summer ends and autumn begins. There is a transitional period where the seasons are muddied a bit. To make things easier, we have an agreed upon a date for the first day of autumn. It's the same way with iterations. An iteration ends on a particular day and yet some of the activities associated with preparing for the next iteration feel like the end of the last iteration while others feel like the beginning of the next.

As an iteration ends, you must again ask what you've accomplished. In this case Steve and Kathy completed tasks 19-B, Place Marker task, and 19-D, Determine Current Player task, but didn't quite complete 19-C, Ignore Bad Clicks task. From their perspective, they completed six units. They are allowed to commit to six units each week in the next iteration. This actually matches the reality of the first week in this iteration. Steve and Kathy completed six units each week during the iteration. Although they estimated that they would complete twenty units, they only completed twelve. This is an important and telling data point. They can bid on twelve units in the next iteration and have a fair amount of confidence that this is a more accurate estimate than twenty.

Take a minute to look at the situation from the client's point of view. Remember, the client doesn't care about tasks. The client cares about completed user stories. From that standpoint, only five units of user stories were completed in this iteration. Jason and Kathy are able to revise the estimate of user story (19), Make Move, to be one unit if they'd like because they've completed all but a task that they judged to be worth one unit.

They could also choose to reevaluate the time it would take to accomplish that task. It is likely that Steve will choose user story (19), Make Move, in the next iteration because he has so much invested in it already. For the marginal cost of one unit he gets eight units worth of value.

Before the next iteration begins, Steve needs to look at the remaining user stories and again group them according to their immediate importance to him. The developers need to estimate the stories that Steve sees as being the most important. Steve will then use these estimates to choose twelve units worth of user stories. The developers will break these stories, and a few more that don't make it into this stack, into tasks and estimate and sign-up for each task. Steve may then have to change his preferences and make choices based on these more recent estimates. The cycle continues until one side says *Stop.*

Chapter 13

Refactoring—

Sharpening Your Knife

You and your partner write a test and quickly write the code that gets the test to compile and pass. Something just doesn't feel right. Maybe the problem is with a method. It could be too long, it could be misnamed, or it could belong in a different class. Maybe you've written a very similar method in another class and these methods belong in a common super class. Maybe as you look around the class you've been working with there is complicated conditional logic or duplicated code. For some reason the code doesn't seem as clean as it could be. Take a moment and clean it up. The subtitle of Martin Fowler's book, *Refactoring,* is "improving the design of existing code." You should own a copy of Fowler's book.

Refactoring may initially seem to be outside the scope of eXtreme Programming. After all, while you're messing around with code that already works you don't appear to be spending your time on a user story that the client has selected. You are taking a moment to sharpen your chef's knife so that the next time you need to use it you will be able to cut through food more quickly and more safely. You will return to this code at some point and not remember what you were thinking when you wrote it. If that's too close to home, you may visit code that someone else has written and not know what they were thinking when they wrote it. You'll wish they had taken the time to clean up the code to better communicate its meaning.

A comprehensive suite of unit tests is necessary for fearless refactoring. Unit tests let you know the code is working at all steps of your refactoring. You don't want to make the code prettier and break functionality. You only refactor working code. Your code should be passing all unit tests before you refactor it. As with other XP practices, you will refactor in very small steps, pausing to compile the code and run your unit tests after each small step. You should never need to resort to the debugger. All changes are small and reversible and the unit tests will point to where problems arise.

What if you have just written a unit test and you're about to write the code that makes it pass when you notice something that badly needs refactoring? You've noticed