

Chapter 5

Describing Your Software for Outsourcing

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

—Douglas Adams, author of
The Hitchhiker's Guide to the Galaxy

What if you could outsource your software development and have a guarantee that the results would be successful? That would be very powerful indeed.

A guarantee is one of the most attractive enticements you can offer your customers to help them decide to buy your product. Actually, marketing folks give such incentives the fancier name of “risk reversal,” since they are meant to reverse the feeling of risk within your customers. Whatever you offer as a risk reversal should make them feel that it is less risky to do business with you than it is to continue on their present course. The customer should feel that they have a great deal to gain and very little to lose.

The most common risk reversal is a guarantee. (A free trial is another common risk reversal technique). With a guarantee, not only will your customer get the benefits of your product or service for a reasonable price, but they are also promised guaranteed results.

Do you offer some kind of risk reversal for your business? Perhaps you guarantee the major benefits offered by your software. Or you may offer a free trial for some limited period of time. The more you work to earn the trust of your customers, the easier you will make it for them to do business with you.

Why don't most software outsourcing service firms offer a guarantee? Because developing custom software on time is notoriously difficult to do. An on-time delivery guarantee is virtually never offered for software development, and it would seem hard to believe if it were.

This is mainly because the end goal of a software development project is difficult to define and is not often completely known at the beginning. The competence and ability of the software engineers is rarely the issue, especially if you have carefully selected your vendor using the techniques described in Chapter 3.

The key, you might think, is in defining your software as carefully as possible. If you do this well, it seems, you will reverse your risk in outsourcing your software development. But it is not always so simple.

A Modular Approach

You need to make a fundamental choice concerning your outsourcing. Will you outsource your entire software application? Or just software modules you then assemble into a complete application yourself?

Outsourcing individual modules works well if the modules require specific technical expertise. It can be easier and more cost-effective to find the expertise you need outside your company compared to hiring expert employees. In this case you have to take responsibility for managing your outsourcing team and integrating their work with the software you create internally.

You must also take responsibility for good specifications for your outsourced modules. Typically in this situation, the more traditional "waterfall" techniques of software specification are used.

The Old (and Obsolete?) Waterfall Method

In the past, efforts to make the software development process more productive focused on creating better specifications. As the theory went, you should first try to do a better job of capturing the require-

ments of *what* your software is supposed to do, including both features and responsiveness.

Second, you should create a good design and description of *how* the requirements are to be achieved. Then you can start coding. With enough detail in your specifications and design documents it seems you should get guaranteed results.

The waterfall²² approach is still the recommended way to communicate the desired behavior of your software when you are outsourcing individual modules or reimplementing of an existing system. But it is rare to know all the details of what your software should do when you are at the beginning of a new project. It seems crazy to expect guaranteed results in this kind of situation.

And it is.

Observant software engineers have realized that they jump back and forth between the different waterfall boxes during the course of developing new software. Enforcing a sequential process of requirements specification, design, and then development is artificial and counterproductive for new software application development. Techniques for specifying new software applications using agile methods are discussed later in this chapter.

Architecting Your Software for Outsourcing

One reason to choose the modular approach is to identify the software modules and subsystems that can safely be outsourced, while keeping most of your software private. Intellectual property is the lifeblood of a company, and your software ideas should be guarded carefully.

Most offshore software development companies will fiercely defend your intellectual property as part of their normal business procedures. Nevertheless, you may want to carve out the most sensitive parts and develop them internally.

Or you can parcel out the development of your software to multiple independent outsourced teams. That way no one team will have all of the intellectual property.

22. This old style of software design is called the waterfall method because when you draw boxes around all of the sequential steps and then pour water into the first one, it will fall through each box—from the Requirements box to the Design box and so on down through the Coding and Testing boxes until you get to the end of the project.

The architecture of your software represents a high-level view of its internal structure. It defines how the various modules and technical components of your product are related. Your software architecture should also take into consideration how the software will be developed. Both the structure of your software and whether all or part of your software development will be outsourced have a tremendous impact on the architecture of your software.

Similarly, the architecture of your software will have a major influence on how you outsource and your decision to use outsourcing at all. If you want to keep your core intellectual property in-house, you must have an architecture that supports isolation of the core intellectual property from any outsourced development you want to do.

Three Architectures for Outsourcing

There are three ways to architect your software to take advantage of outsourcing. The first is to partition your software into pieces that can be developed independently. Some of the independent modules can be developed with outsourcing, and others can be developed with your internal team. An integration step is usually handled by the internal team before product release. Loose “coupling” between these modules is required to make this work well.

For example, one company outsourced the development of a new product based on existing technology created by their internal programming team. Both teams worked independently because the original product architecture had a well-documented set of modules (the internally developed technology) that was used as building blocks for new products (like the one created by the outsourced team).

A second approach is to integrate the work of your internal and outsourced development teams in a truly collaborative effort. This approach works best when both teams are involved with the architectural decisions. The architecture not only identifies the technical modules of your product, as in the first approach, but also reflects the areas of technical strength in the development teams involved.

As an example of this second approach, suppose your product has both an HTML human interface and an XML file interface. If your outsourced team has great expertise with XML file processing, your internal team may then focus on the user interface. If the teams are equally skilled, you can divide the work by use cases and let each

team develop all levels of the product for independent sets of functionality. In an online catalog system, for example, one team could do the buyer interface and the other could create the catalog maintenance features used by the seller.

The third approach is to outsource the development of your entire product. In this case, the architecture should be defined with input from the developers and is an important tool for communicating the design of your software to new members of the team.

Marchitecture and Tarchitecture

Many software executives are most comfortable with the approach of outsourcing individual modules. If the programming tasks are well defined, then not much thought or creativity (and therefore potential for delay) is necessary.

However, many outsourcing teams are smarter than this and actually seek out challenging assignments. They understand what is important in the process of creating your software and contributing to the success of your company. If you treat outsourced developers like cattle, as with cattle, you may not be satisfied with what they leave behind.

So far I have been talking about the technical architecture of your software, sometimes called its “tarchitecture.” The tarchitecture is defined by a technical architect—an engineer with a strong programming background.

Sometimes your software architecture will be at such a high level that it will not seem very technical at all. There is also another less technical and more marketing-oriented architecture that helps define the customer options for your software product.

The two architecture types are sometimes confused, as they were at a visit to a client. A venture capitalist interested in investing in the company was also present at the meeting. Actually, the VC was really excited by the technology and was effusive in his praise of the technical co-founder of the company.

I also thought the technology was interesting, but I was having trouble visualizing how the software would be used. So I created a diagram showing how all the modules fit together. This was a tarchitecture diagram.

The technical founder did not see much need for documentation, but he showed the VC my diagram anyway. “Oh that’s just

‘marchitecture!’” the VC said, his tone of voice clearly indicating that it was of little value. He did a good job of intuitively sensing what the techie wanted to hear.

That was not the first time my work has been dismissed by a VC, and I am sure it will not be the last. But there are two problems with this situation (three, if you count how his emotions were being manipulated by the VC).

First, the company really did need a good description of the architecture to better communicate the product requirements, both to customers and newly hired engineers. A well-defined architecture enables engineers to “own” the piece of the software they will develop.

Second, my diagram showed how the internal technical modules were interrelated. As such, it was not a marchitecture diagram. Even if it was, it should not be so easily dismissed.

According to Luke Hohmann in *Beyond Software Architecture*, “Marchitecture is the business perspective of the system’s architecture.” It shows the structure of the software from your customer’s perspective, indicating what options are available and how it can be licensed and installed.

Clearly, your marchitecture is critical to the success of your software and possibly your entire company. When you develop a product to sell, it is best to have some idea of how it will be positioned and offered for sale in the marketplace.

Marchitecture is related to outsourcing when it affects what licensing and installation options need to be developed. Both marchitecture and tarchitecture should be designed collaboratively with your outsourcing team when they are developing your entire software application.

Define the marchitecture and tarchitecture of your new software carefully if you want to take advantage of outsourcing. If you already have an existing software application, your architecture can help you decide which are the best portions to outsource.

In addition, as discussed in Chapter 1, if you have an existing version of your software, you can consider three ways to use outsourcing: (1) create your new version in-house and outsource maintenance of the existing version, (2) outsource development of your new version and continue to maintain the old version with an in-house engineering team, or (3) outsource both and use the internal team to manage your outsourced resources effectively.

The Whole Enchilada

What if you decide to outsource the creation of your entire software application? Can you really specify your software application completely? Will you know all the details of what you and your users will want the software to do? Sometimes you can.

If you are rewriting an existing application—perhaps converting a client/server application to a web application—then you have a well-defined goal. You want to make the web application behave as much like the client/server application as possible. Your existing software functionality, user documentation and old specifications are a great start for defining the new system. You have a pretty good chance of defining the software completely.

However, if you have only broadly defined goals with limited details about what your software will do, the old waterfall techniques are not the best approach. I will discuss this issue in more detail later in this chapter.

What Will Your Software Do?

Software requirements tell *what* your software should do. The design describes *how* the requirements will be implemented. But there is another “how” lurking within your software requirements: How will your users use your software to achieve their goals? Describing the desired behavior of your software is a key part of your software requirements.

There are two approaches to creating software. First is the “build it and they will come” *Field of Dreams*²³ approach. In this case, you do not have to write a description of the software; you just build it yourself. End users and customers are expected to find value in your creation. This actually works sometimes, with eBay and Friendster²⁴ being notable examples.

23. *Field of Dreams* was the very popular 1989 movie starring Kevin Costner, who plays an Iowa farmer who plows under most of his corn crop and builds a baseball field. He did this after hearing a voice that tells him, “Build it and he will come.” Many programmers act as though a similar voice delivers the requirements of the software they write.

24. Who knew we needed online auctions before eBay and that the Internet could enable more efficient use of our social network for dating until there was Friendster?

The second, more common approach is to first gain a reasonable understanding of a market or department/business unit need. Sometimes the market need will only be recognized in the future. Either way, you must first describe the need or goals of the future users of your software application so the software you create has a chance of being successful.

Nobody, least of all programmers, wants to spend an excessive amount of time writing *about* the software. But you must write something in order to have an increased chance of success. What is the minimal amount of writing you need to do to describe your software product effectively? Of course, the answer is, “It depends.”

Your software may be simple or complex, mission-critical or intended for casual use. In all cases, it is important to describe what users can expect to accomplish by using your software.

Using Use Cases

The use case is the main tool of the business analyst, marketing professional, and software designer to describe how your software is used. Written use cases are an extremely useful tool for describing your software. They explain how users interact with your software and attempt to achieve their goals. Use cases should be a major part of the requirements of your product.

Use cases are part of the Unified Modeling Language,²⁵ or UML. UML is a set of tools and techniques to define and describe the behavior and architecture of a system and has great popularity in the design of software applications and systems. Use cases include both a high-level diagram as well as a detailed description in text. The level of detail and precision of use cases is discussed in the next section.

The length and number of your use cases depends on two primary factors: your software’s complexity and the expertise of your development team with the application area.

For example, I once managed an outsourced development team that had previously developed a software application very similar to what we needed, but with a different style of user interface. All the major use cases were already well understood by the developers,

25. Describing all the elements of the UML in detail is not appropriate here, but the bibliography and the book’s web site contain references to excellent books that explain all the diagrams, methods, and tools in the UML toolbox.

and we needed only brief descriptions of use cases for a few new features.

A significant amount of time was spent designing the new user interface however. Actually, use case writing and user interface design are separate tasks, which we will discuss in more detail later.

When you are creating a new software application then you need to create your use cases from scratch. The best way to start is to identify the kinds of users, or actors, that will interact with your software.

Acting Like a User

Actors are the various users of your software and the roles that they play. For example, an order management system might have buyer, salesperson, and shipping clerk as actors. The goals of the buyer actor might include looking up past orders placed, placing a new order, changing an order, canceling an order, and so on.

In some cases, actors can be other systems and equipment that your software interacts with. For example, the Enterprise Resource Planning (ERP) system sends a purchase order document to your software, or a heat sensor sends a message about an excessive temperature.

Your software should help your users (actors playing roles) achieve specific goals. These goals must be aligned with the benefits your software delivers. Who are the users of your software, and what are their goals? Are there multiple user types, or is every user the same?

What Do You Mean, Precisely?

One of the best books about use cases is Alistair Cockburn's *Writing Effective Use Cases*. This book covers all the elements of good use case writing, and the emphasis is on writing—how much detail, what you should say, and how you should say it.

The material in this section draws on the content of Cockburn's book. It is summarized and paraphrased here to give you a good overview of this important tool for defining what your software should do.

Use cases tell your developers what they cannot discover for themselves, especially if they are several oceans away. Use cases tell them what your users want, need, and expect your software to do.

If you can explain that, then good engineers can figure out how to build the software.

But how much detail do you need to provide? If you are spending more time describing your software than it would take to write it, something is wrong. You should start out with a low level of precision in your use cases and then increase the amount of detail needed to communicate with your engineers effectively. There is no point in providing more detail than is needed.

In Table 5-1, the amount of precision you need in your use cases is described as low, medium, or high. Low levels of precision are okay for a development team familiar with the application area, or to get started quickly on a prototype when details are not yet known. In the software project needing just a new user interface that I described earlier, I was able to use low precision for use cases and medium precision for the user interface.

In another project to create a web application product, we used medium precision for the use cases and high precision for the user interface. Error scenarios were straightforward and were added later as they were encountered in the development of the product. The user interface was created in HTML before any coding was done. Then the HTML pages were linked together as a demo to show early customers and investors.

Medium or high levels of precision are often required to effectively outsource software development offshore. The lowest level of precision for use cases involves listing the actors and their goals in using your software product. If you provide only low-precision use cases, you will need more frequent iterations and a higher level of collaboration with the offshore software development team.

Table 5-1. Amount of Precision Needed in Use Cases and User Interface Design

	Precision		
	Low	Medium	High
Use Cases	List of actors and goals	Use cases with success scenarios	Use cases with success and error scenarios
User Interface Design	Screen flow diagram	Screens with data entry and display items and actions defined	Screens defined in HTML or other useful code

Table 5-2. Basic Template for a Use Case

Use Case Element	Description
Scope	The portion of the software under discussion.
Primary actor	The stakeholder or user who initiates an interaction with the software to achieve a goal.
Goal level	The level of the goal (low, medium, or high, as described in Table 5-3).
Stakeholders and interests	People and entities having a vested interest in the behavior of the software
Preconditions	What must be true before the use case runs?
Guarantees	What must be true after the use case runs?
Main success scenario	What happens when nothing goes wrong?
Exception scenarios	What can happen differently during each scenario?

The Gory Details

For medium and high levels of precision, each use case requires more information. The basic template for a use case includes about eight elements, as shown in Table 5-2, with only the first three used in an abbreviated, low-ceremony use case.

In complex products with many actors, goals, and/or scopes, you may feel overwhelmed by the potential number of use cases. One way to organize them is to assign priorities (high, medium, and low) by scope and goal level, as shown in Table 5-3.

Start with the use cases that describe the interaction of individual users with your product. If you have scenarios that involve multiple users or actors using your product, describe them next. Then describe scenarios in which a single user interacts with multiple systems within the environment in which your product runs.

Table 5-3. Setting Priorities by Scope and Goal Level

		Goal Level		
		Multiuser	User	Subfunction
Scope	Environment	medium	medium	low
	System	medium	high	low
	Component	low	low	low

Of lowest priority are interactions between users, subfunctions, and low-level components. Subfunction-level goals are those required to carry out user goals. They are usually shared by other use cases and are separated out to simplify the design. For example, a software system that archives every database transaction in a second database could have low-level components that automatically carry out the data copy. The sequence of steps to do the archiving is important but unlikely to be complex or of high priority for detailed use case specification.

Of course, if your product is complex and its behavior is described by use cases that cover all of the scope/goal level combinations, you will eventually need to write them all. You can use the table to organize and structure your efforts.

Each use case has multiple scenarios. The main scenario is called the success scenario, in which the user does everything correctly and nothing goes wrong. Variations of the main success scenario describe errors and exceptions that can occur and how they are handled. A high level of precision requires you to describe as many of these scenarios as possible.

Use cases are similar to user stories, used for planning in the agile development methodologies discussed later. They may also be decomposed into subfunctions or sub-use cases that are reused in other parts of the software. Subfunctions may also be used as story points for more accurate estimation of programming time required.

Get a Handle on Your User Interface

Writing use cases is different from designing the user interface for your software. If you have a user interface designed, it is best not to let your use cases degrade into descriptions of how to work the user interface (click this, then that, etc.).

This is because a use case is supposed to describe, at a higher level, the goals and intentions of your software's users. Creating use cases is a way to draw out desired software behavior from the users and stakeholders, not a lecture on how the system will work.

However, designing the user interface concurrently while writing the use cases will provide valuable insights into the behavior of your software. New functions will be discovered, and others may be found to be impractical.

Furthermore, as the old saying goes, “A picture is worth a thousand words.” Providing screen shots or screen sequences is a very effective technique for communicating the desired behavior of your software to an outsourced development team.

I recommend developing the user interface while capturing the use cases as a way to confirm what you are hearing from users and stakeholders. The user interface design is a common device for both confirming system function and specifying the software to developers.

A Picture Is Worth a Thousand Words

The point of this section is not to show you how to design an auction web site. I am using eBay as an example of a familiar web site to illustrate the visual technique I recommend for specifying your software.

Even if you are not developing a web application, you should use some sort of visual approach when specifying your software. You can use HTML to emulate your actual user interface. Or you can use a drawing program like Visio or even PowerPoint to design your screens and specify their logical sequence.

First let’s look at the standard, non-visual, text-based approach to specifying use cases.

Imagine that you are building an auction web site and you need to specify the functionality to your outsourced software development team. You have done some reading and have decided that use cases describing how the buyers and sellers will carry out auction activities are the best way to define what the software should do.

So far, so good. Let’s see what a use case will look like for a buyer (the use case actor or role) who wants to place a bid on an item.

Suppose the buyer is a stamp collector. He does a search and finds a stamp to bid on. There can be multiple ways to search for a stamp, and these could be covered in separate use cases.

By definition, use cases consist of text. They contain the sequence of steps the user carries out to accomplish the task. Here is how a buyer bids on a stamp:

1. The user Steve Stamps²⁶ finds a collectible postage stamp to bid on in the system. (Use cases 8 to 14 describe searches.)

26. Cute and memorable names can make your use cases easier to read, and they make it easier for both the target user and the programmer to visualize how the software will be used.

2. Steve displays the page containing a stamp auction item. This page contains an image of the stamp, the starting bid, the time of auction close, and the number of bids on this item.
3. Steve decides to try to buy this stamp. He clicks on the Place Bid button.
4. A page is displayed with the item title, the current bid, and a text field where Steve can enter his maximum bid.
5. Steve enters his bid in the text field and clicks on the Continue button.
6. A new page is displayed showing the item title and Steve's bid. There is a Confirm Bid button and also other text stating that Steve must agree to purchase the item if he is the winning bidder. Steve clicks on the Confirm Bid button.
7. The original item page is displayed as before, except that the number of bids has increased and Steve is now the high bidder if his maximum bid is higher than the previous bidder.

This is a simple use case that shows how a buying user named Steve places a bid on an item in the auction web site. From this text we learn several things. For example, a bid is carried out in a sequence of different pages. The item for sale has several attributes, including title, picture, minimum bid, number of bids, current bid amount, and end time.

It should also raise some questions, especially from a reader who must figure out how to implement these pages. Can an item have more than one picture? Should the picture(s) be shown on the bid page (step 4)? Does Steve need to be signed into the auction site? What happens if he is not signed in? And so on.

Then there are the exceptions, when not everything goes according to plan. The steps presented in the use case assume that Steve is a rational person and generally knows what to do. But suppose he makes a mistake—like not signing in before bidding. Or what if he puts in a bid amount that is less than the current high bid? These and other exceptions will need to be dealt with when the software is written.

Sometimes the way to handle an exception is straightforward and can be left to the software engineer to create. Other times, you will need to specify in detail what the software should do.

Now, are you ready to start coding this feature?

Maybe. Maybe not. You may want to see a couple more use cases to get a better sense of what the software will do.

And if you are writing the use cases, you need to have them reviewed by the actors, the buyers and sellers in this example, to make sure you have the functionality they need.

That leads to the next problem. It can be tough to get your target users to read through a bunch of use cases and have them visualize what the software is supposed to do. It will take some dedication on their part to keep focused and give you good feedback.

After a while their attention will wander. You will probably need to spend time with them in a live meeting to read the use cases with them, hold their interest, and then judge their true reactions. If they are your future customers, or users within your own company busy with other tasks, it is unlikely that you will get them to volunteer the feedback you need to validate the use cases.

What would be better is a demo of the software. But how can you do that if the software is not written yet?

Here's how. You can create mockups of the pages or screens of your user interface in HTML. Then you can link them together in a storyboard kind of demo that shows the use cases in living and moving color.

Here is that use case again, now illustrated with screen shots of a demo made up of HTML pages as a mockup of the user interface.

- 1.** User Steve Stamps finds a collectible postage stamp to bid on in the system. (Use cases #8 to 14 describe searches.)
- 2.** Steve displays the page containing a stamp auction item. This page contains an image of the stamp, the starting bid, the time of auction close, and the number of bids on this item. (See Figure 5-1.)²⁷
- 3.** Steve decides to bid on this stamp. He clicks on the Place Bid button. If Steve is not signed in, he is asked to do so. (See Figure 5-2.)

27. Okay, I am obviously cheating here. I did not craft these pages in HTML; I just took screen shots of eBay. But you can easily design your web pages from scratch. If you are artistically challenged, as I am, you can use low-cost HTML templates to add nice graphics and menus to your pages. Go to the book's web site, www.SoftwareWithoutBordersBook.com, for links to sites where you can obtain these kinds of templates.

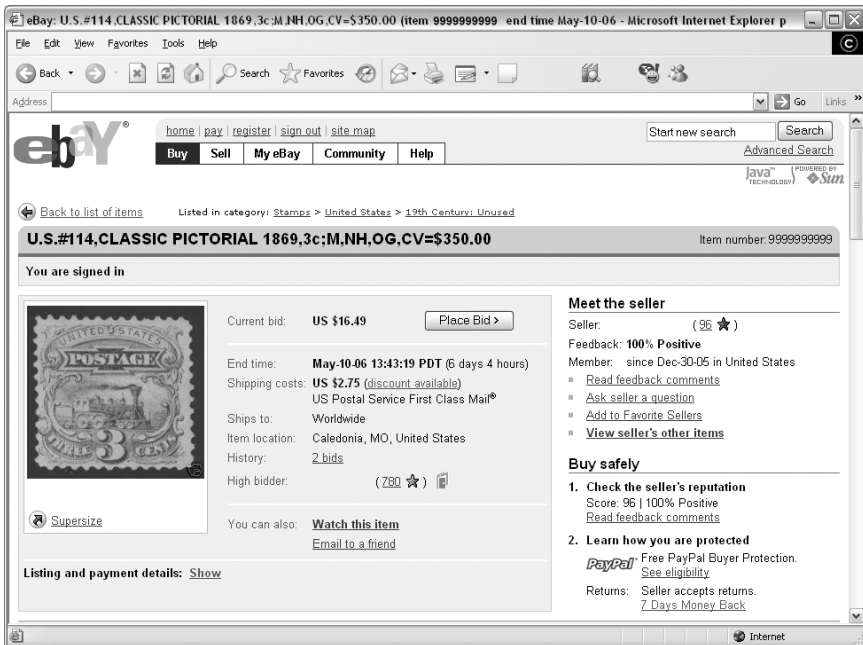


Figure 5-1. Auction Listing

4. A page is displayed with the item title, the current bid, and a text field where Steve can enter his maximum bid. (See Figure 5-3.)
5. Steve enters his bid in the text field and clicks on the Continue button.
6. A new page is displayed showing the item title and Steve's bid. There is a Confirm Bid button and also other text stating that Steve must agree to purchase the item if he is the winning bidder. Steve clicks on the Confirm Bid button. (See Figure 5-4.)
7. The original item page is displayed as before, except that the number of bids has increased and Steve is now the high bidder if his maximum bid is higher than the previous bidder.

Now isn't that a whole lot clearer? Isn't it more interesting to look at? Of course, these screen shots are rich in additional detail. This detail comes from other use cases that use the same pages.

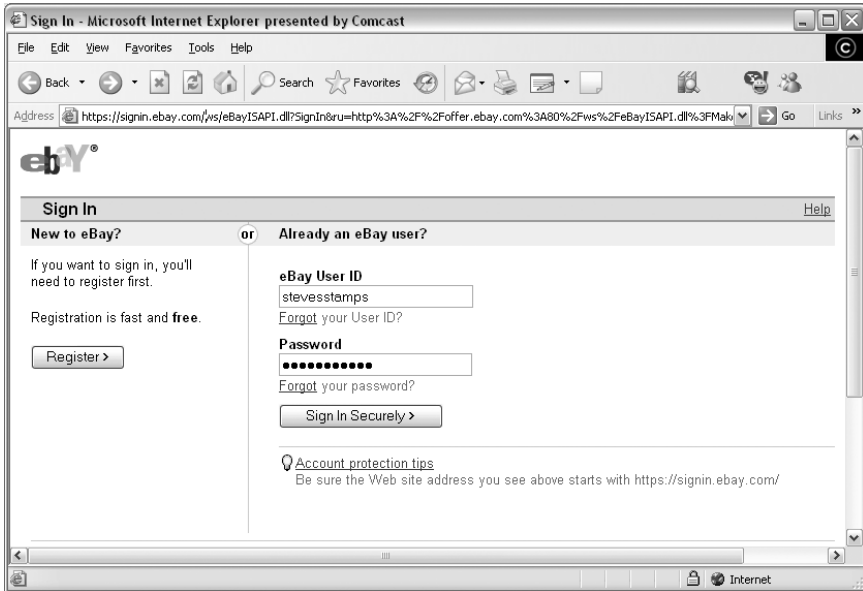


Figure 5-2. Sign-in Screen

For example, the item display page shows a lot of information about the seller. That wasn't mentioned in the use case. And the sign-in page says something about Microsoft Passport. Another use case can describe how that is used.

As you can see, there are many, many use cases that you could specify to completely define your software. But all that text by itself would be very boring!

I recommend that you start with the design of your pages and add details to the pages as new user scenarios are defined. Link the pages together to tell the story of your use cases as a demo. That will bring them to life and simulate how your software will actually run.

Then later, take screen shots of the pages and put the text of the use cases around them, as I have done here. Now you have a relatively complete specification on paper and a demo in color that shows what your software should do.

You can share the demo with your outsourced development team on the web using webinar software like WebEx or GoToMeeting. Or you can record your demo using software like Camtasia that captures the screens as well as your voice to tell the complete story and bring your most important use cases to life. See the book's web

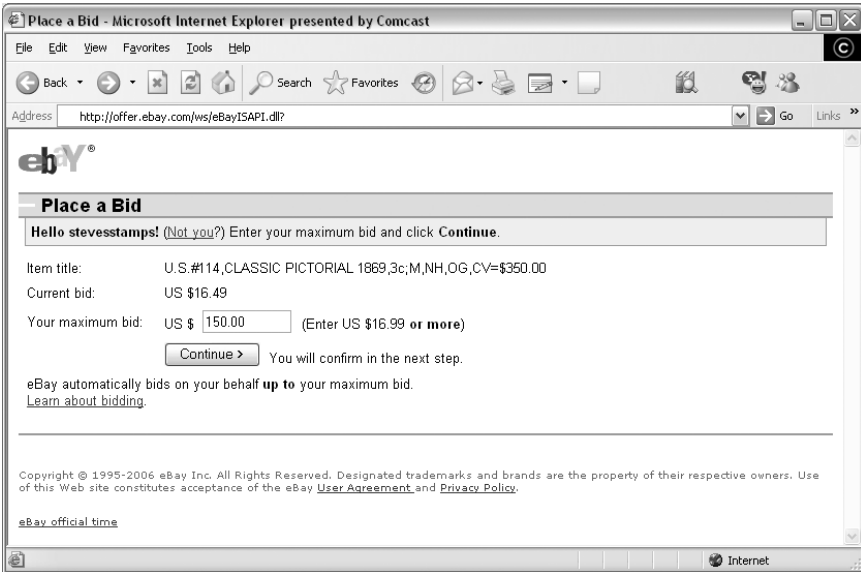


Figure 5-3. Initiating a Bid

site for links to these and other tools you can use to document your software design.²⁸

However, designing the user interface does not provide enough information about user intentions and goals. Effort must also be put into creating written descriptions of product behavior. Screen shots or diagrams are not enough.

Business Vision and Technical Skill

Creating a software product requires two kinds of skills. Let's call them business vision and technical expertise. Excellent business vision provides a deep understanding of market needs, and technical expertise is the ability to create quality software.

Either one by itself is insufficient to guarantee success. Successful organizations recognize that these two skills must be married together in a collaborative team effort to create great software in a timely fashion. The key is for the business visionary half of the team to effectively communicate what the software should do and the

28. Links to these and other software services and products are listed on the www.SoftwareWithoutBordersBook.com web site.

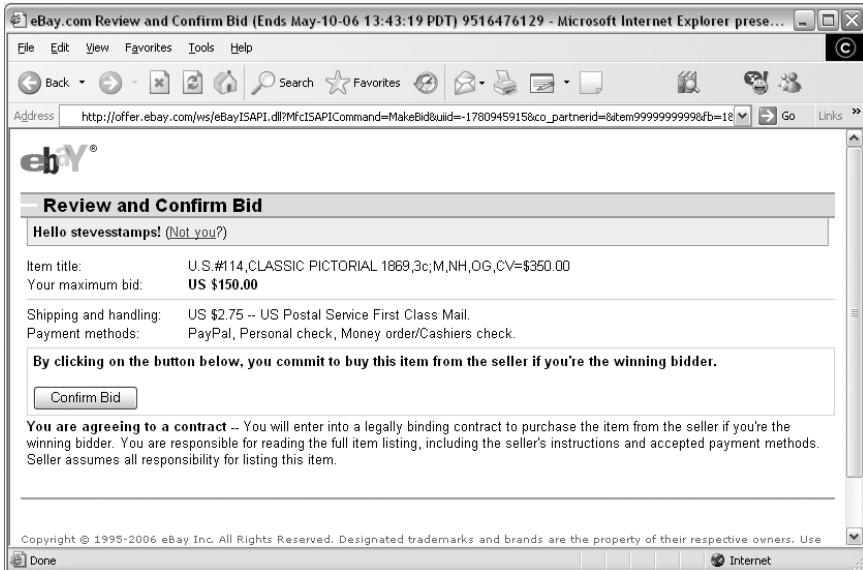


Figure 5-4. Confirming a Bid

technical half to manage engineers to deliver quality software as efficiently as possible. Without this balanced contribution, the vision of the business becomes a hallucination that never becomes a reality for others, and the application created by the technical expert becomes a science project that no one will ever want to use.

There are many good techniques and methods for describing the requirements and the design of software. For example, UML, discussed earlier in this chapter, contains many diagrams and methods for describing software functionality and architecture. These methods are usually used by the technical half of a successful team, which also makes choices requiring technical skill, such as software architecture, computer languages, and databases.

In contrast, someone possessing business vision does not usually care about these technical details. Whether the software uses Oracle or MySQL, Java or C# probably does not matter. Instead, the business visionary knows that Mary over in purchasing is up to her elbows in alligators with the problems of getting her job done. A new software system with three specific features will rein in the alligators, deliver the benefits Mary needs, and save her company money. A person with business vision knows how much money Mary's company

will save, how much Mary will pay for the software, and how many Marys are out there in the world with the same problem.

You need both technical and business vision to create great software. And if you want other people to write and/or test your software, you need to describe what the software needs to do.

Creating Your Software Release Roadmap

Your software release roadmap is where you plan out the future releases of your product, along with the major features and benefits they will provide. Your product roadmap usually comes out of the process of gathering product requirements.

For example, at one of my software companies we had product releases planned out for a year and a half. This was mainly because our minds went wild imagining the product features customers would want. Or what we thought they would want.

We knew customers would not wait for a super-duper product release containing all the features we envisioned. We budgeted out the features according to need, difficulty, and customer desire. It is also difficult to design and develop a large software system all at once. Divide and conquer is a better strategy.

We assigned features to the releases in a product roadmap according to our estimates of their usefulness and difficulty of implementation. Features that were less useful and harder to implement were pushed out to a later release. Then we would show customers and prospects the roadmap. If they drooled over a future feature, we would try to move it up to a more near-term release.

Having a product roadmap is important for outsourcing because it helps you accurately judge the size of the engineering team you need and how long you will need them. Defining future features and releases also helps you avoid architecting yourself into a corner. You don't want to have the, "Oops! I didn't know it had to run on a Macintosh!" problem.

Extracting Use Cases from "Experts"

It is common for "subject matter experts" to be unaware of software design techniques. In such cases, outside help is required to create a useful description that communicates product requirements to developers.

A series of meetings is usually required to brainstorm and document the desired behavior of a software product. It is best to start with a low level of precision and then increase the level of detail and precision as details become known.

In the early stages of one project, the company founders worked in the supply chain group at a large company. They were frustrated in their attempts to get materials and parts to the right location at the right time to support the manufacturing process. Based on their experience, knowledge, and difficulties, they wanted a software product to track shipments made between companies. They envisioned at least two user types: the shipping and the receiving clerks.

They were asked how the information about the shipments got into the system. “Shipments usually come from orders,” they said. They then realized that buyers and sellers were additional actors needing to interact with the product.

That led to the realization that the orders existed in other computer systems used by their prospective customers. What started out as a simple system to track shipments blossomed into a sophisticated product requiring integration with multiple enterprise systems.

In another situation, the company founders were experts at managing the construction of retail stores. Their computer skills were limited to occasional use of Microsoft Excel, Project, and Outlook for email. They had no experience with editing HTML to create a user interface. However, there were many tabular displays of data in the construction management product they envisioned, and they found Excel useful for drawing mockups of the screens.

Another effective technique is to bring together the subject matter experts and your offshore team for several direct communication sessions. This can occur in either direction by sending the expert(s) to the offshore location or bringing one or more key technical leaders from your offshore team to your facilities. Generally it is better to send one or two people offshore where they can interact with your entire offshore team.

The Requirements of Good Software Requirements

The waterfall approach will divide the overall process of product definition into the creation of two documents—a marketing requirements document (MRD) and a functional design specification (FDS). Some people combine the two into one big document that covers

marketing issues as well as a relatively complete description of the software functionality. Or they ignore marketing issues entirely and jump right into the details of software features.

Separating the process into two strands is useful. There are important marketing issues that should be addressed in the MRD. You don't want to distract yourself with the gory details of the software's functionality at this point.

Sometimes the MRD is the only definition of the software and is called a PRD, for product requirements document. In this case, more emphasis is placed on software functionality, and it is easy to overlook the marketing issues that may be critical in defining a new software product.

For example, one company was very focused on software features. The founding team designed a software system and wrote the PRD and FDS without considering potential partners in the marketplace. The system had a sophisticated approach to capture information that tracked shipments sent out by customers using the software.

Later, after most of the software was written, they realized that much of the functionality could have been implemented with an interface to the FedEx and UPS web sites. Such an interface would have had many more features to begin with. Furthermore, an alliance with one of these big companies would have been more important than any set of features the founders could have dreamed up.

In contrast, your MRD may be heavily weighted toward the size and texture of your market. Your opportunity may be the weaknesses of several competitors. The MRD can describe the market situation and indicate how your software will overcome these weaknesses and achieve a significant market share. It may go so far as to prioritize features in terms of their ability to gain market share.

Another recommended component of an MRD is a detailed comparison of competitors by product features in a matrix. You can use the matrix for customer presentations. Of course, your software column should contain the most checkmarks next to the feature descriptions. It is even better if you can have checkmarks in entire categories of features that your competitors have neglected to implement. Internally, this is a tremendous tool to motivate the developers and to describe what the product needs to do. "If we don't have this feature in the next release, we won't be any better than Brand X!"

How you write your MRD depends on what you consider your most important innovation. Is the software by itself so new and cool

that customers will beat a path to your door? If so, a feature-centered MRD or PRD may be sufficient.

Or will you need to include specific features that your competitors do not have? Is your innovation the use of a different sales channel that works only if your product is easily self-installed by the customer? If so, requirements like these must be included in your more marketing-centered MRD.

There are multiple formats of MRD to choose from. Two standards that cover requirements are as follows:

- IEEE Std 1220-1998—IEEE Standard for Application and Management of the Systems Engineering Process
- ANSI/IEEE Std-830-1984—IEEE/ANSI Guide to Software Requirements Specifications

These standards cover the specification and design of large systems and are heavily influenced by the defense industry. Standards like these are usually overkill for most companies and IT departments.

Here are the major sections I recommend be included in an MRD for most software companies and projects:

- **Introduction or executive summary.** Describes the purpose of the software or the problem it solves, the market opportunity, and a general plan for product releases and revenue. The introduction describes a vision of success achieved after the software is released.
- **Functional requirements.** Lists the major features and capabilities that must be included in the software. It can include a definition of user roles and high-level use cases.
- **Performance.** Describes the required speed of the software in terms of number of users, page views per second, and so on. Also includes the disk space and memory requirements or restrictions of the product.
- **Technology.** Describes requirements such as system architecture, databases, and computer languages. It can be combined with the “environment and integration” section in some MRDs.
- **Hardware.** Lists supported computers, network equipment, and so on.
- **Environment and integration.** Describes the environment or context in which the software will be used. Usually this

means specifying sources of data that the software must tap and other existing software packages that the product must integrate with.

Optional sections you may include in your MRD include the following:

- **Market opportunity.** Market size, segments, and competitor and partner information.
- **Schedule requirements.** Needed if your product must be released for an industry show, customer-specified date, or other date that defines your window of market opportunity.
- **Assumptions and dependencies.** Covers dependence on other companies' products or your own technical breakthroughs.
- **Glossary.** Defines the specific terms unique to your software category that may not be familiar to the development team, investors, and other stakeholders.

It is critical not to get hung up in an overly formal process when it comes to requirements and specifications. Requirements will change. They will evolve. Don't churn in your efforts to create an MRD. Set a time limit on the process. For many projects, the process of creating a useful MRD should take no longer than a week or two.

The MRD describes what the software *must* include and not how it is implemented. The "how" is described in the FDS.

Most of your writing time should be spent in creating an FDS after the MRD step is completed. The FDS contains more detailed uses cases, a sketch of your user interface, definitions of internal objects used to model information in your software, and many other details.

Is That Spec Done Yet, Tolstoy?

You can also gather requirements and write a specification as a way to explore and explain what your software needs to do. People have different emotions about specifications, however. Some consider them dreaded and dead-end programming artifacts that quickly become out of date and are never read anyway. Others are overly formal and treat the specification as if it is more important than the software itself.

A major myth of outsourcing is that you need a huge and detailed specification to do it effectively. The more detailed the specification,

the more of an insurance policy it becomes for getting exactly what you want. Therefore, the thinking goes, you can safely outsource only the creation of modules or the rewrite of an existing software application for which there is a complete specification.

It's just not true.

Your specification should be a guide to the development team, describing how to get started designing and developing your software. Involve the outsourced team in the specification process. They can ask probing questions and offer insights into your software design.

For example, one VP of engineering told me he starts his outsourced software development by sending the outsourced team a page or two of product requirements. He then carries out an online discussion during which the product details are fleshed out. The outsourced team asks questions, he answers (or gets the answers), and after a week or two there is enough information to begin coding.

It's like designing software using the Socratic method.

Most people start off with more of a specification than this. I recommend that you at least start by specifying the major use cases with low precision.

Designing a Completely New Software Application

We have covered several techniques for specifying your software and ways to capture your software's requirements. Unfortunately, such techniques are often not enough to ensure success when you are creating a completely new software application. Specifying every detail ahead of time is difficult, bordering on impossible.

It requires tremendous forethought to create a specification that covers the details of use cases, exceptions and error handling, performance requirements, and many other details that are often simply not known at the beginning of software development. And the specifications created are often ponderous documents that few will read before they become out of sync with the software under development.

Most software engineers are not dummies, and you don't need to specify every single detail. Doing so often takes the "fun" out of writing software anyway. Use the innate intelligence of your engineering team, either in-house or offshore, to your advantage. The ability to create new software applications is a critical criterion for hiring employees and it's the same for selecting an offshore outsourcing vendor.

Be More Agile

It is rare to see software accurately specified the first time. It always changes because you get new ideas as you see the software application start to take shape. Or your customers and users change their minds about what the software should do.

And yet many companies treat offshore outsourcing strictly as a business decision. They give a general description of what the software should do and then ask for the lowest possible price. If you are developing a new software application, this can be a recipe for disaster.

Or you can spend months debating what your new software application should do and what should be in the specification. However, you are better off selecting an offshore vendor with experience developing new applications using agile software development methods with whom you can collaborate.

Agile software development methods and user stories embrace change and are recommended for creating new software applications. Agile methods get your offshore vendor developing software as soon as possible. Frequent releases—every three to four weeks—gives you steadily increasing amounts of high-quality working software. A funny way to think of it is—Agile methods lets your programming team do something useful while you figure out what your software is supposed to do.

User Stories and the Planning Game

The techniques described earlier for creating use cases provide structure to the process of describing your software's behavior. This is a major part of capturing the requirements of your software.

Another approach called user stories is described well in the seminal book by Mike Cohn called *User Stories Applied for Agile Software Development*. This section gives you an overview of the ideas presented in greater detail in that book. The main idea is to integrate the software design and development process and delay getting into details until just before implementation.

The main goal of the software development process is to organize and optimize the collaboration between subject matter experts, the people who have the vision for a new software application, and the programmers who will bring the software to life. In extreme programming (XP),²⁹ this process is called *the planning game*. I think

29. See *Extreme Programming Explained: Embrace Change* by Kent Beck.

of it as the development game, since it covers both planning and development activities.

The planning game has four major activities:

- Exploration
- Release planning
- Iteration planning
- Task planning

Exploration is the process of creating stories. You write stories about how your customers will use your product. These stories are very similar to use cases. Then you estimate how long it will take to implement each story.

A concept of story “points” is used to achieve a consistent granularity of stories. For example, you might have a story that describes the login process for a web application. Logging in and looking up past activities or account information for display can be separate points in the story.

If a story is too complex, it should be split into multiple stories. The goal is to explore the functionality of the system by defining stories made up of story points that all take approximately the same amount of time to implement.

The number of story points that can be implemented in a given amount of time is called the *velocity*. Developers may need to spend a few days prototyping a story to estimate velocity. Sometimes work on similar projects may be used. Other elements, such as the number of screens, screen controls, and database tables involved with each story point, can also be used to estimate velocity.

Release planning involves determining the priority of the stories. These are business decisions. Users will be more interested in some stories than in others. You will usually want to implement the most important stories in the first or second release of the software.

During *iteration planning*, you choose an iteration size and decide which stories to implement. An iteration size is the amount of time (usually two to eight weeks in length) that the engineers will work to deliver some number of implemented stories. Both business and technical experts plan the iterations, using velocity to determine which stories to implement in each iteration. To improve accuracy, the actual velocity achieved in the last iteration is used for the next iteration.

Task planning is where developers divide the stories in an iteration into development tasks. The developers then sign up or are

assigned tasks, and make a commitment to complete their tasks by the end of the iteration.

If commitments cannot be met, this should be known halfway through the iteration. At that point adjustments are made with the business experts to determine which stories should be completed. It is more important to have completed stories at the end of an iteration than it is to have a maximum number of completed development tasks.

As iterations are completed, more and more user stories are implemented. Users can be shown these iteration results to make sure the product contains functionality they really want. As the product takes shape, new features and functions are discovered, by both business and technical resources. These new features may appear in stories implemented in future iterations.

For example, a new software application enabled the user to define a schedule for a construction project. After an early iteration was complete, a user could log in, select a project, and display the schedule. The schedule milestones and their order were fixed. A later iteration allowed the user to define new milestones and enter estimated and actual completion dates.

In fact, during this later iteration the subject matter experts realized (and then told the developers) that multiple dates per milestone were required. They described a new user story in which the construction manager could update the completion date of a milestone while preserving the original estimated date. No problem! It was easy to add this new functionality during an iteration when the focus was on just a few features for several new user stories. The developers were able to complete the iteration on time.

Another big user story involved updating the milestone dates by importing a schedule from Microsoft Project. This complex feature had a lower priority and was put off to a later release. The developers started with small chunks of functionality supporting complete user stories that they knew they could implement easily. As the developers learned more about the application, they were able to tackle larger tasks with the same level of confidence.

Outsourcing and Collaboration

Collaboration between business resources and technical resources, such as an outsourced software development team, is critical to success. You begin the process with a handful of important user stories.

Each iteration of development implements some of your stories and increases the accuracy of the velocity estimate and the predictability of the subsequent major release dates.

In the beginning of this chapter, I wrote that you are unlikely to find any guarantees in outsourcing your software development. Although you may not have a true guarantee, you can now see that it is possible to reduce the risk involved. You cannot guarantee that your original software idea (which may be inaccurate and incomplete) will be implemented in a fixed period of time. However, you can be assured that you will get a predictable amount of software in a predictable amount of time, and that the software will be likely to deliver a product that your customers will actually want to buy.

The point of a guarantee is not to get something for nothing, but to remove risk from the development process. Whether you develop your software with offshore outsourcing or your own development team, you can use these principles of agile software development to obtain your own reliable results.

Techniques for Estimating Your Schedule

Have you ever been surprised when your software takes longer to develop than you expected? Who hasn't?

If only there was a way to accurately estimate how long it will take to develop your software, and therefore how much it will cost.

Over the years, several techniques have been created to estimate the effort needed to complete software projects. Function Point Analysis, devised at IBM in the 1970s, defines function points in five groups: outputs, inquiries, inputs, files, and interfaces.

Each function point is defined as a business function, such as the user entering an input. This makes it easy to map function points into user-oriented requirements, like use cases. But it can also hide internal functions, which take time and resources to develop.

In 1981 another method, the Constructive Cost Model or COCOMO, was created by Barry Boehm³⁰. It was updated in 1997 and renamed COCOMO II. It uses a formula based on the number of lines of code expected to be in a software project to estimate the effort required to complete the development.

30. Software Cost Estimation with Cocomo II by Barry W. Boehm, et al, Prentice Hall PTR, 2000

I don't know about you, but I find it difficult to make the leap from a software idea or a page in a user interface to the number of lines of code needed to implement it.

The COCOMO formula also uses 5 values to estimate the scale of the software project, along with 17 cost drivers. But estimating the scale and cost of your software is also an art.

These methods are a bit formal and require research, skill, and a little guesswork. They are suited for large software projects in which you have a good grasp of the requirements and effort required to begin with.

For many software projects, the requirements are not as well defined. In these cases, other, less formal techniques are used to create rough estimates. For example, you can estimate the time needed to implement each of your user screens or pages. Screens may be complex or simple, and you can make three different time estimates for each screen based on its type.

You can also use the number of functions contained in your software or the number of database tables as other factors for making your estimate. Clearly, experience developing similar software will enhance your accuracy.

If you are using agile methods, you can use the planning game described earlier to calculate story points when you employ user stories.

An advantage of the planning game is that it helps you plan tasks, iterations and releases, and gains in accuracy as your software is developed over time.

Vendors use both formal and informal methods to estimate the effort needed to create your software. Estimating the time and effort required to develop your software is not a science, to say the least. If you are faced with a tight budget that will not accommodate delays and extra costs, you need a fixed-price bid, at least to get started.

A fixed-price bid needs to have a good specification of what you need. The downside of fixed price is the lack of flexibility. If you need to collaborate with an outsourced engineering team to design your software, a fixed price will not work.

Your alternative is to select a great outsourcing team with an excellent track record of carrying out software projects for other clients, and to work with them to create an estimate you can both believe in.

Using a fixed-price project is a good way to limit your financial risk when you are getting started with an outsourcing vendor.

However, it requires a good specification and should not be an excuse for judging the validity of their estimates. Work with your outsourcing team to develop and improve the accuracy of your estimates for both fixed-price and time and materials engagements.

Keep Your Eyes on the Prize

The ideas and techniques described in this chapter can get you on your way to using an outsourced software development team effectively. Your software will be well enough defined to start the development process, and your programmers will have a good sense of what you need. As the saying goes,

Success is getting what you want
And happiness is liking what you get.³¹

Your outsourcing can proceed well from this good start. Your next challenge is to keep things on track. That is covered in the next chapter, “Controlling Your Outsourced Software Development.”

31. Attributed to H. Jackson Brown, author of *Life's Little Instruction Book*.