4

Requirement Gathering

If you don't know where you are going, you'll end up someplace else.

—Yogi Berra

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Why requirements are important
- The characteristics of good requirements
- The MOSCOW method for prioritizing requirements
- Audience-oriented, FURPS, and FURPS+ methods for categorizing requirements
- Methods for gathering customer goals and turning them into requirements
- Brainstorming techniques
- Methods for recording requirements such as formal specifications, user stories, and prototypes

It's tempting to say that requirement gathering is the most important part of a software project. After all, if you get the requirements wrong, the resulting application won't solve the users' problems. You'll be like a tourist in Boston with a broken GPS. You may get somewhere interesting, but you probably won't get where you want to go.

Even though requirements are important for setting a project's direction, a project can fail at any other stage, too. If you build a flawed design, write bad code, fail to test properly, or even provide incorrect training materials, the project can still fail. If any one of the links in the development chain fails, the project will fail.

Let's just say that requirement gathering is the first link in the chain, so it's the first place where you can screw things up badly. Requirements *do* set the stage for everything that follows, so while you can argue over whether this is the most important step, it's definitely *an* important step.

This chapter explains what requirement gathering is and lists some typical requirements that are useful in many projects. It also describes some techniques you can use to gather requirements effectively.

REQUIREMENTS DEFINED

Requirements are the features that your application must provide. At the beginning of the project, you gather requirements from the customers to figure out what you need to build. Throughout development, you use the requirements to guide development and ensure that you're heading in the right direction. At the end of the project, you use the requirements to verify that the finished application actually does what it's supposed to do.

Depending on the project's scope and complexity, you might need only a few requirements, or you might need hundreds of pages of requirements. The number and type of requirements can also depend on the level of formality the customers want.

For example, if you're working on a casual in-house project, your boss may be satisfied with a few blanket requirements such as "find ways to improve order processing" or "write a tool to send spam to customers." As long as you create something vaguely useful, your project will probably be viewed as a success. (If not, you'll find out at your annual review.) As you'll see shortly, these sorts of vague requirements have some problems.

Large projects with higher stakes typically have far more requirements that are spelled out much more formally and in great detail. For example, if you're building an autopilot system for 747s or you're writing software to control pacemakers, your requirements must be unambiguous. You can't wait until the final weeks of testing to start thinking about whether "easy installation" means patients should change their own pacemaker parameters from a cell phone.

The following sections describe some of the properties that requirements should have to be useful.

Clear

Good requirements are clear, concise, and easy to understand. That means they can't be pumped full of management-speak, florid prose, and confusing jargon.

It is okay to use technical terms and abbreviations if they are defined somewhere or they are common knowledge in the project's domain. For example, when I worked at a phone company research lab, we often used terms like POTS (plain old telephone service), PBX (public branch exchange), NPA (numbering plan area, known to nontelephone people as an area code), and ISDN (integrated services digital network, or as some of us used to call it, "I still don't know"). The customers and development team members all knew those terms, so they were safe to use in the requirements.

To be clear, requirements cannot be vague or ill-defined. Each requirement must state in concrete, no-nonsense terms exactly what it requires.

For example, suppose you're working on a program to schedule appointments for utility repair people. (Those appointments that typically say, "We'll be there sometime between 6:00 a.m. and midnight during the next 2 weeks.") A requirement such as, "Improve appointment scheduling," is too vague to be useful. Does this mean you should tighten the appointment windows even if it means missing more appointments? Does it mean repair people should leave and make a new appointment if they can't finish a job within 1 hour? Or does it mean something crazy like letting customers tell you what times they can actually be home and then fitting appointments to those times?

A better requirement would be, "Reduce appointment start windows to no more than 2 hours while meeting 90 percent of the scheduled appointments."

Unambiguous

In addition to being clear and concrete, a requirement must be unambiguous. If the requirement is worded so that you can't tell what it requires, then you can't build a system to satisfy it. Although this may seem like an obvious feature of any good requirement, it's sometimes harder to guarantee than you might think.

For example, suppose you're building a street map application for inline skaters, and you have a requirement that says the program will, "Find the best route from a start location to a destination location." This can't be all that hard. After all, Google Maps, Yahoo Maps, MapQuest, Bing Maps, and other sites all do something like this.

But how do you define the "best" route? The shortest route? The route that uses only physically separated bike paths so that the user doesn't have to skate in the street? Or maybe the route that passes the most Starbucks locations?

Even if you decide the "best" route means the shortest one, what does that mean? The route that's the shortest in distance? Or the shortest in time? What if the route of least distance goes up a steep hill or down a set of stairs and that increases its time? (In this example, you might change the requirements to let the users decide how to pick the "best" route at run time.)

As you write requirements, do your best to make them unambiguous. Read them carefully to make sure you can't think of any way to interpret them other than the way you intend.

Then run them past some other people (particularly customers and end user representatives) to see if they agree with you.

A TIMELY JOKE

CUSTOMER: I need you to write a program to find customers that haven't paid their bills within 5 seconds.

DEVELOPER: Harsh! Most companies give their customers 30 days to pay their bills.

Consistent

A project's requirements must be consistent with each other. That means not only that they cannot contradict each other, but that they also don't provide so many constraints that the problem is unsolvable. Each requirement must also be self-consistent. (In other words, it must be possible to achieve.)

Consider again the earlier example of utility repair appointments. You might like to include the following two requirements:

- Reduce appointment start windows to no more than 2 hours.
- Meet 90 percent of the scheduled appointments.

It may be that you cannot satisfy these two requirements at the same time. (At least using only software. You might do it if you hire more repair people.)

In a complex project, it's not always obvious if a set of requirements is mutually consistent. Sometimes, any pair of requirements is satisfiable but larger combinations of requirements are not.

A common software engineering expression is, "Fast, good, cheap. Pick two." The idea is you can trade development speed, development quality, and cost, but you can't win in all three dimensions. Only three possible combinations work:

- Build something quickly with high quality and high cost.
- Build something quickly and inexpensively but with low quality.
- Build with high quality and low cost but over a long time.

Try to keep new requirements consistent with existing requirements. Or rewrite older requirements as necessary. When you finish gathering all the requirements, go through them again and look for inconsistencies.

Prioritized

When you start working on the project's schedule, it's likely you'll need to cut a few nice-to-haves from the design. You might like to include every feature but don't have the time or budget, so something's got to go.

At this point, you need to prioritize the requirements. If you've assigned costs (usually in terms of time to implement) and priorities to the requirements, then you can defer the high-cost, low-priority requirements until a later release.

Customers sometimes have trouble deciding which requirements they can live without. They'll argue, complain, and generally act like you're asking which of their children they want to feed to the dingoes. Unfortunately, unless they can come up with a big enough budget and timescale, they're going to need to make some sort of decision.

The exception occurs when you work on life-critical applications such as nuclear reactor cooling, air traffic control, and space shuttle flight software. In those types of applications, the customer may have a lot of "must have" requirements that you can't remove without compromising the applications' safety. You may remove cosmetic requirements like a space shuttle's automatic

turn-signal cancellation feature, but you're probably going to need to keep the fuel monitor and flight path calculator.

THE MOSCOW METHOD

MOSCOW is an acronym to help you remember a common system for prioritizing application features. The consonants in MOSCOW stand for the following:

M—Must. These are *required* features that must be included. They are necessary for the project to be considered a success.

S—Should. These are *important* features that should be included if possible. If there's a work-around and there's no room in the release 1 schedule, these may be deferred until release 2.

C—Could. These are *desirable* features that can be omitted if they won't fit in the schedule. They can be pushed back into release 2, but they're not as important as the "should" features, so they may not make it into release 2, either.

W—Won't. These are *completely optional* features that the customers have agreed will not be included in the current release. They may be included in a future release if time permits. (Or they may just be included in the requirements list to make a particularly loud and politically connected customer happy, and you have no intention of ever including these features.)

Let's face it. If a feature isn't a "must" or "should," then its chances of ever being implemented are slim. After this release has been used for a while, you'll probably receive tons of bug reports, requests for changes, and pleas for new features, so in the next release you still won't have time for the "could" and "won't" features.

(Unless you're one of these big software companies, who shall remain nameless, that thinks it needs to push a new version of its products out every 2 years to make customers buy something. Sometimes those products reach deep into the "could" and "won't" categories, and perhaps even the "why?" and "you must be joking!" categories.)

EXAMPLE ClassyDraw

For an example of using the MOSCOW method, consider a fictional drawing program named ClassyDraw. It's somewhat similar to MS Paint and allows you to draw line segments, ellipses, polygons, text, and other shapes. The big difference is that ClassyDraw represents each shape you draw as an object that you can later select, move, resize, modify, and delete.

Here's an initial requirement list:

1. Draw: line segments, sequences of line segments, splines, polygons, ellipses, circles, rectangles, rounded rectangles, stars, images, and other shapes.
2. Save and load files.

3. Protect the current drawing. For example, if the user tries to close the program while there are unsaved changes, prompt the user.
4. Let the user specify the line style and colors used to draw shapes.
5. Let the user specify the fill style and colors used to draw shapes.
6. Click to select an object.
7. Click and drag to select multiple objects.
8. Click or click and drag with the Shift key down to add objects to the current selection.
9. Click or click and drag with the Ctrl key down to toggle objects in and out of the current selection.
10. Click and drag the selected objects to move them.
11. Edit the selected objects' line and fill styles.
12. Delete the selected objects.
13. Select colors from a palette.
14. Place custom colors in a custom palette.
15. Support transparency.
16. Copy and paste the entire drawing, a rectangular selection, or an irregular selection as a bitmapped image.
17. Copy, cut, and paste the currently selected objects.
18. Allow the user to write scripts to add shapes to a drawing.
19. Let the user rearrange the palettes and toolbars.
20. Auto-save the current drawing periodically. If the program crashes, allow the user to reload the most recently saved version.
21. Auto-save the current drawing every time a change is made. If the program crashes, allow the user to reload the most recently saved version.
22. Provide online help.
23. Provide online tutorials.

Now you can use the MOSCOW method to prioritize these requirements.

Must. To identify the "must" requirements, examine each requirement and ask yourself: Could that requirement be omitted? Would the program be usable without that feature? Will users give the product 1-star reviews and say they wish they could give 0 stars? That the product would be overpriced if it were freeware?

The ClassyDraw application *must* be able to save and load files (2). You could build early test versions that couldn't, but it would be unacceptable to users.

Similarly the program must ensure the safety of the current drawing (3). The users would never forgive the program if it discarded a complicated drawing without any warning.

The program wouldn't be useful if it didn't draw, so the program must draw at least a few shapes (1). For starters, it could draw line segments, rectangles, and ellipses. You could add more shapes in later releases.

The program should probably allow the user to click objects to select them. Otherwise, the user may as well use MS Paint, so requirement 6 is a must. Of course, there's little point in selecting an object if you can't do anything with it, so the program must let the user at least move (10) and delete (12) the selected objects.

The "must" requirements include 1 (partial), 2, 3, 6, 10, and 12.

Should. To identify the "should" requirements, examine each of the remaining requirements and ask yourself, "Does that feature significantly enhance the product? If it were omitted, would users be constantly asking why it wasn't included? Is the feature common in other, similar applications? Will users give the product 2-star and 3-star reviews?"

Several requirements that are fairly standard for drawing applications didn't make the cut for the "must" category. (You could say they didn't pass muster.)

Most (if not all) of the other shapes in requirement 1 should be included in this group. When drawing new shapes, the user should also indicate the line and fill styles the new shapes should have (4, 5). That will require specifying colors, at least from a palette (13).

The click-and-drag selection technique (7) should be included, as should the ability to hold down the Shift or Ctrl key while making selections (8, 9).

Any decent application should have help (22) and documentation (23), so those should also be included.

The "should" requirements include 1 (remaining), 4, 5, 7, 8, 9, 13, 22, and 23.

Could. To identify the "could" requirements, examine each of the remaining requirements and ask yourself, "Would that requirement be useful to the users? Is it something special that other similar applications don't have? Will this help bump reviews up to 4 or 5 stars? Is this a feature that we should include at some point, just not in the first release?"

Another way to approach this category is to ask: Which features will we need in the long term? Which of the remaining features shouldn't be dumped in the trash heap labeled "won't"?

Most of the remaining requirements should probably not go in the "won't" pile. If they were that bad , they probably wouldn't have made it into the requirements list in the first place.

The "could" category should definitely include the ability to edit selected objects (11). This is another of the main reasons for allowing the user to select objects.

Support for custom colors (14) and transparency (15) would also be nice, if time permits. Cut, copy, and paste for images (16) and selected objects (17) would be useful, so they should be included.

The "could" requirements include 11, 14, 15, 16, and 17.

Won't. To identify the "won't" requirements, examine the remaining requirements and ask yourself, "Is this unnecessary, confusing, or just plain stupid? Will it be used only rarely? Does it add nothing useful to the application?" If you can't answer "yes" to those questions for a particular requirement, then you should think about moving that requirement into one of the other categories.

For this application, allowing users to write scripts (18) would be cool but probably rarely used. Letting the user rearrange palettes and toolbars (19) would be a nice touch, but isn't important.

Auto-saving (20, 21) is also a nice touch, but probably unnecessary. We can look at user requests and conduct surveys to see if this feature would be worth adding to a future release.

The "won't" requirements include 18, 19, 20, and 21.

After you've assigned each requirement to a category, go back through them and make sure you're happy with their assignments. If a requirement in the "could" category seems more important than one in the "should" category, switch them.

Also make sure every requirement is in some category and that every category contains some requirement. If every requirement is in the "must" category, then you may need to rethink your priorities (or your customer's priorities), or be sure you'll have enough time to get everything done.

Verifiable

Requirements must be verifiable. If you can't verify a requirement, how do you know whether you've met it?

Being verifiable means the requirements must be limited and precisely defined. They can't be open-ended statements such as, "Process more work orders per hour than are currently being processed." How many work orders is "more?" Technically, processing one more work order per hour is "more," but that probably won't satisfy your customer. What about 100? Or 1,000?

A better requirement would say, "Process at least 100 work orders per hour." It should be relatively easy to determine whether your program meets this requirement.

Even with this improved requirement, verification might be tricky because it relies on some assumptions that it doesn't define. For example, the requirement probably assumes you're processing work orders in the middle of a typical workday, not during a big clearance event, during peak ordering hours, or during a power outage.

An even better requirement might be, "Process at least 100 work orders per hour on average during a typical work day." You may want to refine the requirement a bit to try to say what a "typical work day" is, but this version should be good enough for most reasonable customers.

Words to Avoid

Some words are ambiguous or subjective, and adding them to a requirement can make the whole thing fuzzy and imprecise. The following list gives examples of words that may make requirements less exact.

- **Comparatives**—Words like faster, better, more, and shinier. How much faster? Define "better." How much more? These need to be quantified.
- **Imprecise adjectives**—Words like fast, robust, user-friendly, efficient, flexible, and glorious. These are just other forms of the comparatives. They look great in management reports, business cases, and marketing material, but they're too imprecise to use in requirements.

- **Vague commands**—Words like minimize, maximize, improve, and optimize. Unless you use these in a technical algorithmic sense (for example, if you optimize flow through a network), these are just fancy ways to say, "Do your best." Even in an algorithmic sense, these sorts of words are often applied to hard problems where exact solutions may not exist. In any case, you need to make the goals more concrete. Provide some numbers or other criteria you can use to determine whether a requirement has been met.

REQUIREMENT CATEGORIES

In general, requirements tell what an application is supposed to do. Good requirements share certain characteristics (they're clear, unambiguous, consistent, prioritized, and verifiable), but there are several kinds of requirements that are aimed at different audiences or that focus on different aspects of the application. For example, business requirements focus on a project's high-level objectives and functional requirements give the developers more detailed lists of goals to accomplish.

Assigning categories to your requirements isn't the point here. (Although there are two kinds of people in the world: those who like to group things into categories and those who don't. If you're one of the former, then you may need to do this for your own peace of mind.) The real point here is that you can use the categories as a checklist to make sure you've created requirements for the most important parts of the project. For example, if you look through the requirements and the reliability category is empty, you might consider adding some new requirements.

You can categorize requirements in several ways. The following sections describe four ways to categorize requirements.

Audience-Oriented Requirements

These categories focus on different audiences and the different points of view that each audience has. They use a somewhat business-oriented perspective to classify requirements according to the people who care the most about them.

For example, the corporate vice president of Plausible Deniability probably doesn't care too much about which button a call center clerk needs to press to launch a customer into a never-ending call tree as long as it works. In contrast, the clerk needs to know which button to press.

The following sections describe some of the more common business-oriented categories.

Business Requirements

Business requirements lay out the project's high-level goals. They explain what the customer hopes to achieve with the project.

Notice the word "hopes." Customers sometimes try to include all their hopes and dreams in the business requirements in addition to verifiable objectives. For example, they might say the project will "Increase profits by 25 percent" or "Increase demand and gain 10,000 new customers." Although those goals have numbers in them, they're probably outside the scope of what you can achieve through software engineering alone. They're more like marketing targets than project requirements. You can craft the best application ever put together, but someone still needs to use it properly to realize the new profits and customers.

Sometimes, those vague goals are unavoidable in business requirements, but if possible you should try to push them into the business case. The business case is a more marketing-style document that attempts to justify the project. Those often include graphs and charts showing projected costs, demand, sales figures, and other values that aren't known exactly in advance.

To think of this another way, I have no qualms about promising to write a system that can pull up a customer's records in less than 3 seconds or find the closest donut shop that's open at 2 a.m. (if you give me the data). But I wouldn't want to promise to improve morale in the Customer Complaints department by 15 percent. (What would that even mean?)

User Requirements

User requirements (which are also called *stakeholder requirements* by managers who like to use the word "stakeholder"), describe how the project will be used by the eventual end users. They often include things like sketches of forms, scripts that show the steps users will perform to accomplish specific tasks, use cases, and prototypes. (The sections "Use Cases" and "Prototypes" later in this chapter say more about the last two.)

Sometimes these requirements are very detailed, spelling out exactly what an application must do under different circumstances. Other times they specify *what* the user needs to accomplish but not necessarily *how* the application must accomplish it.

EXAMPLE Overly Specific Selections

In this example, you see how you can turn an overly specific requirement into one that's flexible without making it vague.

Suppose you're building a phone application that lets customers place orders at a sandwich and bagel shop called The Loxsmith. The program should let customers select the toppings they want on their bagels. They include lox (naturally), butter, cream cheese, gummy bears, and so on. Here's one way you could word this requirement:

> *The toppings form will display a list of toppings. The user can check boxes next to the toppings to add them to the bagel.*

That's a fine requirement. Clear, concise, verifiable. Everything you could want in a requirement. Unfortunately, it's also unnecessarily specific. It forces the designers and developers to use a specific technique to achieve the higher-level goal of letting the customer select toppings.

During testing, you might discover that The Loxsmith provides more than 200 toppings. In that case, the program won't be able to display a list of every topping at the same time. The user will need to scroll through the list, and that will make it hard for the customer to see what toppings are selected.

Here's a different version of the same requirement that doesn't restrict the developers as much.

> *The toppings form will allow the user to select the toppings put on the bagel.*

The difference is small but important. With this version, the developers can explore different methods for selecting toppings. If you have user-interface specialists on your team, they may create a variety of

possible solutions. For example, customers might drag and drop selections from a big scrollable list on the left onto a shorter list of selected items on the right. Then they could always see what toppings were selected. You might even display a cartoon picture of a bagel holding the user's four dozen selected toppings piled up like the Leaning Tower of Pisa.

Vague requirements are bad, but flexible requirements let you explore different options before you start writing code. To keep requirements as flexible as possible, try to make the requirements spell out the project's *needs* without mandating a particular approach.

Functional Requirements

Functional requirements are detailed statements of the project's desired capabilities. They're similar to the user requirements but they may also include things that the users won't see directly. For example, they might describe reports that the application produces, interfaces to other applications, and workflows that route orders from one user to another during processing.

These are things the application should do.

Note that some requirements could fall into multiple categories. For example, you could consider most user requirements to be functional requirements. They not only describe a task that will be performed by the user, but they also describe something that the application will do.

Nonfunctional Requirements

Nonfunctional requirements are statements about the quality of the application's behavior or constraints on how it produces a desired result. They specify things such as the application's performance, reliability, and security characteristics.

For example, a functional requirement would be, "Allow users to reserve a hovercraft online." A nonfunctional requirement would be, "The application must support 20 users simultaneously making reservations at any hour of the day."

Implementation Requirements

Implementation requirements are temporary features that are needed to transition to using the new system but that will be later discarded. For example, suppose you're designing an invoice-tracking system to replace an existing system. After you finish testing the system and are ready to use it full time, you need a method to copy any pending invoices from the old database into the new one. That method is an implementation requirement.

The tasks described in implementation requirements don't always involve programming. For example, you could hire a bunch of teenagers on summer break to retype the old invoices into the new system. (Although you'll probably get a quicker and more consistent result if you write a program to convert the data into the new format. The program won't get bored and stop coming to work when the next release of *Grand Theft Auto* comes out.)

Other implementation requirements include hiring new staff, buying new hardware, preparing training materials, and actually training the users to use the new system.

FURPS

FURPS is an acronym for this system's requirement categories: functionality, usability, reliability, performance, and scalability. It was developed by Hewlett-Packard (and later extended by adding a + at the end to get FURPS+).

The following list summarizes the FURPS categories:

- **Functionality**—What the application should do. These requirements describe the system's general features including what it does, interfaces with other systems, security, and so forth.
- **Usability**—What the program should look like. These requirements describe user-oriented features such as the application's general appearance, ease of use, navigation methods, and responsiveness.
- **Reliability**—How reliable the system should be. These requirements indicate such things as when the system should be available (12 hours per day from 7:00 a.m to 8:00 p.m.), how often it can fail (3 times per year for no more than 1 hour each time), and how accurate the system is (80 percent of the service calls must start within their predicted delivery windows).
- **Performance**—How efficient the system should be. These requirements describe such things as the application's speed, memory usage, disk usage, and database capacity.
- **Supportability**—How easy it is to support the application. These requirements include such things as how easy it will be to maintain the application, how easy it is to test the code, and how flexible the application is. (For example, the application might let users set parameters to determine how it behaves.)

FURPS+

FURPS was extended into FURPS+ to add a few requirements categories that software engineers thought were missing. The following list summarizes the new categories:

- **Design constraints**—These are constraints on the design that are driven by other factors such as the hardware platform, software platform, network characteristics, or database. For example, suppose you're building a financial application and you want an extremely reliable backup system. In that case, you might require the project to use a shadowed or mirrored database that stores every transaction off-site in case the main database crashes.
- **Implementation requirements**—These are constraints on the way the software is built. For example, you might require developers to meet the Capability Maturity Model Integration (CMMI) or ISO 9000 standards. (For more information on those, see `www.cmmifaq.info` and `www.iso.org/iso/iso_9000` respectively.)
- **Interface requirements**—These are constraints on the system's interfaces with other systems. They tell what other systems will exchange data with the one you're building. They describe things like the kinds of interactions that will take place, when they will occur, and the format of the data that will be exchanged.
- **Physical requirements**—These are constraints on the hardware and physical devices that the system will use. For example, they might require a minimum amount of processing power, a maximum amount of electrical power, easy portability (such as a tablet or smartphone), touch screens, or environmental features (must work in boiling acid).

EXAMPLE FURPS+ Checklist

In this example, we'll use FURPS+ to see if any requirements are missing for the The Loxsmith ordering application. Consider the following abbreviated list of requirements. The program should allow the user to:

Start an order that might include multiple items.

Select bagel type.

Select toppings.

Select sandwich bread.

Select sandwich toppings.

Select drinks.

Select pickup time.

Pay or decide to pay at pickup.

I've left out a lot of details from this list such as the specific bagel, bread, and topping types that are available, but at first glance, this seems like a reasonable set of requirements. It describes what the application should do but doesn't impose unnecessary constraints on how the developers should build it. It's a bit more vague than I would like (how do you *verify* that the user can select toppings?), but you can flesh that out. (In fact, I'll talk a bit about ways you can do that later in this chapter, particularly when I talk about use cases in the section "Use Cases.")

For this example, assume the requirements are spelled out in specific (but flexible) detail. Then use FURPS+ to see if there's anything important missing from this list. Spend a few minutes to decide in which FURPS+ category each of the requirements belongs.

Although the initial requirements all seem reasonable, they're all functionality requirements. They tell what the application should do but don't give much information about usability, reliability, performance, and other requirements that should belong to the other FURPS+ categories.

You might think that a requirements list containing only functionality requirements would be an unusual situation. However, left to their own devices, many programmers come up with exactly this sort of list. They focus on the work they are going to do and how it will look to the users. That's a good place to start the design, but in the background they're making a huge number of assumptions about things they take for granted.

For example, suppose you're a developer who writes Java applications running on Android tablets. In that case, you may think the previous list of requirements is just fine. Your version of the Eclipse Java development environment is up to date, you've installed the Android Software Development Kit (SDK), and you have "Eye of the Tiger" blasting on your headphones. You're ready to start cranking out code.

Unfortunately you're also making a ton of assumptions that may or may not sit well with the customer. In this example, you're assuming the application will be written in Java to run on Android tablets. What if the customer wants the application to run on an iPhone, Windows Phone, mobile-oriented web page, Google Glass, or some sort of smart wearable ankle bangle device? Or maybe all of the above?

Sometimes, you may not want any requirements in a particular category, but the fact that the preceding list contains *only* functionality requirements is a strong hint that we're doing something wrong. You

should at least think about every category and either (1) come up with some new requirements that belong there, or (2) write down why you don't think you need any requirements for that category.

So now look at the FURPS+ requirement categories:

Functionality—(What the program should do.) The initial list of requirements covers this category.

Usability—(What the program should look like.) You could add some requirements indicating how the user navigates from starting an order to picking sandwich and bagel ingredients. You could also provide details about login (should we create customer accounts?) and the checkout method. You should also specify that each form will display The Loxsmith logo.

Reliability—(How reliable the system should be.) Should the application be available only while The Loxsmith is open? Or should customers be able to pre-order a morning jalapeno popper bagel and double kopi luwak to pick up on the way in to work?

Performance—(How efficient the system should be.) How quickly should the application respond to customers (assuming they have a fast Internet connection)?

Supportability—(How easy should the system be to support?) The requirements should indicate that The Loxsmith employees can edit the information about the types of breads, bagels, toppings, and other items that are available. You might also want to add automated testing requirements, information about help available to customers, and any plans for future versions of the project.

Design—(Design constraints.) Here's where you would specify the target hardware and software platforms. For example, you might want the program to run on iPhones (code written with Xcode) and Windows Phones (code written in C#).

Implementation—(Constraints on the way the software is built.) You can specify software standards. For example, you might require pair programming or agile methods. (Those are described in Chapter 14, "RAD.")

Interface—(Interfaces with other systems.) Perhaps you want the application to call web services that use Simple Object Access Protocol (SOAP) to let other programs place sandwich orders. (Although it's not clear how many other companies will want an automated ordering interface to The Loxsmith, so perhaps this category will be intentionally left blank.)

Physical—(Hardware requirements.) For this application, the customers provide their own hardware (such as phones and tablets) so you don't need to specify those. You might want to specify the server hardware. Or you might want to lease space on an Internet service provider so that you don't need to buy your own hardware. (You should probably still study the available options so that you know how powerful they are and how much they cost.)

Using requirements categories as a checklist can help you notice if you are missing certain kinds of requirements. In this example, it helped identify a lot of requirements that might have been missed or hidden inside developer assumptions.

Common Requirements

The following list summarizes some specific requirements that arise in many applications.

- **Screens**—What screens are needed?
- **Menus**—What menus will the screens have?

- **Navigation**—How will the users navigate through different parts of the system? Will they click buttons, use menus, or click forward and backward arrows? Or some combination of those methods?
- **Work flow**—How does data (work orders, purchase requests, invoices, and other data) move through the system?
- **Login**—How is login information stored and validated? What are the password formats (such as, must require at least one letter and number) and rules (as in, passwords must be changed monthly)?
- **User types**—Are there different kinds of users such as order entry clerk, shipping clerk, supervisor, and admin? Do they need different privileges?
- **Audit tracking and history**—Does the system need to keep track of who made changes to the data? (For example, so you can see who changed a customer to premier status.)
- **Archiving**—Does the system need to archive older data to free up space in the database? Does it need to copy data into a data warehouse for analysis?
- **Configuration**—Should the application provide configuration screens that let the system administrators change the way the program works? For example, those screens might let system administrators edit product data, set shipping and handling prices, and set algorithm parameters. (If you don't build these sorts of screens, you'll have to make those changes for the customers later.)

GATHERING REQUIREMENTS

At this point you know what makes a good requirement (clear, unambiguous, consistent, prioritized, and verifiable). You also know how to categorize requirements using audience-oriented, FURPS, or FURPS+ methods. But how do you actually pry the requirements out of the customers?

The following sections describe several techniques you can use to gather and refine requirements.

Listen to Customers (and Users)

Sometimes, customers come equipped with fully developed requirements spelling out exactly what the application should do, how it should work, and what it should look like. More often they just have a problem that they want solved and a vague notion that a computer might somehow help.

Start by listening to the customers. Learn as much as you can about the problem they are trying to address and any ideas they may have about how the application might solve that problem. Initially, focus as much as possible on the problem, not on the customers' suggested solutions, so you can keep the requirements flexible.

If the customers insist on a particular feature that you think is unimportant, or if they request something that just seems strange, ask them why they want it. Sometimes, the requirement may be a random thought that isn't actually important, but sometimes the customers have a good reason that you just don't understand. Often the reason is so obvious to them that it doesn't occur to them to explain it until you ask. The customers probably know a lot more about their business than you do, and they may make assumptions about facts that are common knowledge to them but mysterious to you.

AN OFFER YOU CAN'T REFUSE

Suppose The Don's Waste Removal Service asks you to write an application that lets users plot out routes for garbage trucks. You're working through the list of requirements with the owner, Don, and he says, "A route that contains lots of left turns should be given no respect."

To most people, that may seem like a strange requirement. What has Don got against left turns?

Don's been working with garbage trucks for a long time so, like many people who do a lot of vehicle routing, he knows that trucks turning left spend more time waiting for cross traffic, so they burn more fuel. They are also more likely to be involved in accidents. (It always amazes me that people can fail to notice a 20-ton garbage truck stopped in front of them, but it happened in my neighborhood not long ago.) Penalizing routes that contain left turns (and U-turns) will save the company money.

Take lots of notes while you're listening to the customers. They sometimes mention these important but puzzling tidbits in passing. If a customer requirement seems odd, dig a bit deeper to find out what, if anything, is behind the request.

Use the Five Ws (and One H)

Sometimes customers have trouble articulating their needs. You can help by using the five Ws (who, what, when, where, and why) and one H (how).

Who

Ask who will be using the software and get to know as much as you can about those people. Find out if the users and the customers are the same and learn as much about the users as you can.

For example, if you're writing medical billing software, the users might be data entry operators who type in patient data all day. In contrast, your customers may be corporate executives. They may have worked their way up through the ranks (in which case they probably know everything about medical data entry down to the last billing code) or they may have followed a more business-school-oriented career path (in which case they may not know a W59.22 from a V95.43). (It's worth the time to look these up in your favorite browser.)

What

Figure out what the customers need the application to do. Focus on the goals as much as possible rather than the customers' ideas about how the solution should work. Sometimes, the customers have good ideas about what the application should look like, but you should try to keep your options open. Often the project members have a better idea than the customers of the kinds of things an application can do, so they may come up with better solutions if they focus on the goals.

(Of course, the customer is always right, at least until your paycheck is signed, so if the customer absolutely insists that the application must include a graphical slide rule instead of a calculator, chalk it up as an interesting exercise in graphics programming and make it happen.)

When

Find out when the application is needed. If the application will be rolled out in phases, find out which features are needed when.

When you have a good idea about what the project requires, use Gantt charts and the other techniques described in Chapter 3, "Project Management," to figure out how much time is actually needed. Then compare the customers' desired timeline to the required work schedule. If the two don't match, you need to talk to the customers about deferring some features to a later release.

Don't let the customers assume they can get everything on their time schedule just by "motivating you harder." In *Star Trek*, Scotty can squeeze eight weeks' worth of work into just two, but that rarely works in real-world software engineering. You're far more likely to watch helplessly as your best programmers jump ship before your project hits the rocky shoals of impossible deadlines.

Where

Find out where the application will be used. Will it be used on desktop computers in an air-conditioned office? Or will it be used on phones in a noisy subway?

Why

Ask why the customers need the application. Note that you don't need to be unnecessarily stupid. If the customers say, "We want to automate our parts ordering system so that we can build custom scooters more quickly," you don't need to respond with, "Why?" The customers just told you why.

Instead, use the "why" question to help clarify the customers' needs and see if it is real. Sometimes, customers don't have a well-thought-out reason for building a new system. They just think it will help but don't actually know why. (Or customers may have just received a new copy of *Management Buzzwords Monthly* and they're convinced they can crowdsource custom scooter design.)

Find out if there is a real reason to believe a new application will help. Is the problem really that ordering parts is inefficient? Or is the problem that each order requires a different set of parts that have a long shipping time? If streamlining the ordering process will cut the ordering time from 2 days to 1.5 days, while still leaving 4–6 weeks of shipping delay, then a new software application may not be the best place to spend your resources. (It might be better to maintain an inventory of slow-to-order parts such as wheel spinners and spoilers.)

How

The "What" section earlier in this chapter said you should focus on the goals rather than the customers' ideas about the solution. That's true, but you shouldn't completely ignore the customers' ideas. Sometimes, customers have good ideas, particularly if they relate to existing practices. If the users are used to doing something a certain way, you may reduce training time by making the application mimic that approach. Be sure to look outside the box for other solutions, but don't automatically think that software developers always make better decisions than the customers.

Study Users

Interviewing customers (and users) can get you a lot of information, but often customers (and users) won't tell you everything they do or need to do. They often take for granted details that they consider trivial but that may be important to the development team.

For example, suppose the users grind through long, tedious reports every day. The reports are so long, they often end the day in the middle of a report and need to continue working on it the next day. This may seem so obvious to the users that you don't discuss the issue.

A typical reporting application might require the users to log in every day, search for a particular report, and double-click it to open it. That could take a while (particularly if the user forgets which report it is). Fortunately, you know that users often start the day by reopening the last report of the previous day, so you can streamline the process. Instead of making users remember what report they last had open, the program can remember. You can then provide a button or menu item to immediately jump to that report.

By studying users as they work, you can learn more about what they need to do and how they currently do it. Then with your software-engineering perspective, you can look for solutions that might not occur to the users.

PRINTING PUZZLE

Watching users in their natural habitat often pays off. Many years ago, I was visiting a telephone company billing center in preparation for a project that automatically identified customers who hadn't paid their bills and so it could disconnect their service. We spent a week there studying the existing software systems and the users. It was interesting, but the reason I'm mentioning it now is a small comment made by one of the managers. In passing, she said something like, "I sure wish you could do something about the Overdue Accounts Report. Ha, ha."

That's the sort of comment that should make you dig deeper. What was this report and why was it a problem? It turned out that the existing software system printed out a list of every customer with an outstanding balance for every billing cycle. This was a *big* billing center serving approximately 15 million customers, so every two days (there were 15 billing cycles per month) the printer spit out a 3-foot tall pile of paper listing every customer in the cycle with an outstanding balance.

Balances ranged from a few cents to tens of thousands of dollars, and the big-balance customers were costing the company tons of money. Unfortunately, the printout listed the customers in some weird arrangement (sorted by customer ID or zodiac sign or something), so the billing people couldn't find the customers with the big balances.

What the customers didn't know (but we did) is that it's relatively easy to build a printer emulation program. It took approximately one week (mostly spent getting management approval) to write a program that pretended to be a printer, sucked up all the overdue account information, and sorted it by balance. It turned out that of

the thousands of pages of data produced every two days, the customers only needed the first two.

The moral of the story is, you need to pay attention to the customers' comments. They don't know what you can do with the computer, and you don't know their needs.

As you study the users, pay attention to how they do things. Look at the forms they fill out (paper or online). Figure out where they spend most of their time. Look for the tasks that go smoothly and those that don't. You can use that information to identify areas in which your project can help.

REFINING REQUIREMENTS

After you've talked to the customers and users, and watched the users at work, you should have a good understanding about the users' current operations and needs. (If you don't, ask more questions and watch the users some more until you do.)

Next, you need to use what you've learned to develop ideas for solving the user's problems. You need to distill the goals (what the customers need to do) into approaches (how the application will do it).

At a high level, the requirement, "Process customer records" is fine. It's also nice and flexible, so it allows you to explore many options for achieving that goal.

At some point, however, you need to turn the goals into something that you can actually build. You need to figure out how the users will select records to edit, what screens they will use, and how they will navigate between the screens. Those decisions will lead to requirements describing the forms, navigation techniques, and other features that the application must provide to let the users do their jobs.

NOTE *Moving from goals to requirements often forces you to make some design decisions. For example, you may need to specify form layouts (at least roughly) and the way work flows through the system.*

You might think of those as design tasks, but they're really part of requirements gathering. The following two chapters, which talk about design, deal with program design (how you structure the code) not user interface design and the other sorts of design described here.

The following two sections describe three approaches for converting goals into requirements.

Copy Existing Systems

If you're building a system to replace an existing system or a manual process, you can often use many of the behaviors of the existing system as requirements for the new one. If the old system sends customers e-mails on their birthdays, you can require that the new system does that,

too. If the users currently fill out a long paper form, you can require that the new system has a computerized form that looks similar—possibly with some tabs, scrolled windows, and other format changes to make the form look a bit better on a computer.

This approach has a few advantages. First, it's reasonably straightforward. It doesn't take an enormous amount of software engineering experience to dig through an existing application and write down what it does. (If you're lucky, you might even get the customers to do at least some of it so that you can focus on software design issues.)

This approach also makes it more likely that the requirements can actually be satisfied, at least to the extent the current system works. If an existing system does something, then you at least know it's possible.

Finally, this approach provides an unambiguous example of what you need to do. In the specification, you don't need to write out in excruciating detail exactly how the "Lazy Backup" screen works. Instead you can just say, "The Lazy Backup screen will work as it does in the existing system with the following changes: …."

Even though this approach is straightforward, it has some disadvantages. First, you probably wouldn't be building a new version of an existing system unless you planned to make some changes. Those changes aren't part of the original system, so there's no guarantee that they're even possible. They may also be incompatible with the original system. (Not all pieces of software play nicely together.)

A second problem with this approach is that users are often reluctant to give up even the tiniest features in an existing program. In the projects I've worked on, I've found that no matter how obscure and worthless a feature is, there's at least one user willing to fight to the death to preserve it. If the software has been in use for a long time, it may contain all sorts of odd quirks and peccadillos. You might like to streamline the new system by removing the feature that changes the program's background color to match the weather each day, but that's not always possible.

FOREVER FEATURES

I was once asked to help port part of an application to a new platform. The key piece of the application that the customer wanted to keep was fairly small, and the project manager estimated it would take a few hundred hours of work to get the job done.

When I dug through the original application, however, I found that it included more than 100 forms, each of which was moderately complicated. The system also included interfaces to a number of external databases and automated systems.

At this point, we went back to the customer and asked if they were willing to give up most of those 100+ forms and just keep the key tools we were trying to port.

By now you've probably guessed the punchline. The customer wouldn't give up any of the existing application's features. The project's estimated time jumped from a few hundred hours to several thousand hours, and the whole thing was scrapped.

There is some good news in this tale, however. We discovered the problem quickly during initial requirements gathering, so we hadn't wasted too much time before the project was canceled. It would have been much worse if we had started work only to have the requirements gradually expand to include everything in the original application. Then we would have wasted hundreds of hours of work before the project was canceled.

Using an existing system to generate requirements can be a big time-saver, as long as the development team and the customers all agree on which parts of the existing system will be included in the new one.

Clairvoyance

A lot more often than you might think, one or more people simply look at the project's goals, visualize a finished result, and start cranking out requirements. For example, the project lead might use gut feelings, common sense, tea leaves, tarot cards, and other arcane techniques to cobble together something that he thinks will work. If the project is large, pieces might be doled out to team leads so that they can work on their own pieces of the system, but the basic approach is the same: Someone sits down and starts churning out form designs, work flow models, login procedures, and descriptions of reports.

I'm actually being a bit unfair characterizing this approach as clairvoyance because it's actually quite effective in practice. Assuming the people writing the requirements understand the customers' needs and have previous experience, they often produce a good result. Ideally team leads are chosen for their experience and technical expertise (not because they're the boss's cousin), so they know what the computer can do and they can design a system that works.

This technique is particularly effective if the project lead has previously built a similar system. In that case, the lead already knows more or less what the application needs to do, which things will be easy and which will be hard, how much time everything requires, and which kinds of donuts motivate the programmers the best.

Having an experienced project lead greatly increases the chances that the requirements will include everything you need to make the project succeed. It also greatly increases the chances that the team will anticipate problems and handle them easily as development continues. In fact, this is such an important point, it's a best practice.

BEST PRACTICE: EXPERIENCED PROJECT LEADS

A project's chances for success are greatly improved if the project lead has previous experience with the same kind of project.

The same holds true for the other project members. Programmers with previous experience with the same kind of project will encounter fewer problems and meet their scheduled milestones more often.

Documenters who have written user manuals for similar applications will find writing manuals for the new project easier. Project managers with similar experience will know what tasks are likely to be difficult. Even customers with previous software engineering experience will be better at creating good requirements.

If you have access to design specialists such as user interface designers or human factors experts, get them to help. Any programmer can build forms, menus, and colorful labels, but some don't do a good job. A good user interface makes users productive. A bad one is frustrating and ineffective. (It's like trying to empty a bathtub with a teaspoon. You'll eventually succeed, but you'll spend the whole time thinking, "This is stupid. There has to be a better way!")

Brainstorm

Copying an existing application and clairvoyance are good techniques for generating requirements, but they share a common disadvantage: They are unlikely to lead you to new innovative solutions that might be better than the old ones. To find truly revolutionary solutions, you need to be more creative. One way to look for creative solutions is the group creativity exercise known as *brainstorming*.

You're probably somewhat familiar with brainstorming, at least in an informal setting, but there are several approaches that you can use under different circumstances.

The basic approach that most people think of as brainstorming is called the *Osborn method* because it was developed by Alex Faickney Osborn, an advertising executive who tried to develop new, creative problem-solving methods starting in 1939. Basically, he was tired of his employees failing to come up with new and innovative advertising campaigns. (As is the case with the Gantt charts described in Chapter 3, the fact that we're still using Osborn's techniques after all these years shows how useful they are.) Osborn's key observation is summed up nicely in his own words.

> *It is easier to tone down a wild idea than to think up a new one.*
>
> —Alex Faickney Osborn

Basically, the gist of the method is to gather as many ideas as possible, not worrying about their quality or practicality. After you assemble a large list of possible ideas, you examine them more closely to see which deserve further work.

To allow as many approaches as possible, you should try to get a diverse group of participants. In software engineering, that means the group should include customers, users, user interface designers, system architects, team leads, programmers, trainers, and anyone else who has an interest in the project. Get as many different viewpoints as you can. (Although in practice brainstorming becomes less effective if the group becomes larger than 10 or 12 people.)

To keep the ideas flowing, don't judge or critique any of the ideas. If you criticize someone's ideas, that person may shut down and stop contributing. Even a truly crazy idea can spark other ideas that may lead somewhere promising. Just write down every idea no matter how impractical it may seem. Even if an idea is impossible to implement using today's technology, it may be simple by next Wednesday.

(It wasn't that long ago that portable phones had the size, weight, and functionality of a brick. Now they're small enough to lose in the sofa cushions and have more computing power than NASA had when Neil Armstrong flubbed his "one small step" line on the moon.)

Osborn's method uses the following four rules:

1. **Focus on quantity.** Do everything you can to keep the ideas flowing. The more ideas you collect, the greater your chances of finding a really creative and revolutionary solution.
2. **Withhold criticism.** Criticism can make people stop contributing. Early criticism can also eliminate seemingly bad ideas that lead to better ideas.
3. **Encourage unusual ideas**. You can always "tone down a wild idea" but you may need to think way outside of the box to find really creative solutions.
4. **Combine and improve ideas.** Form new ideas by combining other ideas or using one idea to modify another.

Only after the flow of ideas is slowing to a trickle should you start evaluating the ideas to see what you've got. At that point, you can pick out the most promising ideas to develop further (possibly with more brainstorming).

Many people are familiar with Osborn's method (although they may not know its name), but there are also several other brainstorming techniques, some of which can be even more effective. The following list describes some of those techniques.

- **Popcorn**—(I think of this as the Mob technique.) People just speak out as ideas occur to them. This works fairly well with small groups of people who are comfortable with each other.
- **Subgroups**—Break the group into smaller subgroups (possibly in the same room) and have each group brainstorm. When the subgroups are finished, have the larger group discuss their best ideas. This works well if the main group is very large, if some people feel uncomfortable speaking in the larger group (the new developer in shorts and sandals may be afraid to speak out in front of the corporate vice president in a thousand dollar suit), or if one or two people are monopolizing the discussion.
- **Sticky notes**—Also called the nominal group technique (NGT). Participants write down their ideas on sticky notes, index cards, papyrus, or whatever. The ideas are collected, read to the group, and the group votes on each idea. The best ideas are developed further, possibly with other rounds of brainstorming.
- **Idea passing**—Participants sit in a circle. (I suppose you could use some other arrangement such as an ellipse, rectangle, or nonagon. As long as you have an ordering for the participants.) Each person writes down an idea and passes it to the next person. The participants add thoughts to the ideas they receive and pass them on to the next person. The ideas continue moving around the circle until everyone gets their original idea back. At this point, each idea should have been examined in great detail by the group. (Instead of a circle, nonagon, or whatever, you can also swap ideas randomly.)
- **Circulation list**—This is similar to idea passing except the ideas are passed via e-mail, envelope, or some other method outside of a single meeting. This can take a lot longer than idea passing but may be more convenient for busy participants.
- **Rule breaking**—List the rules that govern the way you achieve a task or goal. Then everyone tries to think of ways to break or circumvent those rules while still achieving the goal.

- **Individual**—Participants perform their own solitary brainstorming sessions. They can write (or speak) their trains of thought, use word association, draw mind maps (diagrams relating thoughts and ideas—search online for details), and any other technique they find useful. Some studies have shown that individual brainstorming may be more effective than group brainstorming.

The following list describes some tips that can make brainstorming more productive.

- Work in a comfortable room where everyone can feel at ease.
- Provide food and drinks. (I'll let you decide what kinds of drinks.)
- Start by recapping the users' current processes and the problems you are trying to solve.
- Use a clock to keep sessions short and lively. If you're using an iterative approach such as idea passing, keep the rounds short.
- Allow the group's attention to wander a bit, but keep the discussion more or less on topic. If you're designing a remote mining rig control system, then you probably don't need to be discussing Ouija boards or Monty Python quotes.
- However, a few jokes can keep people relaxed and help ideas flow, so a few Monty Python quotes may be okay.
- If you get stuck, restate the problem.
- Allow silent periods so that people have time to think about the problem and their ideas.
- Reverse the problem. For example, instead of trying to think of ways to build better blogging software, think of ways to build worse blogging software. (Obviously, don't actually do them.)
- Write ideas in slightly ambiguous ways and let people give their interpretations.
- At the end, summarize the best ideas and give everyone copies so that they can think about them later. Sometimes, a great idea pops into someone's head after the official brainstorming sessions are over.

Brainstorming is useful any time you want to find creative solutions to complex problems, not just during requirements gathering. You can use it to pick problems in your company that you might solve with a new software project. You can use it to design user interfaces, explore possible system architectures, create high-level designs, and plan interesting exercises for training classes. (You can even use brainstorming techniques outside of software engineering to decide where to go on your next vacation, reduce pollution in your city, or pick a school science fair project.)

Keep brainstorming in mind throughout the project as a technique you can use to attack difficult problems.

RECORDING REQUIREMENTS

After you decide what should be in the requirements, you need to write them down so that everyone can read them (and argue about whether they're correct). There are several ways you can record requirements so team members can refer to them throughout the project's lifetime.

Obviously, you can just write the requirements down as a sequence of commandments as in, "Thou shalt make the user change passwords on every full moon." There's a lot to be said for writing down requirements in simple English (or whatever your team's native language is). For starters, the team members already know that language and have been using it for many years.

You can still mess things up by writing requirements ambiguously or in hard-to-understand formats (such as limericks or haiku), but if you're reasonably careful, requirements written in ordinary language can be very effective.

The following sections describe some other methods for recording requirements.

UML

The Unified Modeling Language (UML) lets you specify how parts of the system should work. Despite its name, UML isn't a single unified language. Instead it uses several kinds of diagrams to represent different pieces of the system. Some of those represent program items such as classes. Others represent behaviors, such as the way objects interact with each other and the way data flows through the system.

I won't bash UML (it's too popular and I'm not famous enough to get away with it), but it does have some drawbacks. Most notably it's complicated. UML includes two main categories of diagrams that are divided into more than a dozen specific types, each with its own complex set of rules.

Specifying complex requirements with UML is only useful if everyone understands the UML. Unfortunately, many customers and users don't want to learn it. It's not that they couldn't. They just usually have better things to do with their time, like helping you understand their needs. (I did actually work on one project where the customers taught themselves how to use some types of UML diagrams so that they could specify parts of the system. It worked reasonably well, but it took a long time.)

I'll talk more about UML in the next chapter. For now during requirement gathering, you probably shouldn't rely heavily on UML unless your customers are already reasonably familiar with it. (For example, if you're writing a library for use by other programmers who already use UML.)

User Stories

> *Storytelling strikes me as a more powerful tool than quantification or measurement for what we do.*
>
> —Alan Cooper

A *user story* is exactly what you might think: a short story explaining how the system will let the user do something. For example, the following text is a story about a user searching a checkers database to find opponents:

> *The user enters his Harkness rating (optional), whether moves should be timed or untimed, and the variant (such as traditional, three-dimensional, upside-down, or Gliński). When the user clicks Search, the application displays a list of possible opponents that have compatible selections.*

Many developers write stories on index cards to encourage brevity. The scope of each story should also be limited so that no story should take too long to implement (no more than a week or two).

Notice that the story doesn't contain a lot of detail about things like whether the game variants are given in a list or set of radio buttons. The story lets you defer those decisions until later during design.

User stories should come with acceptance testing procedures that you can use at the end of development to decide whether the application satisfied the story.

User stories may seem low-tech, but they have some big advantages, not least of which is that people are already familiar with them. They are easy to write, easy to understand, and can cover just about any situation you can imagine. They can be simple or complex depending on the situation. Unlike UML, your customers, developers, managers, and other team members already know how to understand stories without any new training.

User stories give you a lot of expressiveness and flexibility without a lot of extra work. (In management speak, user stories allow you to leverage existing competencies to empower stakeholders.)

User stories do have some drawbacks. For example, you can easily write stories that are confusing, ambiguous, inconsistent with other stories, and unverifiable. Of course, that's true of any method of recording requirements.

Use Cases

A *use case* is a description of a series of interactions between actors. The actors can be users or parts of the application.

Often a use case has a larger scope than a user story. For example, a use case might explain how the application will allow a user to examine cardiac ultrasound data for a patient. That user might need to use many different screens to examine different kinds of recordings and measurements. Each of those subtasks could be described by a user story, but the larger job of examining all the data would be too big to describe on a single index card and would take longer to implement than a week or two.

Use cases also follow a template more often than user stories. A simple template might require a use case to have the following fields:

- **Title**—The name of the goal as in, "User Examines Cardiac Data." Usually, the title includes an action (examines) and the main actor (user).
- **Main success scenario**—A numbered sequence of steps describing the most normal variation of the scenario.
- **Extensions**—Sequences of steps describing other variations of the scenario. This may include cases such as when the user enters invalid data or the application can't handle a request. (For example, if the user searches for a nonexistent patient.)

Other templates include a lot more fields such as lists of stakeholders interested in the scenario, preconditions that must be met before the scenario begins, and success and failure variations.

Prototypes

A *prototype* is a mockup of some or all of the application. The idea is to give the customers a more intuitive hands-on feel for what the finished application will look like and how it will behave than you can get from text descriptions such as user stories and use cases.

A simple user interface prototype might display forms that contain labels, text boxes, and buttons showing what the finished application will look like. In a *nonfunctional prototype*, the buttons, menus, and other controls on the forms wouldn't actually do anything. They would just sit there and look pretty.

A *functional prototype* (or *working prototype*) looks and acts as much like the finished application will but it's allowed to cheat. It may do something that looks like it works, but it may be incomplete and it probably won't use the same methods that the final application will use. It might use less efficient algorithms, load data from a text file instead of a database, or display random messages instead of getting them from another system. It might even use hard-coded fake data.

For example, the prototype might let you enter search criteria on a form. When you clicked the Search button, the prototype would ignore your search criteria and display a prefilled form showing fake results. This gives the customers a good idea about how the final application will work but it doesn't require you to write all the code.

There are a couple of things you can do with a prototype after it's built. First, you can use it to define and refine the requirements. You can show it to the customers and, based on their feedback, you can modify it to better fit their needs.

After you've fine-tuned the prototype so that it represents the customers' requirements as closely as possible, you can leave it alone. You can continue to refer to it if there's a question about what the application should look like or how it should work, but you start over from scratch when building the application. This kind of prototype is called a *throwaway prototype*.

Alternatively, you can start replacing the prototype code and fake data with production-quality code and real data. Over time, you can evolve the prototype into increasingly functional versions until eventually it becomes the finished application. This kind of prototype is sometimes called an *evolutionary prototype*. This approach is used by some of the iterative approaches described in Chapter 13.

SURVIVAL OF THE LAZIEST

You need to be careful if you use an evolutionary prototype. While throwing together an initial version to show the customers what the final application will do, developers can (and should) take shortcuts to get things done as quickly as possible. That can result in code that's sloppy, riddled with bugs, and hard to maintain.

As long as the prototype works, that's fine. The prototype is only supposed to give you an idea about how the program will work, so it doesn't need to be as maintainable as the finished application in the long run.

That's fine if you're using the prototype only to define requirements, but if you try to evolve the prototype into a production application, you need to be sure to go back and remove all the shortcuts and rewrite the code properly. If you don't remove all the prototype code, you'll certainly pay the price later in increased bug fixes and maintenance.

Requirements Specification

How formally you need to write up the requirements depends on your project. If you're building a simple tool to rename the files on your own computer in bulk, a simple description may be enough. If you're writing software to fill out legal forms for a law firm, you probably need to be much more formal. (And you might want to hire a different law firm to review your contract.)

If you search the Internet, you can find several templates for requirement specifications. These typically list major categories of requirements such as user documentation, user interface design, and interfaces with other systems.

For example, Villanova University has an example template in Word format at `tinyurl.com/obqhatt`. The North Carolina Enterprise Project Management Office has another one at `tinyurl.com/n7ttqfh`. (These are really long URLs so I used tinyurl to shorten them.)

VALIDATION AND VERIFICATION

After you record the requirements (with whatever methods you prefer), you still need to validate them and later verify them. The two terms validation and verification are sometimes used interchangeably. Here are the definitions I use. (I think these are the most common interpretations.)

Requirement validation is the process of making sure that the requirements say the right things. Someone, often the customers or users, need to work through all the requirements and make sure that they: (1) Describe things the application should do. (2) Describe *everything* the application should do.

Requirement verification is the process of checking that the finished application actually satisfies the requirements.

VALIDATION VERSUS VERIFICATION

Another way to think of this is

Validation—Are we doing the right things?

Verification—Are we doing the things right?

Those two statements are glib, but it's hard to remember which is which. Perhaps a better way to remember the difference is that "validation" comes before "verification" alphabetically and validation comes before verification in a software project.

CHANGING REQUIREMENTS

In many projects, requirements evolve over time. As work proceeds, you may discover that something you thought would be easy is hard. Or you may stumble across a technique that lets you add a high-value feature with little extra work.

Often changes are driven by the customers. After they start to see working pieces of the application, they may think of other items that they hadn't thought of before.

Depending on the kind of project, you may accommodate some changes, as long as they don't get out of hand. You can help control the number of changes by creating a *change control board*. Customers (and others) can submit change requests to this board (which might actually be a single person) for approval. The board decides whether a change should be implemented or deferred to a later release.

The development methods described in Chapters 13 and 14 are particularly good at dealing with changing requirements because they tend to build an application in small steps with frequent opportunities for refinement. If you add new features in a mini-project every two weeks, it's easy to add new requirements into the next phase. There's still a danger of never finishing the project, however, if the change requests keep trickling in.

SUMMARY

Requirements gathering may not be *the most* important stage of a project, but it certainly is *an* important stage. It sets the direction for future development. If you get the requirements wrong, you may develop something but there's no guarantee that it will be useful.

Good requirements must satisfy some basic requirements of their own. For example, they must be clear and consistent. Having hundreds of requirements won't do you any good if no one can understand what they mean or if they contradict each other.

Some developers group requirements into categories. For example, you can use audience-oriented categories, FURPS, or FURPS+ to organize requirements. Categorizing requirements alone doesn't help you define the project, but you can use the categories as a checklist to make sure you haven't forgotten anything obvious. (They also make it easier to understand other software engineers at parties when they say, "This party has good functional requirements but the nonfunctionals could use some work!")

There are several ways you can gather requirements. Obviously, you should talk with the customers and, if possible, the users. You can use the five Ws and one H to help guide the conversation. Studying the users as they currently perform their jobs is often instructive. It can help clarify the project's goals, and occasionally you may discover simple things you can add to the project that will make the users' jobs a whole lot easier.

After you understand the customers' needs, you must refine those needs into requirements. Three techniques that can help include: copying an existing system, using previous experience to just write them down, and brainstorming. Brainstorming is often more work but can sometimes lead to creative solutions that you might not have discovered otherwise.

Finally, after you know what the requirements are, you need to record them so everyone can refer to them as the project continues. Some ways you can record requirements include formal written specifications, UML diagrams, user stories, use cases, and prototypes.

Before you move on to the next phase of development, you should validate the requirements to ensure that they actually meet the customers' needs. (Later, near the end of the project, you'll also need to verify that the project has met the requirements.)

If you think this seems like a lot of work before the project "actually" begins, you're right. However, it's critical to the project's eventual success. Without sound requirements, how will you know what

to build? Unless the requirements are clear and verifiable, how will you know if you've achieved your goals?

After you gather, record, and validate the requirements, you're ready to move on to the next stage of development: high-level design. You may have already incorporated some design decisions into the requirements. For example, you might have made some user interface decisions or picked a system architecture.

The next chapter explains some of the high-level design decisions you might need to make, whether in the requirements phase or during a separate high-level design step. It also provides some guidance on how to make those decisions.

EXERCISES

1. List five characteristics of good requirements.
2. What does MOSCOW stand for?
3. Suppose you want to build a program called TimeShifter to upload and download files at scheduled times while you're on vacation. The following list shows some of the application's requirements.
   a. Allow users to monitor uploads/downloads while away from the office.
   b. Let the user specify website log-in parameters such as an Internet address, a port, a username, and a password.
   c. Let the user specify upload/download parameters such as number of retries if there's a problem.
   d. Let the user select an Internet location, a local file, and a time to perform the upload/download.
   e. Let the user schedule uploads/downloads at any time.
   f. Allow uploads/downloads to run at any time.
   g. Make uploads/downloads transfer at least 8 Mbps.
   h. Run uploads/downloads sequentially. Two cannot run at the same time.
   i. If an upload/download is scheduled for a time when another is in progress, it waits until the other one finishes.
   j. Perform scheduled uploads/downloads.
   k. Keep a log of all attempted uploads/downloads and whether they succeeded.
   l. Let the user empty the log.
   m. Display reports of upload/download attempts.
   n. Let the user view the log reports on a remote device such as a phone.

o. Send an e-mail to an administrator if an upload/download fails more than its maximum retry number of times.

p. Send a text message to an administrator if an upload/download fails more than its maximum retry number of times.

For this exercise, list the audience-oriented categories for each requirement. Are there requirements in each category?

4. Repeat Exercise 3 using the FURPS requirement categories.

5. What are the five Ws and one H?

6. List three techniques for gathering requirements from customers and users.

7. Explain why brainstorming can be useful in defining requirements.

8. List the four rules of the Osborn method.

9. Figure 4-1 shows the design for a simple hangman game that will run on smartphones. When you click the New Game button, the program picks a random mystery word from a large list and starts a new game. Then if you click a letter, either the letter is filled in where it appears in the mystery word, or a new piece of Mr. Bones's skeleton appears. In either case, the letter you clicked is grayed out so that you don't pick it again. If you guess all the letters in the mystery word, the game displays a message that says, "Congratulations, you won!" If you build Mr. Bones's complete skeleton, a message says, "Sorry, you lost."

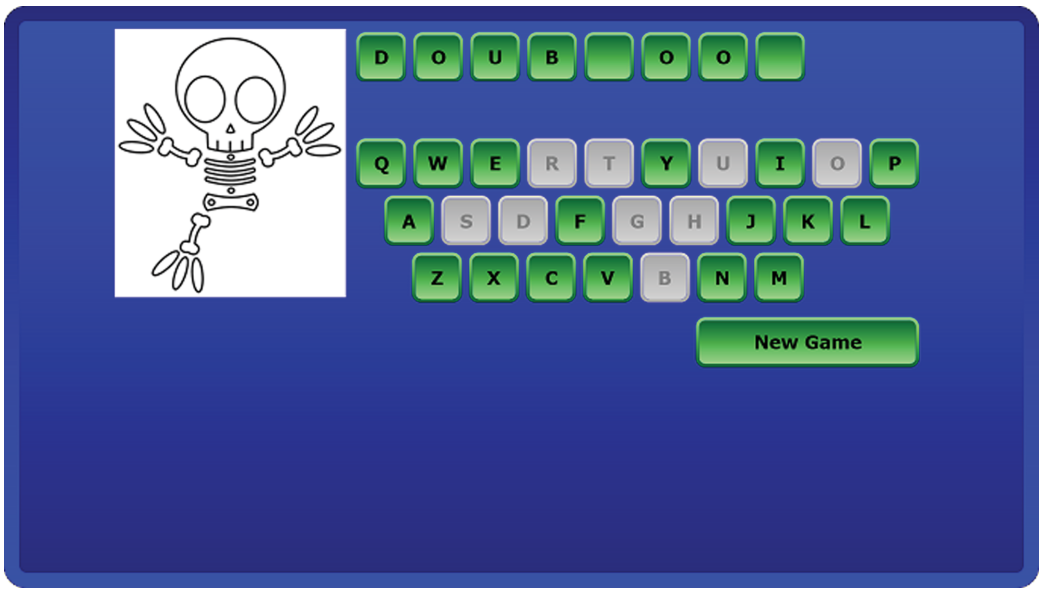

FIGURE 4-1: The Mr. Bones application is a hangman word game for Windows Phone.

Brainstorm this application and see if you can think of ways you might change it. Use the MOSCOW method to prioritize your changes.

10. (Instructors) Have the class brainstorm ideas to address a fairly difficult issue (such as reversing global warming, ending global hunger, or making politicians honor their campaign promises). If time permits, try a couple different brainstorming variations such as popcorn, subgroups, and individual. Discuss what went well and what didn't.

WHAT YOU LEARNED IN THIS CHAPTER

- Requirements are important to a project because they set the project's goals and direction.
- Requirements must be clear, unambiguous, consistent, prioritized, and verifiable.
- The MOSCOW method provides one way to prioritize requirements.
- FURPS stands for Functionality, Usability, Reliability, Performance, and Supportability.
- FURPS+ also adds design constraints, implementation requirements, interface requirements, and physical requirements.
- You can gather requirements by talking to customers and users, watching users at work, and studying existing systems.
- You can convert goals into requirements by copying existing systems and methods, using previous experience to write them down, and brainstorming.
- You can record requirements in written specifications, UML diagrams, user stories, use cases, and prototypes.
- Requirements may change over time. That's okay as long as it happens in a controlled manner.
- Requirement validation is the process of checking that the requirements meet the customers' needs.
- Requirement verification is the process of checking that the finished project satisfies the requirements.