CHAPTER 15

Risk reduction through prototyping

"Sharon, today I'd like to talk with you about the requirements that the buyers in the Purchasing Department have for the new Chemical Tracking System," began Lori, the business analyst. "Can you tell me what you want to be able to do with the system?"

"I'm not sure what to say," replied Sharon with a puzzled expression. "I can't describe what I need, but I'll know it when I see it."

The phrase IKIWISI—"I'll know it when I see it"—chills the blood of business analysts. It conjures an image of the development team having to make their best guess at the right software to build, only to have users tell them, "Nope, that's not right; try again." To be sure, envisioning a future software system and articulating its requirements is hard. People have difficulty describing their needs without having something tangible in front of them to contemplate; critiquing is much easier than conceiving.

Software prototyping takes a tentative step into the solution space. It makes the requirements more real, brings use cases to life, and closes gaps in your understanding of the requirements. Prototyping puts a mock-up or an initial slice of a new system in front of users to stimulate their thinking and catalyze the requirements dialog. Early feedback on prototypes helps stakeholders arrive at a shared understanding of the system's requirements, which reduces the risk of customer dissatisfaction.

Even if you apply the requirements development practices described in earlier chapters, portions of your requirements might still be uncertain or unclear to customers, developers, or both. If you don't correct these problems, an expectation gap between a user's vision of the product and a developer's understanding of what to build is guaranteed. Prototyping is a powerful way to introduce those all-important customer contact points that can reduce the expectation gap described in Chapter 2, "Requirements from the customer's perspective." It's hard to visualize exactly how software will behave by reading textual requirements or studying analysis models. Users are more willing to try out a prototype (which is fun) than to read an SRS (which is tedious). When you hear *IKIWISI* from your users, think about what you can provide that would help them articulate their needs or help you better understand what they have in mind (Boehm 2000). Prototypes are also a valuable tool for requirements validation. A business analyst can have users interact with prototypes to see if a product based on the prototype would truly meet their needs.

The word *prototype* has multiple meanings, and participants in a prototyping activity can hold very different expectations. A prototype airplane actually flies—it's the first instance of a new type of airplane. In contrast, a software prototype is only a portion or a model of a real system—it might not do anything useful at all. Software prototypes can be static designs or working models; quick sketches or highly detailed screens; visual displays or full slices of functionality; or simulations (Stevens et al. 1998; Constantine and Lockwood 1999).

This chapter describes how prototyping provides value to the project and different kinds of prototypes you might create for different purposes. It also offers guidance on how to use them during requirements development, as well as ways to make prototyping an effective part of your software engineering process.

Prototyping: What and why

A software *prototype* is a partial, possible, or preliminary implementation of a proposed new product. Prototypes can serve three major purposes, and that purpose must be made clear from the very beginning:

- **Clarify, complete, and validate requirements** Used as a requirements tool, the prototype assists in obtaining agreement, finding errors and omissions, and assessing the accuracy and quality of the requirements. User evaluation of the prototype points out problems with requirements and uncovers overlooked requirements, which you can correct at low cost before you construct the actual product. This is especially helpful for parts of the system that are not well understood or are particularly risky or complex.
- **Explore design alternatives** Used as a design tool, a prototype lets stakeholders explore different user interaction techniques, envision the final product, optimize system usability, and evaluate potential technical approaches. Prototypes can demonstrate requirements feasibility through working designs. They're useful for confirming the developer's understanding of the requirements before constructing the actual solution.
- **Create a subset that will grow into the ultimate product** Used as a construction tool, a prototype is a functional implementation of a subset of the product, which can be elaborated into the complete product through a sequence of small-scale development cycles. This is a safe approach only if the prototype is carefully designed with eventual release intended from the beginning.

The primary reason for creating a prototype is to resolve uncertainties early in the development process. You don't need to prototype the entire product. Focus on high-risk areas or known uncertainties to decide which parts of the system to prototype and what you hope to learn from the prototype evaluations. A prototype is useful for revealing and resolving ambiguity and incompleteness in the requirements. Users, managers, and other nontechnical stakeholders find that prototypes give them something concrete to contemplate while the product is being specified and designed. For each prototype you create, make sure you know—and communicate—why you're creating it, what you expect to learn from it, and what you'll do with the prototype after you've had people evaluate it.

Because of the risk of confusion, it's important to put some descriptors in front of the word "prototype" so the project participants understand why and when you might create one type of prototype or another. This chapter describes three classes of prototype attributes, each of which has two alternatives:

- **Scope** A mock-up prototype focuses on the user experience; a proof-of-concept prototype explores the technical soundness of a proposed approach.
- **Future use** A throwaway prototype is discarded after it has been used to generate feedback, whereas an evolutionary prototype grows into the final product through a series of iterations.
- **Form** A paper prototype is a simple sketch drawn on paper, a whiteboard, or in a drawing tool. An electronic prototype consists of working software for just part of the solution.

Each prototype you create will possess a specific combination of these attributes. For instance, you could devise a throwaway paper mock-up having simple drawings of possible screens. Or you might build an evolutionary electronic proof-of-concept, working software that demonstrates a desired technical capability that you can then grow into a deliverable product. Certain combinations don't make sense, though. For instance, you couldn't create an evolutionary paper proof of concept.

Mock-ups and proofs of concept

When people say "software prototype," they are usually thinking about a *mock-up* of a possible user interface. A mock-up is also called a *horizontal prototype*. Such a prototype focuses on a portion of the user interface; it doesn't dive into all the architectural layers or into detailed functionality. This type of prototype lets you explore some specific behaviors of the intended system, with the goal of refining the requirements. The mock-up helps users judge whether a system based on the prototype will let them do their job in a reasonable way.

A mock-up implies behavior without actually implementing it. It displays the facades of user interface screens and permits some navigation between them, but it contains little or no real functionality. Think of the set for a Western movie: the cowboy walks into the saloon and then walks out of the livery stable, yet he doesn't have a drink and he doesn't see a horse because there's nothing behind the false fronts of the buildings.

Mock-ups can demonstrate the functional options the user will have available, the look and feel of the user interface (colors, layout, graphics, controls), and the navigation structure. The navigations might work, but at certain points the user might see only a message that describes what would really be displayed or will find that some controls don't do anything. The information that appears in response to a database query could be faked or constant, and report contents are hardcoded. If you create a mock-up, try to use actual data in sample displays and outputs. This enhances the validity of the prototype as a model of the real system, but be sure to make it clear to the prototype evaluators that the displays and outputs are simulated, not live.

A mock-up doesn't perform any useful work, although it looks as if it should. The simulation is often good enough to let the users judge whether any functionality is missing, wrong, or unnecessary. Some prototypes represent the developer's concept of how a specific use case might be implemented. User evaluations of the prototype can point out alternative flows for the use case, missing interaction steps, additional exceptions, overlooked postconditions, and pertinent business rules.

When working with a throwaway mock-up prototype, the user should focus on broad requirements and workflow issues without becoming distracted by the precise appearance of screen elements (Constantine 1998). Don't worry at this stage about exactly where the screen elements will be positioned, fonts, colors, or graphics. The time to explore the specifics of user interface design is after you've clarified the requirements and determined the general structure of the interface. With an evolutionary mock-up, building in those refinements moves the user interface closer to being releasable.

A *proof of concept*, also known as a *vertical prototype*, implements a slice of application functionality from the user interface through all the technical services layers. A proof-of-concept prototype works like the real system is supposed to work because it touches on all levels of the system implementation. Develop a proof of concept when you're uncertain whether a proposed architectural approach is feasible and sound, or when you want to optimize algorithms, evaluate a proposed database schema, confirm the soundness of a cloud solution, or test critical timing requirements. To make the results meaningful, such prototypes are constructed by using production tools in a production-like operating environment. A proof of concept is also useful for gathering information to improve the team's ability to estimate the effort involved in implementing a specific user story or block of functionality. Agile development projects sometimes refer to a proof-of-concept prototype as a "spike."

I once worked with a team that wanted to implement an unusual client/server architecture as part of a transitional strategy from a mainframe-centric world to an application environment based on networked UNIX servers and workstations (Thompson and Wiegers 1995). A proof-of-concept prototype that implemented just a bit of the user interface client (on a mainframe) and the corresponding server functionality (on a UNIX workstation) allowed us to evaluate the communication components, performance, and reliability of our proposed architecture. The experiment was a success, as was the ultimate implementation based on that architecture.

Throwaway and evolutionary prototypes

Before constructing a prototype, make an explicit and well-communicated decision as to whether the prototype is exploratory only or will become part of the delivered product. Build a *throwaway prototype* to answer questions, resolve uncertainties, and improve requirements quality (Davis 1993). Because you'll discard the prototype after it has served its purpose, build it as quickly and cheaply as you can. The more effort you invest in the prototype, the more reluctant the project participants are to discard it and the less time you will have available to build the real product.

You don't have to throw the prototype away if you see merit in keeping it for possible future use. However, it won't be incorporated into the delivered product. For this reason, you might prefer to call it a *nonreleasable prototype*.

When developers build a throwaway prototype, they ignore solid software construction techniques. A throwaway prototype emphasizes quick implementation and modification over robustness, reliability, performance, and long-term maintainability. For this reason, you must not allow low-quality code from a throwaway prototype to migrate into a production system. If you do, the users and the maintainers will suffer the consequences for the life of the product.

A throwaway prototype is most appropriate when the team faces uncertainty, ambiguity, incompleteness, or vagueness in the requirements, or when they have difficulty envisioning the system from the requirements alone. Resolving these issues reduces the risks of proceeding with construction. A prototype that helps users and developers visualize how the requirements might be implemented can reveal gaps in the requirements. It also lets users judge whether the requirements will enable the necessary business processes.

> **Trap** Don't make a throwaway prototype more elaborate than is necessary to meet the prototyping objectives. Resist the temptation—or the pressure from users—to keep adding more capabilities to the prototype.

A *wireframe* is a particular approach to throwaway prototyping commonly used for custom user interface design and website design. You can use wireframes to reach a better understanding of three aspects of a website:

- The conceptual requirements
- The information architecture or navigation design
- The high-resolution, detailed design of the pages

The pages sketched when exploring conceptual requirements in the first type of wireframe need not resemble the final screens. This wireframe is useful for working with users to understand the types of activities they might want to perform at the screen. Paper prototypes can work fine for this purpose, as described later in this chapter. The second type of wireframe need not involve page designs at all. The analysis model called the dialog map, described in Chapter 12, "A picture is worth 1024 words," is an excellent tool for exploring and iterating on page navigation for a website. The third type of wireframe gets into the details of what the final pages would look like.

In contrast to a throwaway prototype, an *evolutionary prototype* provides a solid architectural foundation for building the product incrementally as the requirements become clear over time (McConnell 1996). Agile development provides an example of evolutionary prototyping. Agile teams construct the product through a series of iterations, using feedback on the early iterations to adjust the direction of future development cycles. This is the essence of evolutionary prototyping.

In contrast to the quick-and-dirty nature of throwaway prototyping, an evolutionary prototype must be built with robust, production-quality code from the outset. Therefore, an evolutionary prototype takes longer to create than a throwaway prototype that simulates the same system capabilities. An evolutionary prototype must be designed for easy growth and frequent enhancement, so developers must emphasize software architecture and solid design principles. There's no room for shortcuts in the quality of an evolutionary prototype.

Think of the first iteration of an evolutionary prototype as a pilot release that implements an initial portion of the requirements. Lessons learned from user acceptance testing and initial usage lead to modifications in the next iteration. The full product is the culmination of a series of evolutionary prototyping cycles. Such prototypes quickly get useful functionality into the hands of the users. Evolutionary prototypes work well for applications that you know will grow over time, but that can be valuable to users without having all the planned functionality implemented. Agile projects often are planned such that they could stop development at the end of an iteration and still have a product that is useful for customers, even though it is incomplete.

Evolutionary prototyping is well suited for web development projects. On one such project, my team created a series of four prototypes, based on requirements that we developed from a use case analysis. Several users evaluated each prototype, and we revised each one based on their responses to questions we posed. The revisions following the fourth prototype evaluation resulted in the production website.

Figure 15-1 illustrates several possible ways to combine the various prototypes. For example, you can use the knowledge gained from a series of throwaway prototypes to refine the requirements, which you might then implement incrementally through an evolutionary prototyping sequence. An alternative path through Figure 15-1 uses a throwaway mock-up to clarify the requirements prior to finalizing the user interface design, while a concurrent proof-of-concept prototyping effort validates the architecture and core algorithms. What you *cannot* do successfully is turn the deliberately low quality of a throwaway prototype into the maintainable robustness that a production system demands. In addition, working prototypes that appear to get the job done for a handful of concurrent users likely won't scale up to handle thousands of users without major architectural changes. Table 15-1 summarizes some typical applications of throwaway, evolutionary, mock-up, and proof-of-concept prototypes.

TABLE 15-1 Typical applications of software prototypes

	Throwaway	Evolutionary
Mock-up	■ Clarify and refine user and functional requirements. ■ Identify missing functionality. ■ Explore user interface approaches.	■ Implement core user requirements. ■ Implement additional user requirements based on priority. ■ Implement and refine websites. ■ Adapt system to rapidly changing business needs.
Proof of concept	■ Demonstrate technical feasibility. ■ Evaluate performance. ■ Acquire knowledge to improve estimates for construction.	■ Implement and grow core multi-tier functionality and communication layers. ■ Implement and optimize core algorithms. ■ Test and tune performance.

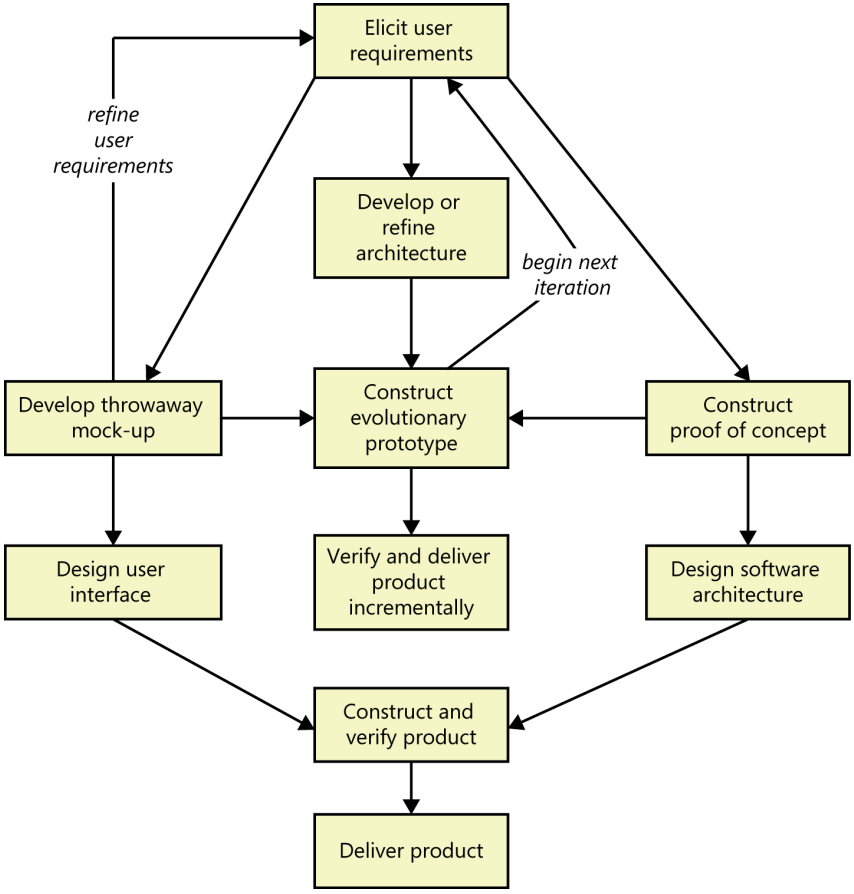

FIGURE 15-1 Several possible ways to incorporate prototyping into the software development process.

Paper and electronic prototypes

You don't always need an executable prototype to resolve requirements uncertainties. A *paper prototype* (sometimes called a *low-fidelity prototype*) is a cheap, fast, and low-tech way to explore how a portion of an implemented system might look (Rettig 1994). Paper prototypes help you test whether users and developers hold a shared understanding of the requirements. They let you take a tentative and low-risk step into a possible solution space prior to developing production code. A similar deliverable is called a *storyboard* (Leffingwell and Widrig 2000). Use low-fidelity prototypes to explore functionality and flow, and use high-fidelity prototypes to determine precise look and feel.

Paper prototypes involve tools no more sophisticated than paper, index cards, sticky notes, and whiteboards. The designer sketches ideas of possible screens without worrying about exactly where the controls appear and what they look like. Users willingly provide feedback on designs drawn on a piece of paper, although they're sometimes less eager to critique a lovely computer-based prototype

in which it appears the developer has invested a lot of work. Developers, too, might resist making substantial changes in a carefully crafted electronic prototype.

When a low-fidelity prototype is being evaluated, someone plays the role of the computer while a user walks through an evaluation scenario. The user initiates actions by saying aloud what she would like to do at a specific screen: "I'm going to select Print Preview from the File menu." The person simulating the computer then displays the piece of paper or index card that represents the display that would appear when the user takes that action. The user can judge whether that is indeed the expected response and whether the item displayed contains the correct elements. If it's wrong, you simply take a blank page or index card and try again.

Off to see the wizard

A development team that designed large commercial photocopiers once lamented to me that their previous copier had a usability problem. A common copying activity required five discrete steps, which the users found clumsy. "I wish we'd prototyped that activity before we designed the copier," one developer said wistfully.

How do you prototype a product as complex as a photocopier? First, buy a refrigerator. Write *COPIER* on the side of the box that it came in. Have someone sit inside the box, and ask a user to stand outside the box and simulate doing copier activities. The person inside the box responds in the way he expects the copier to respond, and the user representative observes whether that response is what he has in mind. A simple, fun prototype like this—sometimes called a *Wizard of Oz prototype*—stimulates the early user feedback that effectively guides the development team's design decisions. Plus, you get to keep the refrigerator.

No matter how efficient your prototyping tools are, sketching displays on paper or a whiteboard is faster. Paper prototyping facilitates rapid iteration, and iteration is a key success factor in requirements development. Paper prototyping is an excellent technique for refining the requirements prior to designing detailed user interfaces, constructing an evolutionary prototype, or undertaking traditional design and construction activities. It also helps the development team manage customer expectations.

Numerous tools are available if you decide to build an electronic throwaway prototype. They range from simple drawing tools such as Microsoft Visio and Microsoft PowerPoint to commercial prototyping tools and graphical user interface builders. Tools also are available specifically for creating website wireframes. Such tools will let you easily implement and modify user interface components, regardless of how inefficient the temporary code behind the interface is. Of course, if you're building an evolutionary prototype, you must use production development tools from the outset. Because tools and their vendors change so rapidly, we won't suggest specific ones here.

Various tools are commercially available that let you simulate your application before you build it. Application simulation lets you quickly assemble screen layouts, user interface controls, navigation flow, and functionality into something that closely resembles the product you think you need to build. The ability to iterate on the simulation provides a valuable mechanism for interacting with user representatives to clarify requirements and revise your thinking about the solution.

With any kind of prototyping—paper prototypes, wireframes, electronic prototypes, or simulations—the business analyst must be careful not get drawn into high-precision user interface designs prematurely. Prototype evaluators often offer feedback like "Can this text be a little darker red?", "Let's move this box up just a little," or "I don't like that font." Unless the purpose of the prototype is to perform detailed screen or webpage design, those sorts of comments are just distractions. The color, font, and box positioning are immaterial if the application doesn't properly support the users' business tasks. Until you're sure you have a rich understanding of the necessary functionality, focus the prototyping efforts on refining requirements, not visual designs.

Working with prototypes

Figure 15-2 shows one possible sequence of development activities that moves from use cases to detailed user interface design with the help of a throwaway prototype. Each use case description includes a sequence of actor actions and system responses, which you can model by using a dialog map to depict a possible user interface architecture. A throwaway prototype or a wireframe elaborates the dialog elements into specific screens, menus, and dialog boxes. When users evaluate the prototype, their feedback might lead to changes in the use case descriptions (if, say, an alternative flow is discovered) or to changes in the dialog map. After the requirements are refined and the screens sketched, each user interface element can be optimized for usability. These activities don't need to be performed strictly sequentially. Iterating on the use case, the dialog map, and the wireframe is the best way to quickly reach an acceptable and agreed-upon approach to user interface design.

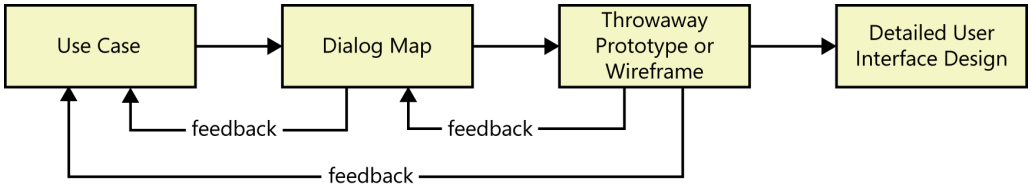

FIGURE 15-2 Activity sequence from use cases to user interface design using a throwaway prototype.

This progressive refinement approach is cheaper than leaping directly from use case descriptions to a complete user interface implementation and then discovering major issues that necessitate extensive rework. You only need to perform as many steps in this sequence as are necessary to acceptably reduce the risk of going wrong on the user interface design. If your team is confident that they understand the requirements, that the requirements are sufficiently complete, and that they have a good handle on the right UI to build, then there's little point in prototyping. Also, you can focus prototyping on user requirements that have a big risk of error or a big impact if there is a problem. One project performed an e-commerce website redesign for a major corporation that would be used by millions of users. The team prototyped the core elements of the website, including the online catalog, shopping cart, and checkout process, to make sure they got those right the first time. They spent less time exploring exception paths and less commonly used scenarios.

To help make this whole process more tangible, let's look at an actual example, a small website to promote a book, a memoir of life lessons called *Pearls from Sand*. The author of the book (Karl, actually) thought of several things that visitors should be able to do at the website, each of which is a use case. There are additional use cases for other user classes (Table 15-2).

TABLE 15-2 Some use cases for PearlsFromSand.com

User class	Use case
Visitor	Get Information about the Book Get Information about the Author Read Sample Chapters Read the Blog Contact the Author
Customer	Order a Product Download an Electronic Product Request Assistance with a Problem
Administrator	Manage the Product List Issue a Refund to a Customer Manage the Email List

The next step was to think of the pages the website should provide and imagine the navigation pathways between them. The final website might not implement all of these pages separately. Some pages might be condensed together; others might function as pop-ups or other modifications of a single page. Figure 15-3 illustrates a portion of a dialog map that illustrates a conceptual page architecture. Each box represents a page that would contribute to providing the services identified in the use cases. The arrows represent links to enable navigation from one page to another. While drawing a dialog map, you might discover new actions a user would want to perform. While working through a use case, you might find ways to simplify and streamline the user's experience.

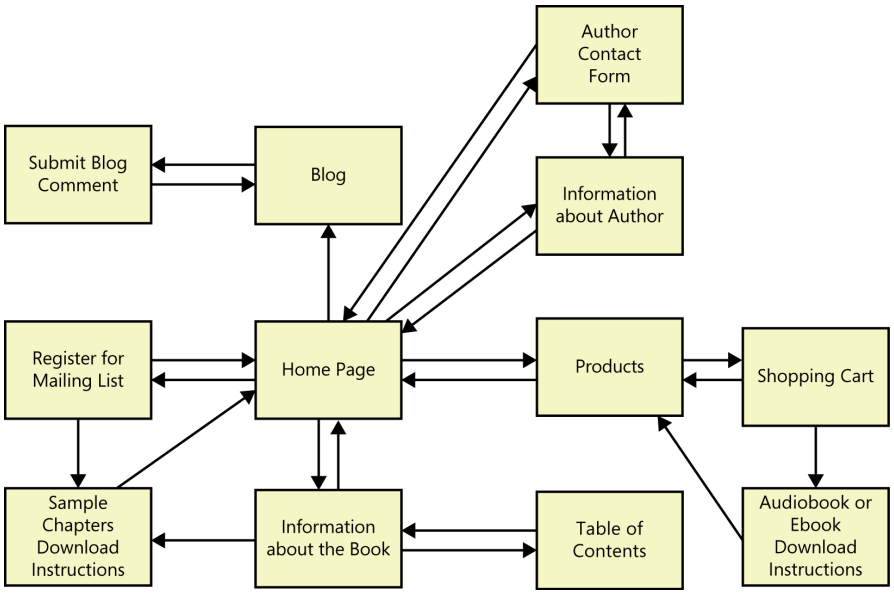

FIGURE 15-3 Partial dialog map for PearlsFromSand.com.

The next step was to construct a throwaway prototype or a wireframe of selected pages to work out the visual design approach. Each of these can be a hand-drawn sketch on paper (see the example in Figure 10-1 in Chapter 10, "Documenting the requirements"), a simple line drawing, or a mock-up created with a dedicated prototyping or visual design tool. The wireframe illustrated in Figure 15-4 was drawn by using PowerPoint in just a few minutes. Such a simple diagram is a tool to work with user representatives to understand the broad strokes of what sort of page layout and cosmetic features would make the pages easy to understand and use.

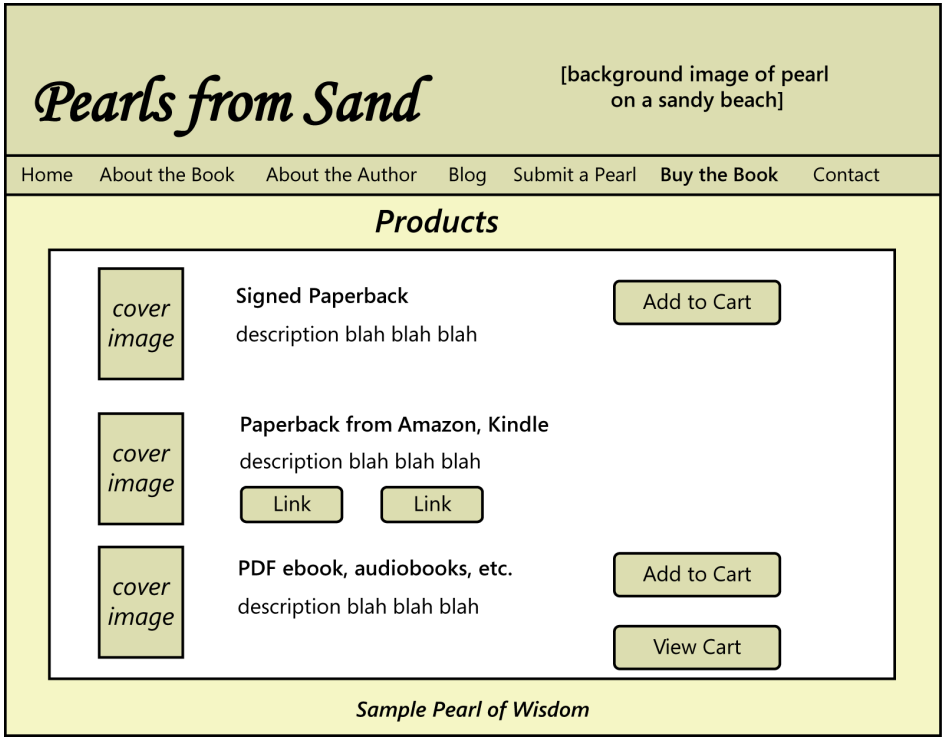

FIGURE 15-4 Sample wireframe of one page for PearlsFromSand.com.

Finally, the fourth step illustrated in Figure 15-2 is to create a detailed user interface screen design. Figure 15-5 shows one final page from the PearlsFromSand.com website, the culmination of the requirements analysis and prototyping activities that came before. This iterative approach to user interface design leads to better results than diving immediately into high-resolution page design without having a clear understanding of what members of various user classes will want to do when they visit a website.

FIGURE 15-5 A final implemented page from PearlsFromSand.com.

Prototype evaluation

Prototype evaluation is related to usability testing (Rubin and Chisnell 2008). You'll learn more by watching users work with the prototype than just by asking them to tell you what they think of it. Watch where the user's fingers or mouse pointer try to go instinctively. Spot places where the prototype conflicts with the behavior of other applications that the evaluators use. The evaluator might try incorrect keyboard shortcuts or have to "mouse around" hunting for the correct menu option. Look for the furrowed brow that indicates a puzzled user who can't determine what to do next, how to navigate to a desired destination, or how to take a side trip to another part of the application. See if the prototype has any dead ends, as happens sometimes when a user submits a form on a website.

Have the right people evaluate the prototype from the appropriate perspectives. Include members of multiple user classes, both experienced and inexperienced. When you present the prototype to the evaluators, stress that it addresses only a portion of the functionality; the rest will be implemented when the actual system is developed.

Trap As with any usability testing, watch out for omitting members of significant user classes from the prototype evaluation. A novice user might love a prototype for its apparent ease of use, but a more experienced or power user could hate the way it slows him down. Make sure both groups are represented.

To improve the evaluation of user interface prototypes, create scripts that guide the users through a series of operations and ask specific questions to elicit the information you seek. This supplements a general invitation to "tell me what you think of this prototype." Derive the evaluation scripts from the use cases, user stories, or features that the prototype addresses. The script asks evaluators to perform specific tasks, working through the parts of the prototype that have the most uncertainty. At the end of each task, and possibly at intermediate points, the script presents specific task-related questions. You might also ask general questions like the following:

- Does the prototype implement the functionality in the way you expected?
- What functionality is missing from the prototype?
- Can you think of any possible error conditions that the prototype doesn't address?
- Are any unnecessary functions present?
- How logical and complete does the navigation seem to you?
- Are there ways to simplify any of the tasks that require too many interaction steps?
- Were you ever unsure of what to do next?

Ask evaluators to share their thoughts aloud as they work with the prototype so that you understand what they're thinking and can detect any issues that the prototype handles poorly. Create a nonjudgmental environment in which the evaluators feel free to express their thoughts, ideas, and concerns. Avoid coaching users on the "right" way to perform some function with the prototype.

Document what you learn from the prototype evaluation. Use the information from a mock-up prototype to refine the requirements. If the evaluation led to some user-interface design decisions, such as the selection of specific interaction techniques, record those conclusions and how you arrived at them. Decisions that lack the accompanying thought processes tend to be revisited repeatedly. For a proof of concept, document the evaluations you performed and their results, culminating in the decisions you made about the technical approaches explored. Resolve any conflicts between the specified requirements and the prototype.

Risks of prototyping

Creating even a simple prototype costs time and money. Although prototyping reduces the risk of software project failure, it poses its own risks, some of which are explained in this section.

Pressure to release the prototype

The biggest risk is that a stakeholder will see a running throwaway prototype and conclude that the product is nearly completed. "Wow, it looks like you're almost done!" says the enthusiastic prototype evaluator. "This looks great. Can you just finish this up and give it to me?"

In a word: NO! A throwaway prototype is never intended for production use, no matter how much it looks like the real thing. It is merely a model, a simulation, an experiment. Unless there's a compelling business motivation to achieve a marketplace presence immediately (and management accepts the resulting high maintenance burden and risk of annoyed users), resist the pressure to deliver a throwaway prototype. Delivering this prototype will likely delay the project's completion because the design and code were intentionally created without regard to quality or durability. Expectation management is a key to successful prototyping. Everyone who sees the prototype must understand its purpose and its limitations. Be clear about why you are creating specific kinds of prototypes, decide what their ultimate fate will be, and communicate this clearly to those stakeholders who are involved with them.

Don't let the fear of premature delivery pressure dissuade you from creating prototypes, though. Make it clear to those who see the prototype that you will not release it as production software. One way to control this risk is to use paper, rather than electronic, prototypes. No one who evaluates a paper prototype will think the product is nearly done! Another option is to use prototyping tools that are different from those used for actual development. No one will mistake a navigable PowerPoint mock-up or a simple wireframe for the real thing. This will help you resist pressure to "just finish up" the prototype and ship it. Leaving the prototype looking a bit rough and unpolished also mitigates this risk. Some of the many tools available for creating wireframes allow for the quick development of a high-fidelity user interface. This increases the likelihood of people expecting that the software is almost done, and it adds to the pressure to transform a throwaway prototype into an evolutionary one.

One developer cobbled together an executable prototype of a user interface with a shocking pink motif. As he explained it, "When we showed the customers the first couple of iterations with this color scheme, NO ONE thought this was a close-to-finished product. I actually retained that abomination for an additional iteration just to avoid falling into some of these prototyping risk traps."

Distraction by details

Another risk of prototyping is that users become fixated on details about how the user interface will look and operate. When working with real-looking prototypes, it's easy for users to forget that they should be primarily concerned with conceptual issues at the requirements stage. Limit the prototype to the displays, functions, and navigation options that will let you clear up uncertain requirements.

Baby with the bath water

I once consulted at a company where a senior manager had banned prototyping. He had seen projects in which customers pressured developers into delivering throwaway prototypes prematurely as the final product, with predictable results. The prototypes did not handle user errors or bad input data well, did not cover all the options users wanted, and were difficult to maintain and enhance. These unpleasant experiences led the senior manager to conclude that prototyping could only lead to trouble.

As you've seen in this chapter, delivering to customers a prototype that was intended to be discarded and calling it a product certainly will cause problems. Nonetheless, prototyping offers a range of powerful techniques that can contribute substantially to building the right product. Rather than dismissing prototyping as a dangerous method to be avoided, it's important to make sure everyone involved understands the various kinds of prototypes, why a particular prototype is being created, and how the results will be used.

Unrealistic performance expectations

A third risk is that users will infer the expected performance of the final product from the prototype's performance. You won't be evaluating a mock-up in the intended production environment, though. You might have built it using tools or languages that differ in efficiency from the production development tools, such as interpreted scripts versus compiled code. A proof-of-concept prototype might not use tuned algorithms, or it could lack security layers that will reduce the ultimate performance. If evaluators see the prototype respond instantaneously to a simulated database query using hard-coded sample query results, they might expect the same fabulous performance in the production software with an enormous distributed database. Consider building in time delays to more realistically simulate the expected behavior of the final product—and perhaps to make the prototype look even less ready for immediate delivery. You might put a message on the screen to clearly state that this is not necessarily representative of the final system.

In agile development and other evolutionary prototyping situations, be sure to design a robust and extendable architecture and craft high-quality code from the beginning. You're building production software, just a small portion at a time. You can tune up the design through refactoring in later iterations, but don't substitute refactoring in the future for thinking about design today.

Investing excessive effort in prototypes

Finally, beware of prototyping activities that consume so much effort that the development team runs out of time and is forced to deliver the prototype as the product or to rush through a haphazard product implementation. This can happen when you are prototyping the whole solution rather than only the most uncertain, high-risk, or complex portions. Treat a prototype as an experiment. You're testing the hypothesis that the requirements are sufficiently defined and the key human-computer

interface and architectural issues are resolved so that design and construction can proceed. Do just enough prototyping to test the hypothesis, answer the questions, and refine the requirements.

Prototyping success factors

Software prototyping provides a powerful set of techniques that can minimize development schedules, ensure customer satisfaction, and produce high-quality products. To make prototyping an effective part of your requirements process, follow these guidelines:

- Include prototyping tasks in your project plan. Schedule time and resources to develop, evaluate, and modify the prototypes.
- State the purpose of each prototype before you build it, and explain what will happen with the outcome: either discard (or archive) the prototype, retaining the knowledge it provided, or build upon it to grow it into the ultimate solution. Make sure those who build the prototypes and those who evaluate them understand these intentions.
- Plan to develop multiple prototypes. You'll rarely get them right on the first try, which is the whole point of prototyping!
- Create throwaway prototypes as quickly and cheaply as possible. Invest the minimum amount of effort that will answer questions or resolve requirements uncertainties. Don't try to perfect a throwaway prototype.
- Don't include input data validations, defensive coding techniques, error-handling code, or extensive code documentation in a throwaway prototype. It's an unnecessary investment of effort that you're just going to discard.
- Don't prototype requirements that you already understand, except to explore design alternatives.
- Use plausible data in prototype screen displays and reports. Evaluators can be distracted by unrealistic data and fail to focus on the prototype as a model of how the real system might look and behave.
- Don't expect a prototype to replace written requirements. A lot of behind-the-scenes functionality is only implied by the prototype and should be documented in an SRS to make it complete, specific, and traceable. Screen images don't give the details of data field definitions and validation criteria, relationships between fields (such as UI controls that appear only if the user makes certain selections in other controls), exception handling, business rules, and other essential bits of information.

Thoughtfully applied and skillfully executed, prototypes serve as a valuable tool to help with requirements elicitation, requirements validation, and that tricky translation from needs into solutions.

Next steps

- Identify a portion of your project that exhibits confusion about requirements or is a high-risk area of functionality. Sketch out a portion of a possible user interface that represents your understanding of the requirements and how they might be implemented—a paper prototype. Have some users walk through your prototype to simulate performing a usage scenario. Identify places where the initial requirements were incomplete or incorrect. Modify the prototype accordingly and walk through it again to confirm that the shortcomings are corrected.
- Summarize this chapter for your prototype evaluators to help them understand the rationale behind the prototyping activities and to help them have realistic expectations for the outcome.
- If your product is a hardware device, think of a way you can physically simulate it so users can interact with it to validate and flesh out their requirements.