rapidly changing business requirements more closely. Once you have learned the concepts of prototyping, it is much easier to grasp the essentials of RAD, which can be thought of as a specific implementation of prototyping.

Some developers are looking at RAD as a helpful approach in new ecommerce, Web-based environments in which so-called first-mover status of a business might be important. In other words, to deliver an application to the Web before their competitors, businesses may want their development team to experiment with RAD.

Phases of RAD

There are three broad phases to RAD that engage both users and analysts in assessment, design, and implementation. Figure 6.4 depicts these three phases. Notice that RAD involves users in each part of the development effort, with intense participation in the business part of the design.

REQUIREMENTS PLANNING PHASE. In the requirements planning phase, users and analysts meet to identify objectives of the application or system and to identify information requirements arising from those objectives. This phase requires intense involvement from both groups; it is not just signing off on a proposal or document. In addition, it may involve users from different levels of the organization (as covered in Chapter 2). In the requirements planning phase, when information requirements are still being addressed, you may be working with the CIO (if it is a large organization) as well as with strategic planners, especially if you are working with an ecommerce application that is meant to further the strategic aims of the organization. The orientation in this phase is toward solving business problems. Although information technology and systems may even drive some of the solutions proposed, the focus will always remain on reaching business goals.

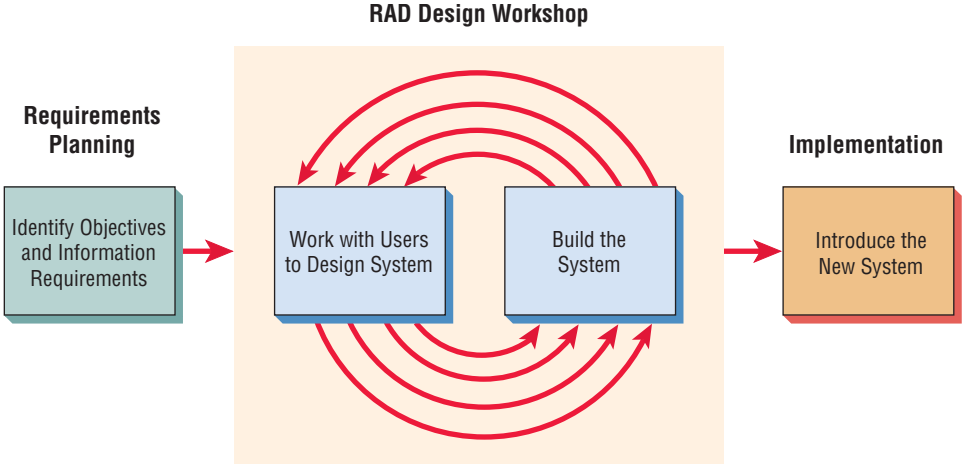
RAD DESIGN WORKSHOP. The RAD design workshop phase is a design-and-refine phase that can best be characterized as a workshop. When you imagine a workshop, you know that participation is intense, not passive, and that it is typically hands on. Usually participants are seated at round tables or in a U-shaped configuration of chairs with attached desks where each person can see the other and where there is space to work on a notebook computer. If you are fortunate enough to have a group decision support systems (GDSS) room available at the company or through a local university, use it to conduct at least part of your RAD design workshop.

During the RAD design workshop, users respond to actual working prototypes and analysts refine designed modules (using some of the software tools mentioned later) based on user responses. The workshop format is very exciting and stimulating, and if experienced users and analysts are present, there is no question that this creative endeavor can propel development forward at an accelerated rate.

IMPLEMENTATION PHASE. In the previous figure, you can see that analysts are working with users intensely during the workshop to design the business or nontechnical aspects of the system. As soon as these aspects are agreed on and the systems are built and refined, the new systems or part of systems are tested and then introduced to the organization. Because RAD can be used to create

FIGURE 6.4

The RAD design workshop is the heart of the interactive development process.



new ecommerce applications for which there is no old system, there is often no need to (and no real way to) run the old and new systems in parallel before implementation.

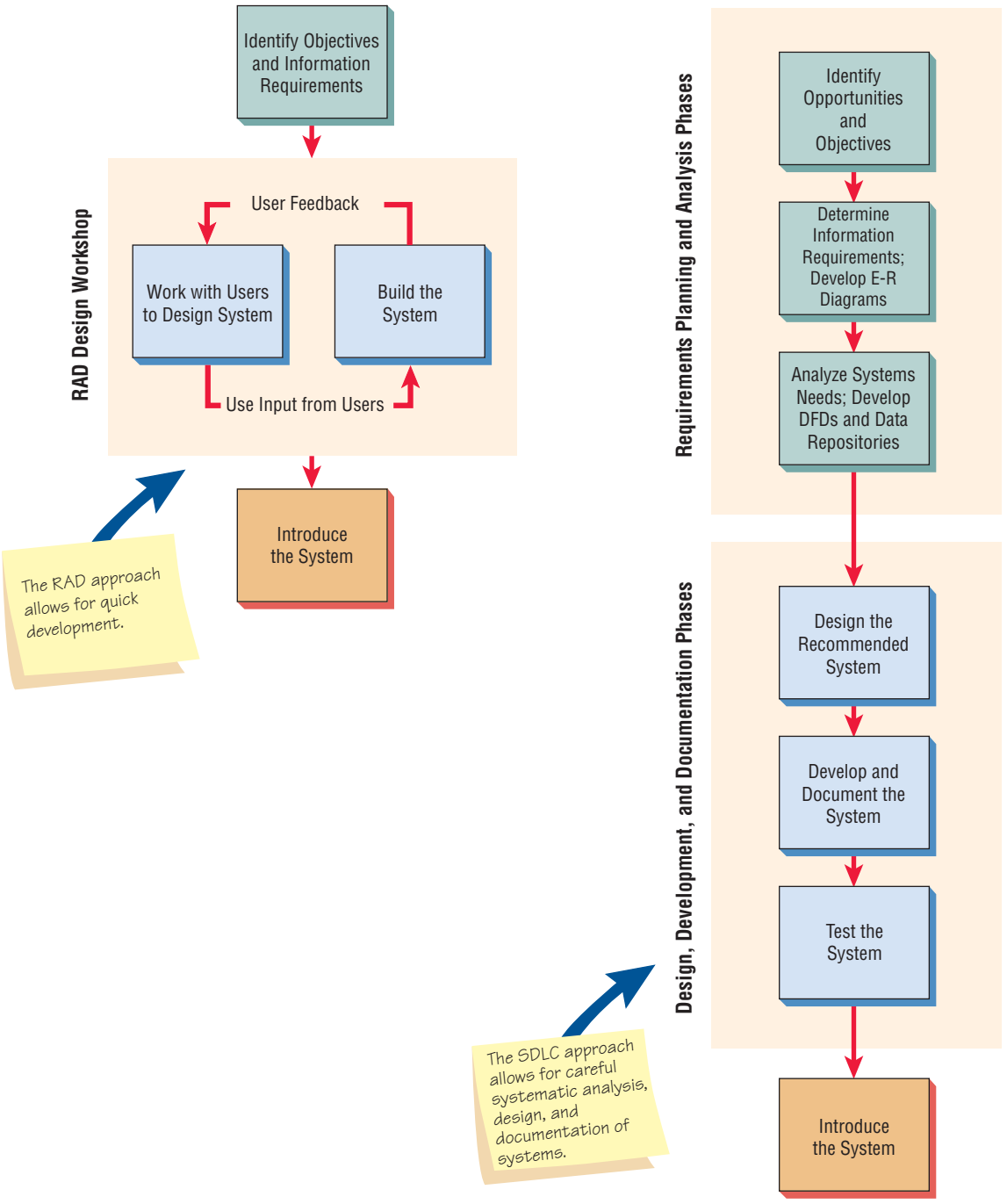
By this time, the RAD design workshop will have generated excitement, user ownership, and acceptance of the new application. Typically, change brought about in this manner is far less wrenching than when a system is delivered with little or no user participation.

Comparing RAD to the SDLC

In Figure 6.5 you can compare the phases of the SDLC with those detailed for RAD at the beginning of this section. Notice that the ultimate purpose of RAD is to shorten the SDLC and in this way respond more rapidly to dynamic information requirements of organizations. The SDLC takes a more methodical, systematic approach that ensures completeness and accuracy and has as its intention the creation of systems that are well integrated into standard business procedures and culture.

FIGURE 6.5

The RAD design workshop and the SDLC approach compared.



The RAD design workshop phase is a departure from the standard SDLC design phases, because RAD software tools are used to generate screens and to exhibit the overall flow of the running of the application. Thus, when users approve this design, they are signing off on a visual model representation, not just a conceptual design represented on paper, as is traditionally the case.

The implementation phase of RAD is in many ways less stressful than others, because the users have helped to design the business aspects of the system and are well aware of what changes will take place. There are few surprises, and the change is something that is welcomed. Often when using the SDLC, there is a lengthy time during development and design when analysts are separated from users. During this period, requirements can change and users can be caught off guard if the final product is different than anticipated over many months.

WHEN TO USE RAD. As an analyst, you want to learn as many approaches and tools as possible to facilitate getting your work done in the most appropriate way. Certain applications and systems work will call forth certain methodologies. Consider using RAD when:

1. Your team includes programmers and analysts who are experienced with it; and
2. There are pressing business reasons for speeding up a portion of an application development; or
3. When you are working with a novel ecommerce application and your development team believes that the business can sufficiently benefit over their competitors from being an innovator if this application is among the first to appear on the Web; or
4. When users are sophisticated and highly engaged with the organizational goals of the company.

DISADVANTAGES OF RAD. The difficulties with RAD, as with other types of prototyping, arise because systems analysts try to hurry the project too much. Suppose two carpenters are hired to build two storage sheds for two neighbors. The first carpenter follows the SDLC philosophy, whereas the second follows the RAD philosophy.

The first carpenter is systematic, inventorying every tool, lawn mower, and piece of patio furniture to determine the correct size for the shed, designing a blueprint of the shed, and writing specifications for every piece of lumber and hardware. The carpenter builds the shed with little waste and has precise documentation about how the shed was built if anyone wants to build another just like it, repair it, or paint it using the same color.

The second carpenter jumps right into the project by estimating the size of the shed, getting a truckload of lumber and hardware, building a frame and discussing it with the owner of the property as modifications are made when certain materials are not available, and making a trip to return the lumber not used. The shed gets built faster, but if a blueprint is not drawn, the documentation never exists.

AGILE MODELING

Agile methods are a collection of innovative, user-centered approaches to systems development. You will learn the values and principles, activities, resources, practices, processes, and tools associated with agile methodologies in the upcoming section. Agile methods can be credited with many successful systems development projects and in numerous cases even credited with rescuing companies from a failing system that was designed using a structured methodology.

Values and Principles of Agile Modeling

The agile approach is not based just on results. It is based on values, principles, and practices. Essential to agile programming are stated values and principles that create the context for collaboration among programmers and customers. In order to be agile analysts, you must adhere to the following values and principles as developed by Beck (2000) in his work on agile modeling that he called "extreme programming" or "XP."

FOUR VALUES OF AGILE MODELING. There are four values that create an environment in which both developers and businesses can be adequately served. Because there is often tension between what developers do in the short term and what is commercially desirable in the long term, it is important that you knowingly espouse values that will form a basis for acting together on a software project. The four values are communication, simplicity, feedback, and courage, as shown in Figure 6.6.

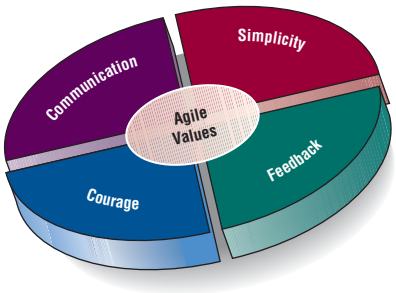


FIGURE 6.6
Values are crucial to the agile approach.

Let's begin with communication. Every human endeavor is fraught with possibilities for miscommunication. Systems projects that require constant updating and technical design are especially prone to such errors. Add to this tight project deadlines, specialized jargon, and the stereotype that programmers would prefer to talk to machines rather than people, and you have the potential for some serious communication problems. Projects can be delayed; the wrong problem can be solved; programmers are punished for even bringing up problems to managers; people leave or join the project in midstream without proper updates; and so the litany goes.

Typical agile practices such as pair programming (two programmers collaborating, described later in the chapter), estimating tasks, and unit testing rely heavily on good communication. Problems are fixed rapidly, holes are closed, and weak thinking is quickly strengthened through interaction with others on the team.

A second value of the agile approach is that of simplicity. When we are working on a software development project, our first inclination is to become overwhelmed with the complexity and bigness of the task. However, you cannot run until you know how to walk, nor walk until you know how to stand. Simplicity for software development means that we will begin with the simplest possible thing we can do.

The agile value of simplicity asks us to do the simplest thing today, with the understanding that it might have to be changed a little tomorrow. This requires a clear focus on the goals of the project and really is a basic value.

Feedback is the third basic value that is important when taking an extreme programming approach. When you think of feedback in this context, it is good to consider that feedback is wrapped up with the concept of time. Good, concrete feedback that is useful to the programmer, analyst, and customer can occur within seconds, minutes, days, weeks, or months, depending on what is needed, who is communicating, and what will be done with the feedback. A fellow programmer may hand you a test case that breaks the code you wrote only hours before, but that feedback is almost priceless in terms of being able to change what is not working before it is accepted and further embedded in the system.

Feedback occurs when customers create functional tests for all of the stories that programmers have subsequently implemented. (See more on user stories later in this chapter.) Critical feedback about the schedule comes from customers who compare the goal of the plan to the progress that has been made. Feedback helps programmers to make adjustments and lets the business start experiencing very early on what the new system will be like once it is fully functional.

Courage is the fourth value enunciated in agile programming. The value of courage has to do with a level of trust and comfort that must exist in the development team. It means not being afraid to throw out an afternoon or a day of programming and begin again if all is not right. It means being able to stay in touch with one's instincts (and test results) concerning what is working and what is not.

Courage also means responding to concrete feedback, acting on your teammates' hunch when they believe that they have a simpler, better way to accomplish your goal. Courage is a high-risk, high-reward value that encourages experimentation that can take the team to its goal more rapidly, in an innovative way. Courage means that you and your teammates trust each other and your customers enough to act in ways that will continuously improve what is being done on the project, even if they require throwing out code, rethinking solutions, or further simplifying approaches. Courage also implies that you, as a systems analyst, eagerly apply the practices of the agile approach.

Analysts can best reflect all of the four values through an attitude of humility. Historically, computer software was developed by experts who often thought they knew how to run a business

better than the local customers who were the true domain experts. Computer experts were often referred to as "gurus." Some of the gurus displayed large egos and insisted on their infallibility, even when customers did not believe it. Many gurus lacked the virtue of humility.

However, maintaining a humble attitude during systems development is critical. You must continually embrace the idea that if the user is expressing a difficulty, then that difficulty must be addressed. It cannot be ignored. Agile modelers are systems analysts who make suggestions, voice opinions, but never insist that they are right 100 percent of the time. Agile modelers possess the self-confidence to allow their customers to question, critique, and sometimes complain about the system under development. Analysts learn from their customers, who have been in business a long time.

THE BASIC PRINCIPLES OF AGILE MODELING. In a perfect world, customers and your software development team would see eye to eye and communication would not be necessary. We would all be in agreement at all times. We know that the ideal world doesn't exist. But how can we bring our software development projects closer to the ideal? Part of why this will not happen is that so far we are trying to operate on a vague system of shared values. They're a good beginning, but they are really not operationalized to the point at which we can measure our success in any meaningful way. So we work to derive the basic principles that can help us check whether what we are doing in our software project is actually measuring up to the values that we share.

Agile principles are the reflections and specifications of agile values. They serve as guidelines for developers to follow when developing systems. They also serve to set agile methodologies apart from the more traditional plan-driven methodologies such as SDLC as well as object-oriented methodologies.

Agile principles were first described by Beck et al. and have evolved ever since. These principles can be expressed in a series of sayings such as:

1. Satisfy the customer through delivery of working software
2. Embrace change, even if introduced late in development
3. Continue to deliver functioning software incrementally and frequently
4. Encourage customers and analysts to work together daily
5. Trust motivated individuals to get the job done
6. Promote face-to-face conversation
7. Concentrate on getting software to work
8. Encourage continuous, regular, and sustainable development
9. Adopt agility with attention to mindful design
10. Support self-organizing teams
11. Provide rapid feedback
12. Encourage quality
13. Review and adjust behavior occasionally, and
14. Adopt simplicity.

Often you will hear agile developers communicate their point through sayings like those mentioned previously or even simpler phrases such as "model with a purpose," "software is your primary goal," and "travel light," a way of saying a little documentation is good enough. Listen to these carefully. These sayings (some call them proverbs) are further discussed in Chapter 16 under an analysis and document tool called FOLKLORE. Catchy phrases are easy to understand, easy to memorize, and easy to repeat. They are very effective.

Activities, Resources, and Practices of Agile Modeling

Agile modeling involves a number of activities that need to be completed sometime during the agile development process. This section discusses these activities, the resources, and the practices that are unique to the agile approach.

FOUR BASIC ACTIVITIES OF AGILE DEVELOPMENT. There are four basic activities of development that agile methods use. They are coding, testing, listening, and designing. The agile analyst needs to identify the amount of effort that will go into each activity and balance that with the resources needed to complete the project.

Coding is designated as the one activity that it is not possible to do without. One author states that the most valuable thing that we receive from code is "learning." The process is basi-

cally this: have a thought, code it, test it, and see whether the thought was a logical one. Code can also be used to communicate ideas that would otherwise remain fuzzy or unshaped. When I see your code, I may get a new thought. Source code is the basis for a living system. It is essential for development.

Testing is the second basic activity of development. The agile approach views automated tests as critical. The agile approach advocates writing tests to check the coding, functionality, performance, and conformance. Agile modeling relies on automated tests, and large libraries of tests exist for most programming languages. These tests need to be updated as necessary during the progress of the project.

There are both long-term and short-term reasons for testing. Testing in the short term provides you with extreme confidence in what you are building. If tests run perfectly you can continue on with renewed confidence. In the long term, testing keeps a system alive and allows you to make changes longer than would be possible if no tests were written or run.

The third basic activity of development is listening. In Chapter 4, we learned about the importance of listening during interviews. In the agile approach, listening is done in the extreme. Developers use active listening to hear their programming partner. In agile modeling there is less reliance on formal, written communication, and so listening becomes a paramount skill.

The developer also uses active listening with the customer. Developers assume that they know nothing about the business they are helping, and so they must listen carefully to businesspeople to get the answers to their questions. The developer needs to come to an understanding of what effective listening is. If you don't listen, you will not know what you should code or what you should test.

The fourth basic activity in development is designing, which is a way of creating a structure to organize all the logic in the system. Designing is evolutionary, and so systems that are designed using the agile approach are conceptualized as evolving, always being designed.

Good design is often simple. Design should allow flexibility as well. Designing well permits you to make extensions to the system by making changes only in one place. Effective design locates logic near the data on which it will be operating. Above all, design should be useful to all those who will need it as the development effort proceeds, including customers as well as programmers.

FOUR RESOURCE CONTROL VARIABLES OF AGILE MODELING. Completing all the activities in the project on time within all the constraints is admirable, but, as you probably have realized by now, in order to accomplish this, project management is crucial. Managing a project doesn't mean simply getting all the tasks and resources together. It also means that the analyst is faced with a number of trade-offs. Sometimes cost may be predetermined, at other junctures time may be the most important factor. These resource control variables (time, cost, quality, and scope) are discussed next.

TIME. You need to allow enough time to complete your project. Time, however, is split into many separate pieces. You need time to listen to the customers, time to design, time to code, and time to test.

One of our friends is an owner of a Chinese restaurant. Recently, he found himself short-staffed as one of the members of his reliable crew returned to Hong Kong to get married. The owner placed himself in the kitchen so the food was served on time, but stopped greeting his customers out front in the usual way. He sacrificed the listening activity to achieve another, but in this case he found out it was hurting his business. Customers wanted the attention.

It is the same in systems development. You can create quality software, but fail to listen. You can design a perfect system, but not allow enough time to test it. Time is difficult to manage. If you find yourself running short of time, what do you do?

The agile approach challenges the notion that more time will give you the results you want. Perhaps the customer would prefer that you finish on time rather than extending the deadline to add another feature. Customers, we often find, are happy if some of the functionality is up and running on time. Our experience shows that often a customer is 80-percent satisfied with the first 20 percent of the functionality. This means that when you complete the final 80 percent of the project, the customer may be only slightly happier than he or she was after you completed the first 20 percent. The message here is be careful not to extend your deadline. The agile approach insists on finishing on time.

COST. Cost is the second variable we can consider adjusting. Suppose that the activities of coding, designing, testing, and listening are weighing the project down, and the resources we put

into time, scope, and quality are not sufficient, even with a normal amount devoted to cost, to balance the project. Essentially we might be required to contribute more resources that require money to balance the project.

The easiest way to increase spending (and hence costs) is to hire more people. This may appear to be the perfect solution. If we hire more programmers, we'll finish faster. Right? Not necessarily. Picture hiring two people to repair a roof and increase that number to four. Soon the people are bumping into one another. Furthermore, they need to ask each other what still needs to be done. And if there's a lightning storm, no one will be working. Going from two to four doesn't mean it will take half of the time. Consider the required increase in communication and other intangible costs when you are considering hiring more people. Remember that when new people join a team, they do not know the project or the team. They will slow the original members down, because the original members must devote time to getting new members up to speed.

Overtime doesn't help much either. It increases the cost, but the productivity doesn't always follow. Tired programmers are less effective than alert programmers. Tired programmers take a long time to complete a task, and they also make mistakes that are even more time consuming to fix.

Is there anything else we can spend our money on? Perhaps. As you read later chapters you will read about a variety of tools that support analysts and programmers. These tools are often a wise investment. Analysts, for example, use graphical packages such as Microsoft Visio to communicate ideas about the project to others, and CASE tools such as Visible Analyst also help speed up projects.

Even new hardware could be a worthwhile expenditure. Laptops and smartphones improve productivity away from the office. Larger visual displays, Bluetooth-enabled keyboards and mice, and more powerful graphics cards can also increase productivity.

QUALITY. The third resource control variable is quality. If ideal systems are perfect, why is so much effort placed in maintaining systems? Are we already practicing agile development by sacrificing quality in software development? In Chapter 16 we will see the importance of quality and methods (such as TQM and Six Sigma) that help ensure software quality is high.

The agile philosophy, however, does allow the analyst to adjust this resource, and perhaps put less effort into maintaining quality than otherwise would be expected. Quality can be adjusted both internally and externally. Internal quality involves testing software for factors such as functionality (Does a program do what it is supposed to do?) and conformance (Does the software meet certain conformance standards and is it maintainable?). It usually doesn't pay to tinker with internal quality.

That leaves us with external quality, or how the customer perceives the system. The customer is interested in performance. Some of the questions a customer may ask are: Does the program act reliably (or do software bugs still exist)? Is the output effective? Does the output reach me on time? Does the software run effortlessly? Is the user interface easy to understand and use?

The extreme philosophy of agile development allows some of the external quality issues to be sacrificed. In order for the system to be released on time, the customer may have to contend with some software bugs. If we want to meet our deadline, the user interface may not be perfect. We can make it better in a follow-up version.

Commercial off-the-shelf software manufacturers do sacrifice quality, and it is debatable whether this is the correct approach. So don't be surprised when your PC software applications (not to mention your operating system and Web browser) are updated often, if developers are using extreme programming as one of their agile practices.

SCOPE. Finally, there is scope. In the agile approach, scope is determined by listening to customers and getting them to write down their stories. Then the stories are examined to see how much can be done in a given time to satisfy the customer. Stories should be brief and easy to grasp. Stories will be described in more detail later in this chapter, but here is a brief example showing four short stories from an online air travel system. Each story is shown in bold type:

Display alternative flights.
Prepare a list of the five cheapest flights.

Offer cheaper alternatives.
Suggest to customers that they travel on other days, make weekend stays, take special promotions, or use alternate airports.

Purchase a ticket.
Allow the customer to purchase a ticket directly using a credit card (check validity).

Allow the customer to choose his or her seat.

Direct the customer to a visual display of the airplane and ask the customer to select a seat.

Ideally, the analyst would be able to determine how much time and money was needed to complete each of these stories and be able to set the level of quality for them as well. It is obvious that this system must not sacrifice quality, or credit card purchases may be invalid or customers may show up at the airport without reservations.

Once again agile practices allow extreme measures, so in order to maintain quality, manage cost, and complete the project on time, the agile analyst may want to adjust the scope of the project. This can be accomplished by agreeing with the customer that one or more of the stories can be delayed until the next version of the software. For example, maybe the functionality of allowing customers to choose their own seats can be put off for another time.

In summary, the agile analyst can control any of the four resource variables of time, cost, quality, and scope. Agility calls for extreme measures and places a great deal of importance on completing a project on time. In doing so, sacrifices must be made and the agile analyst will find out that the trade-offs available involve difficult decisions.

FOUR CORE AGILE PRACTICES. Four core practices markedly distinguish the agile approach from other approaches: short releases; the 40-hour workweek; hosting an onsite customer; and using pair programming.

1. Short releases means that the development team compresses the time between releases of their product. Rather than releasing a full-blown version in a year, using the short release practice they will shorten the release time by tackling the most important features first, releasing that system or product, and then improving it later.
2. Forty-hour workweek means that agile development teams purposely endorse a cultural core practice in which the team works intensely together during a typical 40-hour workweek. As a corollary to this practice, the culture reinforces the idea that working overtime for more than a week in a row is very bad for the health of the project and the developers. This core practice attempts to motivate team members to work intensely at the job, and then to take time off so that when they return they are relaxed and less stressed. This helps team members spot problems more readily, and prevents costly errors and omissions due to ineffectual performance or burnout.
3. Onsite customer means that a user who is an expert in the business aspect of the systems development work is onsite during the development process. This person is integral to the process, writes user stories, communicates to team members, helps prioritize and balance the long-term business needs, and makes decisions about which feature should be tackled first.
4. Pair programming is an important core practice. It means that you work with another programmer of your own choosing. You both do coding, you both run tests. Often the senior person will take the coding lead initially, but as the junior person becomes involved, whoever has the clear vision of the goal will typically do the coding for the moment. When you ask another person to work with you, the protocol of pair programming says he or she is obligated to consent. Working with another programmer helps you clarify your thinking. Pairs change frequently, especially during the exploration stage of the development process. Pair programming saves time, cuts down on sloppy thinking, sparks creativity, and is a fun way to program.

How core agile practices interrelate with and support agile development activities, resources, and values is shown in Figure 6.7.

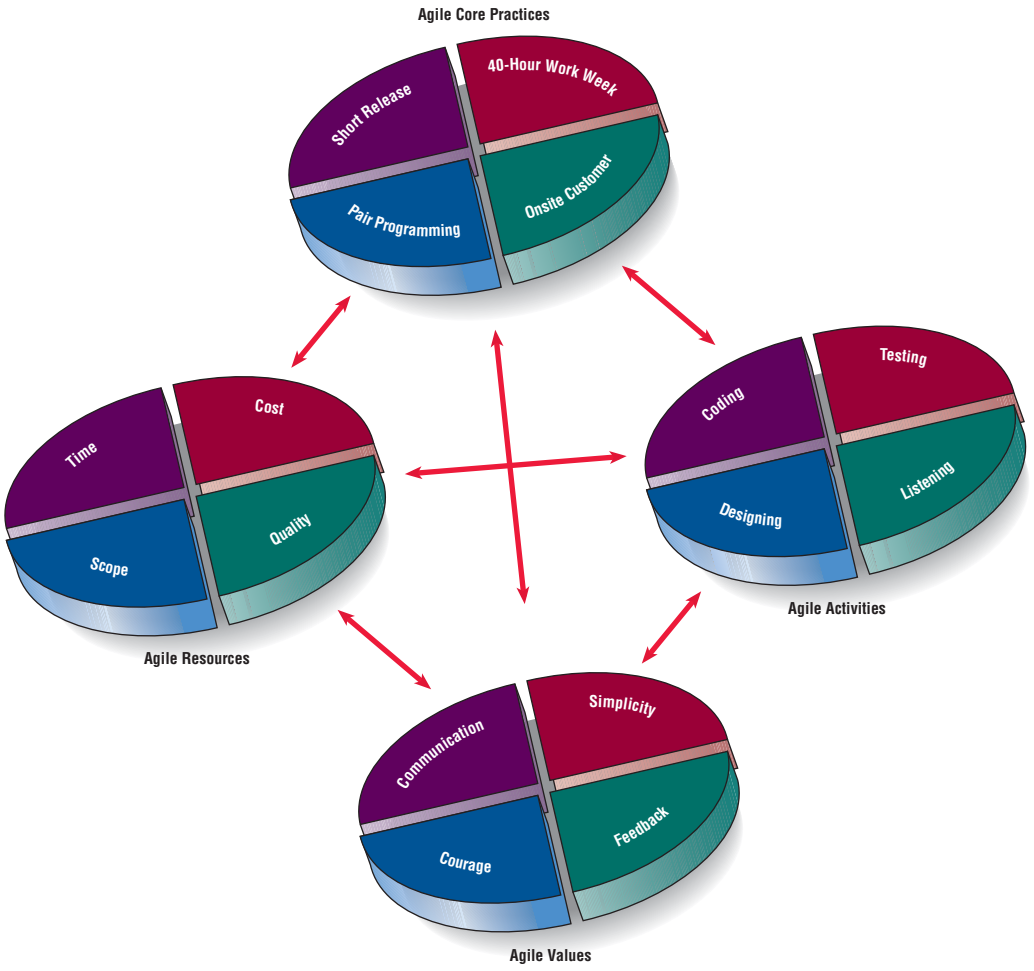
The Agile Development Process

Modeling is a keyword in agile methods. Agile modeling seizes on the opportunity to create models. These can be logical models such as drawings of systems, or mock-ups such as the prototypes described earlier in this chapter. A typical agile modeling process would go something like this:

1. Listen for user stories from the customer.
2. Draw a logical workflow model to gain an appreciation for the business decisions represented in the user story.
3. Create new user stories based on the logical model.

FIGURE 6.7

The core practices are interrelated with agile modeling's resources, activities, and values.



4. Develop some display prototypes. In doing so, show the customers what sort of interface they will have.
5. Using feedback from the prototypes and the logical workflow diagrams, develop the system until you create a physical data model.

Agile is the other keyword in agile modeling. Agile implies maneuverability. Today's systems, especially those that are Web-based, pose twin demands: getting software released as soon as possible and continually improving the software to add new features. The systems analyst needs to have the ability and methods to create dynamic, context-sensitive, scalable, and evolutionary applications. Agile modeling as such is a change-embracing method.

WRITING USER STORIES. Even though the title of this section is "Writing User Stories," the emphasis in the creation of user stories is on spoken interaction between developers and users, not the written communication. In user stories, the developer is seeking first and foremost to identify valuable business user requirements. Users will typically engage in conversations every day with the developers about the meaning of the user stories they have written. These frequent conversations are purposeful interactions that have as their goal the prevention of misunderstandings or misinterpretations of user requirements. Therefore, user stories serve as reminders to the developers that they must hold conversations devoted to those requirements.

The following is an example of a series of stories written for an ecommerce application for an online merchant of books, CDs, and other media products. The stories give a fairly complete picture of what is needed at each of the stages in the purchase process, but the stories are very short and easy to comprehend. The point here is to get all the needs and concerns of the online store out in the open. Although there is not enough of a story to begin programming, an agile

MAC APPEAL

Just as agile methodologies are an alternative to the SDLC, OmniFocus is an alternative to Microsoft Project or other Gantt Chart or PERT diagram approaches.

A casual observer might think that agile methods are unstructured because systems are built without detailed specifics and documentation. A student of agile methods realizes that there is actually quite a bit of structure in the agile approach. Principles include sticking to the 40-hour workweek and coordination through pair-programming. An analyst who adopts agile techniques needs a way to set goals, keep within budget, set priorities for features, and find a way to get things done.

OmniFocus is based on an alternative task management system by David Allen, called Getting Things Done. The overriding principle is to free your mind from remembering things, so that you can concentrate on completing them. An analyst using this system would go through five actions: collect, process, organize, review, and do.

Systems analysts using OmniFocus would collect items from their Web browser, their address book or their calendar, or most applications on a Mac. The analyst can categorize it or assign it to a larger project. OmniFocus contains a planning mode so the analyst can see which task is part of a larger project and a context mode that organizes the tasks so the analyst knows all the tasks that must be done either by phone, by browsing the Web, or by using email. OmniFocus is also available as an iPhone app.

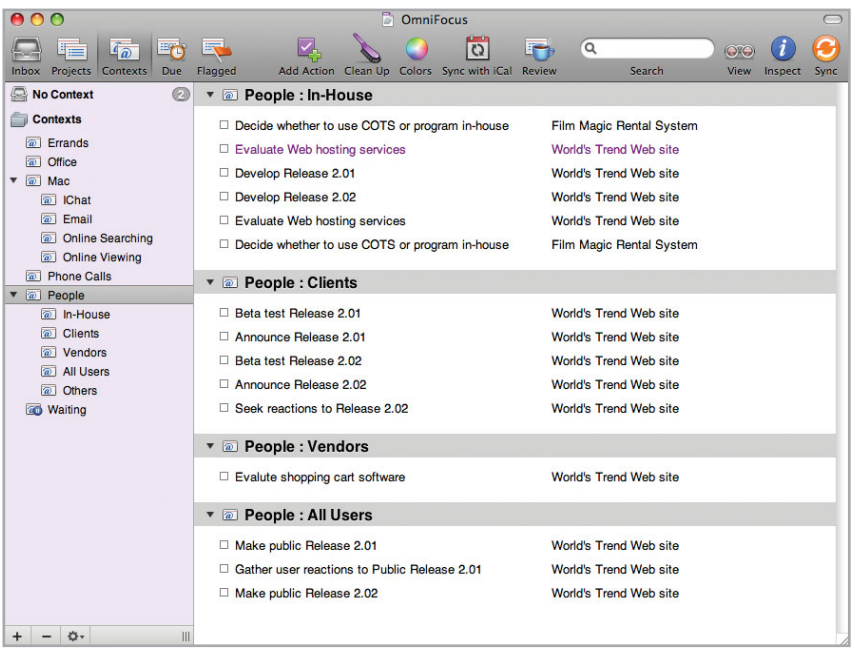


FIGURE 6.MAC

OmniFocus from The Omni Group.

developer might begin to see the overall picture clearly enough to begin estimating what it takes to complete the project. The stories are as follows:

Welcome the customer.

If the customer has been at this site before using this same computer, welcome the customer back to the online store.

Show specials on homepage.

Show any recent books or other products that have recently been introduced. If the customer is identified, tailor the recommendations to that specific customer.

Search for desired product.
Include an effective search engine that will locate the specific product and similar products.

Show matching titles and availability.
Display the results of the search on a new Web page.

Allow customer to ask for greater detail.
Offer the customer more product details, such as sample pages in a book, more photos of a product, or to play a partial track from a CD.

Display reviews of the product.
Share the comments that other customers have about the product.

Place a product into a shopping cart.
Make it easy for the customer to click on a button that places the product into a shopping cart of intended purchases.

Keep purchase history on file.
Keep details about the customer and his or her purchases in a cookie on the customer's computer. Also keep credit card information for faster checkout.

Suggest other books that are similar.
Include photos of other books that have similar themes or were written by the same authors.

Proceed to checkout.
Confirm the identity of the customer.

Review the purchases.
Allow the customer to review the purchases.

Continue shopping.
Offer the customer a chance to make further purchases at the same time.

Apply shortcut methods for faster checkout.
If the identity of the customer is known and the delivery address matches, speed up the transaction by accepting the credit card on file and the remainder of the customer's preferences, such as shipping method.

Add names and shipping addresses.
If the purchase is a gift, allow the customer to enter the name and address of the recipient.

Offer options for shipping.
Allow the customer to choose a shipping method based on cost.

Complete the transaction.
Finish the transaction. Ask for credit card confirmation if the shipping address is different from the customer's address on file.

As you can easily see, there is no shortage of stories. The agile analyst needs to choose a few stories, complete the programming, and release a product. Once this is done, more stories are selected and a new version is released until all the stories are included in the system (or the analyst and customer agree that a particular story lacks merit, or is not pressing, and so need not be included).

An example of a user story as it might appear to an agile developer is shown in Figure 6.8. On cards (or electronically), an analyst might first identify the need or opportunity, and then follow it with a brief story description. The analyst might take the opportunity to begin thinking broadly about the activities that need to be completed as well as the resources it will take to finish the project. In this example from the online merchant, the analyst indicates that the designing activity will take above-average effort, and the time and quality resources are required to rise above average. Notice that the analyst is not trying to be more precise than currently possible on this estimate, but it is still a useful exercise.

SCRUM. Another agile approach is named Scrum. The word *scrum* is taken from a starting position in rugby in which the rugby teams form a huddle and fight for possession of the ball. Scrum is really about teamwork, similar to what is needed in playing a game of rugby.

Need or Opportunity:	Apply shortcut methods for faster checkout.					
Story:	If the identity of the customer is known and the delivery address matches, speed up the transaction by accepting the credit card on file and the rest of the customer's preferences such as shipping method.					
		Well Below	Below Average	Average	Above Average	Well Above
Activities:	Coding			✓		
	Testing			✓		
	Listening			✓		
	Designing				✓	
Resources:	Time				✓	
	Cost		✓			
	Quality				✓	
	Scope			✓		

FIGURE 6.8

User stories can be recorded on cards. The user story should be brief enough for an analyst to determine what systems features are needed.

Just as rugby teams will come to a game with an overall strategy, development teams begin the project with a high-level plan that can be changed on the fly as the "game" progresses. Systems development team members realize that the success of the project is most important, and their individual success is secondary. The project leader has some, but not much, influence on the detail. Rather, the tactical game is left up to the team members, just as if they were on the field. The systems team works within a strict time frame (30 days for development), just as a rugby team would play in a strict time constraint of a game.

We can describe the components of the scrum methodology as:

1. Product backlog, in which a list is derived from product specifications.
2. Sprint backlog, a dynamically changing list of tasks to be completed in the next sprint.
3. Sprint, a 30-day period in which the development team transforms the backlog into software that can be demonstrated.
4. Daily scrum, a brief meeting in which communication is the number-one rule. Team members need to explain what they did since the last meeting, whether they encountered any obstacles, and what they plan to do before the next daily scrum.
5. Demo, working software that can be demonstrated to the customer.

Scrum is indeed a high-intensity methodology. It is just one of the approaches that adopts the philosophy of agile modeling.

Lessons Learned from Agile Modeling

Often posed as an alternative way to develop systems, the agile approach seeks to address common complaints arising over the traditional SDLC approach (for being too time-consuming, focusing on data rather than on humans, and being too costly) by being rapid, iterative, flexible, and participative in responding to changing human information requirements, business conditions, and environments.

Several agile development projects have been chronicled in books, articles, and on Web sites. Many of them were successes, some have been failures, but we can learn a great deal from studying them, as well as the agile values, principles, and core practices. Following are the six major lessons we draw from our examination of agile modeling. Figure 6.9 depicts the six lessons.

The first lesson is that short releases allow systems to evolve. Product updates are made often, and changes are incorporated quickly. In this way the system is permitted to grow and expand in ways that the customer finds useful. Through the use of short releases, the development team compresses the time between releases of their product, improving the product later as the dynamic situation demands.

The second lesson is that pair programming enhances overall quality. Although pair programming is controversial, it clearly fosters other positive activities necessary in systems development

FIGURE 6.9

There are six vital lessons that can be drawn from the agile approach to systems.

such as good communication, identifying with the customer, focusing on the most valuable aspects of the project first, testing all code as it is developed, and integrating the new code after it successfully passes its tests.

The third lesson is that onsite customers are mutually beneficial to the business and the agile development team. Customers serve as a ready reference and reality check, and the focus of the system design will always be maintained via their presence: customers become more like developers and developers empathize more fully with customers.

The fourth lesson we take from the agile approach is that the 40-hour workweek improves effectiveness. Even the hardest-hitting developers are susceptible to errors and burnout if they work too hard for too long a period. When the development team is together, however, every moment counts. Working at a sustainable pace is much more desirable for the life of the project, the life of the system, and the life of the developer! We all know the parable of the hare and the tortoise.

The fifth lesson we draw from taking the agile approach is that balanced resources and activities support project goals. Managing a project doesn't mean simply getting all resources and tasks together. It also means that the analyst is faced with a number of trade-offs. Sometimes cost may be predetermined, at other junctures time may be the most important factor. The resource control variables of time, cost, quality, and scope need to be properly balanced with the activities of coding, designing, testing, and listening.

The last lesson we take from agile modeling approaches is that agile values are crucial to success. It is essential to the overall success of the project that analysts wholeheartedly embrace the values of communication, simplicity, feedback, and courage in all the work that they do. This type of personal and team commitment enables the analyst to succeed where others, who possess similar technical competencies but who lack values, will fail. True dedication to these values is fundamental to successful development.

COMPARING AGILE MODELING AND STRUCTURED METHODS

As you have seen, agile methods are developed quickly; they reportedly work; and users are customers who are directly involved. While it is true that projects developed by agile methods often require tweaking to work properly, agile developers admit that tweaking is part of the process. The agile approach implies many short releases with features added along the way.

Improving Efficiency in Knowledge Work: SDLC Versus Agile

Researchers (Davis & Naumann, 1999) developed a list of seven strategies that can improve the efficiency of knowledge work: reducing interface time and errors; reducing process learning time and dual processing losses; reducing time and effort to structure tasks and format outputs; reducing nonproductive expansion of work; reducing data and knowledge search and storage time and costs; reducing communication and coordination time and costs; and reducing losses from human information overload. They believe this is important, since based on their study of a group of programmers, they claim that the best programmers are five to ten times more productive than the worst ones. They further point out this ratio is only two to one for workers in clerical or physical tasks. Their suggestion is that software can help improve many situations.

We use the standard, traditional systems development approach of structured methods to compare and contrast how structured approaches versus agile methods would implement the seven strategies proposed to improve the efficiency of knowledge workers.

While adopting more software may indeed improve performance, it is reasonable to suggest that changing an approach or methodology may also improve performance. Consequently, we will examine each aspect of knowledge work productivity through lenses from both structured and agile methodologies. Figure 6.10 lists the original seven strategies for productivity improvement and then explains what methods are used to improve the efficiency of systems development for both structured and agile methodologies.

In the upcoming sections we will compare and contrast structured approaches with the agile approach. An overarching observation about the agile methodology is that it is a human-oriented approach that permits people to create nuanced solutions that are impossible to create through formal specifications of process.

REDUCING THE INTERFACE TIME AND ERRORS. Systems analysts and programmers need to analyze, design, and develop systems using knowledge work tools that range from Microsoft Office to sophisticated and costly CASE tools. They also need to document as they develop systems. It is important that analysts and programmers are capable of understanding the interface they use. They need to know how to classify, code, store, and write about the data they gather. Systems developers also need to quickly access a program, enter the required information, and retrieve it when it is needed again.

FIGURE 6.10

How Davis and Naumann's strategies for improving efficiency can be implemented using two different development approaches.

Strategies for Improving Efficiency in Knowledge Work	Implementation Using Structured Methodologies	Implementation Using Agile Methodologies
Reduce interface time and errors	Adopting organizational standards for coding, naming, etc.; using forms	Adopting pair programming
Reduce process learning time and dual processing losses	Managing when updates are released so the user does not have to learn and use software at the same time	Ad hoc prototyping and rapid development
Reduce time and effort to structure tasks and format outputs	Using CASE tools and diagrams; using code written by other programmers	Encouraging short releases
Reduce nonproductive expansion of work	Project management; establishing deadlines	Limiting scope in each release
Reduce data and knowledge search and storage time and costs	Using structured data gathering techniques, such as interviews, observation, sampling	Allowing for an onsite customer
Reduce communication and coordination time and costs	Separating projects into smaller tasks; establishing barriers	Timeboxing
Reduce losses from human information overload	Applying filtering techniques to shield analysts and programmers	Sticking to a 40-hour workweek

Structured approaches encourage adopting standards for everything. Rules set forth include items such as, "Everyone must use Microsoft Word rather than Word Perfect." They may be more detailed instructions to ensure clean data such as, "Always use M for Male and F for Female," thereby ensuring that analysts do not unthinkingly choose codes of their own, such as 0 for Male and 1 for Female. These rules then become part of the data repository. Forms are also useful, requiring all personnel to document their procedures so that another programmer might be able to take over if necessary.

In an agile approach, forms and procedures work well too, but another element is added. The additional practice of pair programming assures that one programmer will check the work of another, thereby reducing the number of errors. Pair programming means that ownership of the design or software itself is shared as in a partnership. Both partners (typically one a programmer, often a senior one) will say they chose a programming partner who desired to have a quality product that is error-free. Since two people work on the same design and code, interface time is not an issue; it is an integral part of the process. The authors have noted that programmers are quite emotional when the topic of pair programming is broached.

REDUCING THE PROCESS LEARNING TIME AND DUAL PROCESSING LOSSES. Analysts and programmers learn specific techniques and software languages required for the completion of a current project. Inefficiencies often result when some analysts and programmers already know the products used while others still need to learn them. Typically, we ask that developers learn these products at the same time they are using them to build the system. This on-the-job training slows down the entire systems development project considerably.

A traditional, structured project requires more learning. If CASE tools were used, an analyst may need to learn the proprietary CASE tools used in the organization. The same applies to the use of a specific computer language. Documentation is also a concern.

Using an agile philosophy, the ability to launch projects without using CASE tools and detailed documentation allows the analysts and programmers to spend most of their time on system development rather than on learning specific tools.

REDUCING THE TIME AND EFFORT TO STRUCTURE TASKS AND FORMAT OUTPUTS. Whenever a project is started, a developer needs to determine the boundaries. In other words, the developers need to know what the deliverable will be and how they will go about organizing the project so they can complete all the necessary tasks.

A traditional approach would include using CASE tools, drawing diagrams (such as E-R diagrams and data flow diagrams), using project management software (such as Microsoft Project), writing detailed job descriptions, using and reusing forms and templates, and reusing code written by other programmers.

Systems development using an agile approach addresses the need to structure tasks by scheduling short releases. The agile philosophy suggests that system developers create a series of deadlines for many releases of the system. The first releases would possess fewer features, but, with each new release, additional features would be added.

REDUCING THE NONPRODUCTIVE EXPANSION OF WORK. Parkinson's law states that "work expands so as to fill the time available for its completion." If there are no specified deadlines, it is possible that knowledge work will continue to expand.

With traditional structured methodologies, deadlines at first seem far into the future. Analysts may use project management techniques to try to schedule the activities, but there is a built-in bias to extend earlier tasks longer than they need to be and then try to shorten tasks later on in the development. Analysts and programmers are less concerned about distant deadlines than approaching ones.

Once again, the agile approach stresses short releases. Releases can be delivered at the time promised, minus some of the features originally promised. Making all deadlines imminent pushes a realistic expectation for (at least partial) completion to the fore.

REDUCING THE DATA AND KNOWLEDGE SEARCH AND STORAGE TIME AND COSTS. System developers need to gather information about the organization, goals, priorities, and details about current information systems before they can proceed to develop a new system. Data-gathering methods include interviewing, administering questionnaires, observation, and investigation by examining reports and memos.

Structured methodologies encourage structured data-gathering methods. Structured techniques would normally be used to structure interviews and design the interview process. Questionnaires would be developed in a structured way, and structured observational techniques such as STROBE would encourage the analyst to specifically observe key elements and form conclusions based on the observations of the physical environment. A sampling plan would be determined quantitatively, in order for the systems analyst to select reports and memos to examine.

Knowledge searches are less structured in an agile modeling environment. The practice of having an onsite customer greatly enhances access to information. The onsite customer is present to answer questions about the organization itself, its goals, the priorities of organizational members and customers, and whatever knowledge is necessary about existing information systems. As the project continues, the picture of customer requirements becomes clearer. This approach seems relatively painless because, when the system developers want to know something, they can just ask. The downside, however, is that the onsite representative may make up information if it is unknown or unavailable or evade telling the truth for some ulterior purpose.

REDUCING COMMUNICATION AND COORDINATION TIME AND COSTS. Communication between analysts and users, as well as among analysts themselves, is at the heart of developing systems. Poor communication is certainly the root of multiple development problems. We know that communication increases when more people join the project. When two people work on a project, there is one opportunity for a one-to-one conversation; when three people are involved, there are three possibilities; when four are involved, there are six possibilities, and so on. Inexperienced team members need time to get up to speed, and they can slow down a project even though they are meant to help expedite it.

Traditional structured development encourages the separation of big tasks into smaller tasks. This allows more tightly knit groups and decreases the time spent communicating. Another approach involves setting up barriers. For example, customers may not be given access to programmers. This is a common practice in many industries. However, increased efficiency often means decreased effectiveness, and it has been noted that dividing up groups and setting up barriers will often introduce errors.

Agile methods, on the other hand, limit time instead of tasks. Timeboxing is used in agile methodologies to encourage completion of activities in shorter periods. Timeboxing is simply setting a time limit of one or two weeks to complete a feature or module. The agile method scrum puts a premium on time, while the developers communicate effectively as a team. Since communication is one of the four values of the agile philosophy, communication costs tend to increase rather than decrease.

REDUCING LOSSES FROM HUMAN INFORMATION OVERLOAD. We have long known that people do not react well in information overload situations. When telephones were an emerging technology, switchboard operators manually connected calls between two parties. It was demonstrated that this system would work until an information overload occurred, at which point the entire system broke down. When too many calls came in, the overwhelmed switchboard operator would simply stop working and give up completely on connecting callers. An analogous overload situation can occur anytime to anyone, including systems analysts and programmers.

A traditional approach would be to try to filter information to shield analysts and programmers from customer complaints. This approach allows developers to continue working on the problem without the interference and subjectivity that would normally occur.

Using an agile philosophy, analysts and programmers are expected to stick to a 40-hour workweek. This might be viewed by some as a questionable practice. How will all the work ever get done? The agile philosophy states, however, that quality work is usually done during a routine schedule, and it is only when overtime is added that problems of poor quality design and programming enter the scene. By sticking to a 40-hour work week schedule, agile methodology claims you will eventually come out ahead.

Risks Inherent in Organizational Innovation

In consultation with users, analysts must consider the risks that organizations face when adopting new methodologies. Clearly this is part of a larger question of when is the appropriate time to upgrade human skills, adopt new organizational processes, and institute internal change.

FIGURE 6.11

Adopting new information systems involves balancing several risks.

In the larger sense, these are questions of a strategic dimension for organizational leadership. Specifically, we consider the case of the systems analysis team adopting agile methods in light of the risks to the organization and the eventual successful outcome for the systems development team and their clients. Figure 6.11 shows many of the variables that need to be considered when assessing the risk of adopting organizational innovation.

ORGANIZATIONAL CULTURE. A key consideration is the overall culture of the organization and how the culture of the development team fits within it. A conservative organizational culture with many stable features that does not seek to innovate may be an inappropriate or even inhospitable context for the adoption of agile methodologies by the systems development group. Analysts and other developers must use caution in introducing new techniques into this type of setting, since their success is far from assured, and long-standing development team members or other organizational members may be threatened by new ways of working that depart from customary, dependable approaches with proven results.

Conversely, an organization that is dependent on innovation to retain its cutting edge in its industry might be the organization most welcoming toward agile innovations in systems development methods. In this instance, the culture of the organization is already permeated with the understanding of the critical nature of many of the core principles of agile development methodologies. From the strategic level downward, the company's members have internalized the need for rapid feedback, dynamic responses to changing environments in real time, dependence on the customer for guidance and participation in problem solving, and so on.

Located between these extremes are organizations that do not rely on innovation as a key strategic strength (in other words, they are not dependent on research and development of new products or services to remain afloat) but that might still wish to adopt innovative practices in small units or groups. Indeed, such small, innovative centers or kernels might eventually drive the growth or competitive advantage of this type of organization.

TIMING. Organizations must ask and answer the question of when is the best time to innovate with the adoption of new systems development methodologies, when all other projects and factors (internally and externally) are taken into account. Organizations must consider the entire panoply of projects in which they are investing, look ahead at project deadlines, schedule the upgrading of physical plants, and absorb key industry and economic forecasts.

COST. Another risk to the adoption of agile methodologies for organizations is the cost involved in education and training of systems analysts and programmers in the new approach. This can involve either costly off-site seminars and courses or hiring consultants to work with current staff onsite. Further, opportunity costs are involved when systems developers are necessarily diverted (albeit temporarily) from ongoing projects to learn new skills. Education in itself can be costly, but an additional burden is recognized when analysts cannot earn income during their training period.

CLIENTS' REACTIONS. When clients (whether they are internal or external) are involved as users or initiators of information systems development efforts, reactions to the use of new methods entailed by the agile approach are also a key consideration. Some clients react with joy once the benefits of timeliness and involvement are described. Others do not want to be used for systems "experiments" with uncertain outcomes. The client-analyst relationship must be resilient enough to absorb and adapt to changes in expected behaviors. For example, the onsite presence of a client during development is a major commitment that should be thoroughly understood and agreed upon by those adopting agile methods.

MEASURING IMPACT. Another consideration for organizations adopting agile methodologies is how to certify and measure that the new methods are going to facilitate successful systems development. The strengths and weaknesses of traditional structured methods used to develop information systems are well-known.

While there is ample anecdotal evidence that agile methodologies are superior for development under some conditions, their history is short-lived and not yet empirically supported. Therefore, the adoption of agile methodologies carries with it the risk that systems created with them will not be successful or will not adequately interface with legacy systems. Measuring the impact of the use of agile methodologies has begun, but organizations need to be vigilant in proposing impact measurements in tandem with the adoption of new methods.

THE INDIVIDUAL RIGHTS OF PROGRAMMERS/ANALYSTS. Successful systems developers (analysts and programmers) exercise creativity in their approach to their work, and they deserve the right to work in the most fruitful configuration possible. It is possible that the working requirements of new agile methods (for example, pair programming) encroach upon some basic rights of creative people to work alone or in groups as the design work dictates. There is no "one best way" to design a system, module, interface, form, or Web page. In the instance of systems developers, creativity, subjectivity, and the right to achieve design objectives through numerous individual paths need to be balanced against the organizational adoption of innovative approaches such as agile methodologies.

As you can see, adopting organizational innovations poses many risks to the organization as well as to individuals. We examined risks to the organization as a whole as well as to those posed to the individual systems analyst who is caught up in the organization's desire to innovate.

SUMMARY

Prototyping is an information-gathering technique useful for supplementing the traditional SDLC; however, both agile methods and human–computer interaction share roots in prototyping. When systems analysts use prototyping, they are seeking user reactions, suggestions, innovations, and revision plans to make improvements to the prototype, and thereby modify system plans with a minimum of expense and disruption. The four major guidelines for developing a prototype are to (1) work in manageable modules, (2) build the prototype rapidly, (3) modify the prototype, and (4) stress the user interface.

Although prototyping is not always necessary or desirable, it should be noted that there are three main, interrelated advantages to using it: (1) the potential for changing the system early in its development, (2) the opportunity to stop development on a system that is not working, and (3) the possibility of developing a system that more closely addresses users' needs and expectations. Users have a distinct role to play in the prototyping process and systems analysts must work systematically to elicit and evaluate users' reactions to the prototype.

One particular use of prototyping is rapid application development (RAD). It is an object-oriented approach with three phases: requirements planning, the RAD design workshop, and implementation.

Agile modeling is a software development approach that defines an overall plan quickly, develops and releases software quickly, and then continuously revises software to add additional features. The values of the agile approach that are shared by the customer as well as the development team are communication, simplicity, feedback, and courage. Agile activities include coding, testing, listening, and designing. Resources available include time, cost, quality, and scope.

HYPERCASE® EXPERIENCE 6

"Thank goodness it's the time of year when everything is new. I love spring; it's the most exhilarating time here at MRE. The trees are so green, with leaves in so many different shades. So many new projects to do, too; so many new clients to meet. We have a new intern, too. Anna Mae Silver. Sometimes the newest employee is the most eager to help. Call on her if you need more answers."

"All the newness reminds me of prototyping. Or what I know about prototyping, anyway. It's something new and fresh, a quick way to find out what's happening.

"I believe that we have a few prototypes already started. Sometimes our new onsite customer, Tessa Silverstone, gets involved by helping create user stories on which to build the prototypes. But the best thing about prototypes is that they can change. I don't know anyone who's really been satisfied with a first pass at a prototype. But it's fun to be involved with something that's happening fast, and something that will change."

HYPERCASE Questions

1. Make a list of the user stories Tessa Silverstone shared as examples.
2. Locate the prototype currently proposed for use in one of MRE's departments. Suggest a few modifications that would make this prototype even more responsive to the unit's needs.
3. Using a word processor, construct a nonoperational prototype for a Training Unit Project Reporting System. Include features brought up by the user stories you found. *Hint:* See sample screens in Chapters 11 and 12 to help you in your design.

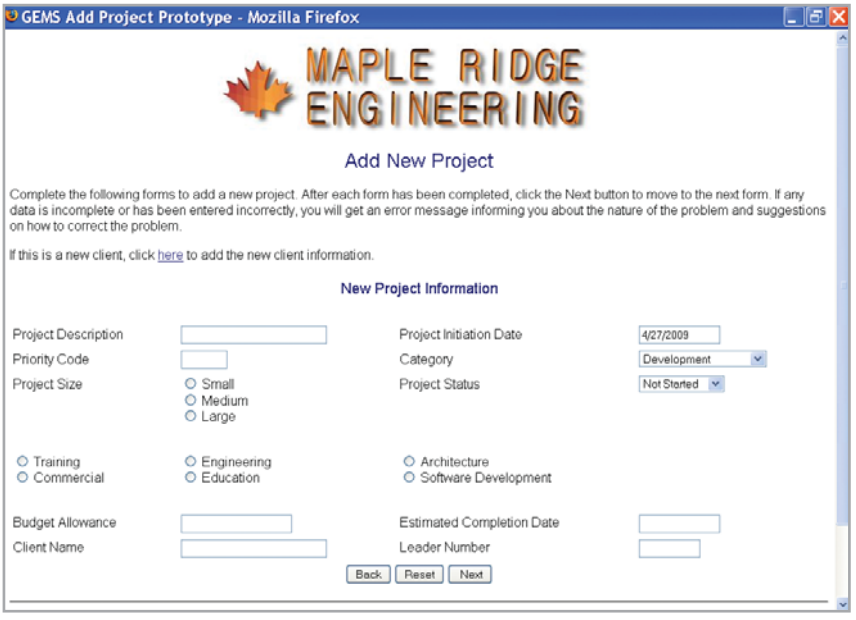


FIGURE 6.HC1

One of the many prototype screens found in HyperCase.

Agile core practices distinguish agile methods, including a type of agile method called extreme programming (XP), from other systems development processes. The four core practices of the agile approach are (1) short releases, (2) 40-hour workweek, (3) onsite customer, and (4) pair programming. The agile development process includes choosing a task that is directly related to a customer-desired feature based on user stories, choosing a programming partner, selecting and writing appropriate test cases, writing the code, running the test cases, debugging it until all test cases run, implementing it with the existing design, and integrating it into what currently exists.

Later in this chapter we compared how SDLC and agile approaches handle improving knowledge work efficiency differently. We then discussed several inherent dangers to organizations adopting innovative approaches, including an incompatible organizational culture, poor timing of the project, cost of training systems analysts, unfavorable client reactions to new behavioral expectations, difficulties in measuring the impact, and the possible compromise of the individual creative rights of programmers and analysts.

KEYWORDS AND PHRASES

40-hour workweek
agile modeling
agile principles
agile values
assume simplicity
embracing change
extreme programming (XP)
first-of-a-series prototype
implementation
incremental change
modifying the prototype
nonoperational prototype
onsite customer
pair programming
patched-up prototype
prototype
RAD design workshop
rapid application development (RAD)
rapid feedback
requirements planning phase
scrum methodology
selected-features prototype
short release
stressing the user interface
user involvement with prototyping
user stories
working in manageable modules

REVIEW QUESTIONS

1. What four kinds of information is the analyst seeking through prototyping?
2. What is meant by the term *patched-up prototype?*
3. Define a prototype that is a nonworking scale model.
4. Give an example of a prototype that is a first full-scale model.
5. Define what is meant by a prototype that is a model with some, but not all, essential features.
6. List the advantages and disadvantages of using prototyping to *replace* the traditional SDLC.
7. Describe how prototyping can be used to augment the traditional SDLC.
8. What are the criteria for deciding whether a system should be prototyped?
9. List four guidelines the analyst should observe in developing a prototype.
10. What are the two main problems identified with prototyping?
11. List the three main advantages in using prototyping.
12. How can a prototype mounted on an interactive Web site facilitate the prototyping process? Answer in a paragraph.
13. What are three ways that a user can be of help in the prototyping process?
14. Define what is meant by RAD.
15. What are the three phases of RAD?
16. What are the four values that must be shared by the development team and business customers when taking an agile approach?
17. What are agile principles? Give five examples.
18. What are the four core practices of the agile approach?
19. Name the four resource control variables used in the agile approach.
20. Outline the typical steps in an agile development episode.
21. What is a user story? Is it primarily written or spoken? State your choice, then defend your answer with an example.
22. List software tools that can aid the developer in doing a variety of tests of code.
23. What is scrum?
24. Name the seven strategies for improving efficiency in knowledge work.
25. Identify six risks in adopting organizational innovation.

PROBLEMS

1. As part of a larger systems project, Clone Bank of Clone, Colorado, wants your help in setting up a new monthly reporting form for its checking and savings account customers. The president and vice presidents are very attuned to what customers in the community are saying. They think that their customers want a checking account summary that looks like the one offered by the other three banks in town. They are unwilling, however, to commit to that form without a formal summary of customer feedback that supports their decision. Feedback will not be used to change the prototype form in any way. They want you to send a prototype of one form to one group and to send the old form to another group.
   a. In a paragraph discuss why it probably is not worthwhile to prototype the new form under these circumstances.
   b. In a second paragraph discuss a situation under which it would be advisable to prototype a new form.

2. C. N. Itall has been a systems analyst for Tun-L-Vision Corporation for many years. When you came on board as part of the systems analysis team and suggested prototyping as part of the SDLC for a current project, C. N. said, "Sure, but you can't pay any attention to what users say. They have no idea what they want. I'll prototype, but I'm not 'observing' any users."
   a. As tactfully as possible, so as not to upset C. N. Itall, make a list of the reasons that support the importance of observing user reactions, suggestions, and innovations in the prototyping process.
   b. In a paragraph, describe what might happen if part of a system is prototyped and no user feedback about it is incorporated into the successive system.
3. "Every time I think I've captured user information requirements, they've already changed. It's like trying to hit a moving target. Half the time, I don't think they even know what they want themselves," exclaims Flo Chart, a systems analyst for 2 Good 2 Be True, a company that surveys product use for the marketing divisions of several manufacturing companies.
   a. In a paragraph, explain to Flo Chart how prototyping can help her to better define users' information requirements.
   b. In a paragraph, comment on Flo's observation: "Half the time, I don't think they even know what they want themselves." Be sure to explain how prototyping can actually help users better understand and articulate their own information requirements.
   c. Suggest how an interactive Web site featuring a prototype might address Flo's concerns about capturing user information requirements. Use a paragraph.
4. Harold, a district manager for the multioutlet chain of Sprocket's Gifts, thinks that building a prototype can mean only one thing: a nonworking scale model. He also believes that this way is too cumbersome to prototype information systems and thus is reluctant to do so.
   a. Briefly (in two or three paragraphs) compare and contrast the other three kinds of prototyping that are possible so that Harold has an understanding of what prototyping can mean.
   b. Harold has an option of implementing one system, trying it, and then having it installed in five other Sprocket locations if it is successful. Name a type of prototyping that would fit well with this approach, and in a paragraph defend your choice.
5. "I've got the idea of the century!" proclaims Bea Kwicke, a new systems analyst with your systems group. "Let's skip all this SDLC garbage and just prototype everything. Our projects will go a lot more quickly, we'll save time and money, and all the users will feel as if we're paying attention to them instead of going away for months on end and not talking to them."
   a. List the reasons you (as a member of the same team as Bea) would give Bea to dissuade her from trying to scrap the SDLC and prototype every project.
   b. Bea is pretty disappointed with what you have said. To encourage her, use a paragraph to explain the situations you think would lend themselves to prototyping.
6. The following remark was overheard at a meeting between managers and a systems analysis team at the Fence-Me-In fencing company: "You told us the prototype would be finished three weeks ago. We're still waiting for it!"
   a. In a paragraph, comment on the importance of rapid delivery of a portion of a prototyped information system.
   b. List three elements of the prototyping process that must be controlled to ensure prompt delivery of the prototype.
   c. What are some elements of the prototyping process that are difficult to manage? List them.
7. Prepare a list of activities for a systems development team for an online travel agent that is setting up a Web site for customers. Now suppose you are running out of time. Describe some of your options. Describe what you will trade off to get the Web site released in time.
8. Given the situation for Williwonk's chocolates (Problem 1 in Chapter 3), which of the four agile modeling resource variables may be adjusted?
9. Examine the collection of user stories from the online merchant shown earlier in the chapter. The online media store would now like to have you add some features to its Web site. Following the format shown earlier in this chapter in Figure 6.9, write a user story for the features listed below:
   a. Include pop-up ads.
   b. Offer to share the details of the customer's purchases with his or her friends.
   c. Extend offer to purchase other items.
10. Go to the Palm gear Web site at www.palmgear.com. Explore the Web site and write up a dozen brief user stories for improving the Web site.
11. Go to the iTunes Web site and write up a dozen brief user stories for improving the Web site.
12. Using the stories you wrote for Problem 9, walk through the five stages of the agile development process and describe what happens at each one of the stages.

GROUP PROJECTS

1. Divide your group into two smaller subgroups. Have group 1 follow the processes specified in this chapter for creating prototypes. Using a CASE tool or a word processor, group 1 should devise two nonworking prototype screens using the information collected in the interviews with Maverick Transport employees accomplished in the group exercise in Chapter 4. Make any assumptions necessary to create two screens for truck dispatchers. Group 2 (playing the roles of dispatchers) should react to the prototype screens and provide feedback about desired additions and deletions.
2. The members of group 1 should revise the prototype screens based on the user comments they received. Those in group 2 should respond with comments about how well their initial concerns were addressed with the refined prototypes.
3. As a united group, write a paragraph discussing your experiences with prototyping for ascertaining information requirements.
4. Within your group, assign some of the roles that people take on in agile development. Make sure that one person is an onsite customer and at least two people are programmers. Assign other roles, as you see fit. Simulate the systems development situation discussed in Problem 7, or have the person acting as the onsite customer choose an ecommerce business with which he or she is familiar. Assume that the customer wants to add some functionality to his or her Web site. Role-play a scenario showing what each person would do if this was being approached through agile methods. Write a paragraph that discusses the constraints that each person faces in enacting his or her role.

SELECTED BIBLIOGRAPHY

Alavi, M. "An Assessment of the Prototyping Approach to Information Systems Development." *Communications of the ACM,* Vol. 27, No. 6, June 1984, pp. 556–563.

Avison, D., and D. N. Wilson. "Controls for Effective Prototyping." *Journal of Management Systems,* Vol. 3, No. 1, 1991.

Beck, K. *Extreme Programming Explained: Embrace Change.* Boston: Addison-Wesley Publishing Co., 2000.

Beck, K., and M. Fowler. *Planning Extreme Programming.* Boston: Addison-Wesley Publishing Co., 2001.

Cockburn, A. *Agile Software Development.* Boston: Addison-Wesley Publishing Co., 2002.

Davis, G. B., and J. D. Naumann. "Knowledge Work Productivity." In *Emerging Information Technologies: Improving Decisions, Cooperation, and Infrastructure.* Edited by K. E. Kendall, pp. 343–357. Thousand Oaks, CA: Sage, 1999. 42.

Davis, G. B., and M. H. Olson. *Management Information Systems: Conceptual Foundations, Structure, and Development,* 2d ed. New York: McGraw-Hill, 1985.

Fitzgerald, B., and G. Hartnett. "A Study of the Use of Agile Methods Within Intel," In *Business Agility & IT Diffusion.* Edited by L. Matthiassen, J. Pries-Heje, and J. DeGross, pp. 187–202. Proc Conference, Atlanta, May 2005. New York: Springer, 2005.

Ghione, J. "A Web Developer's Guide to Rapid Application Development Tools and Techniques." *Netscape World,* June 1997.

Gremillion, L. L., and P. Pyburn. "Breaking the Systems Development Bottleneck." *Harvard Business Review,* March–April 1983, pp. 130–137.

Harrison, T. S. "Techniques and Issues in Rapid Prototyping." *Journal of Systems Management.* Vol. 36, No. 6, June 1985, pp. 8–13.

Kendall, J. E., and K. E. Kendall. "Agile Methodologies and the Lone Systems Analyst: When Individual Creativity and Organizational Goals Collide in the Global IT Environment." *Journal of Individual Employment Rights*, Vol. 11, No. 4, 2004–2005, pp. 333–347.

Kendall, J. E., K. E. Kendall, and S. Kong. "Improving Quality Through the Use of Agile Methods in Systems Development: People and Values in the Quest for Quality." In *Measuring Information Systems Delivery Quality.* Edited by E. W. Duggan and H. Reichgelt, pp. 201–222. Hershey, PA: Idea Group Publishing, 2006.

Liang, D. *Rapid Java Application Development Using JBuilder 3.* Upper Saddle River, NJ: Prentice Hall, 2000.

McBreen, P. *Questioning Extreme Programming.* Boston: Addison-Wesley Publishing Co., 2003.

Naumann, J. D., and A. M. Jenkins. "Prototyping: The New Paradigm for Systems Development." *MIS Quarterly,* September 1982, pp. 29–44.

EPISODE 6

CPU CASE

ALLEN SCHMIDT, JULIE E. KENDALL, AND KENNETH E. KENDALL

Reaction Time

"We need to get a feel for some of the output needed by the users," Anna comments. "It will help to firm up some of our ideas on the information they require."

"Agreed," replies Chip. "It will also help us determine the necessary input. From that we can design corresponding data entry screens. Let's create prototype reports and screens and get some user feedback. Why don't we use Microsoft Access to quickly create screens and reports? I'm quite familiar with the software. Let's start by writing some agile stories to summarize what is needed and then develop some prototypes. We can also use the user requirements, and should create a prototype for each 'communicates' line connecting an actor and a use case."

Anna smiles and remarks, "I've already written the following agile stories for the preventive maintenance problem." They are:

1. There is no way to know when to perform preventive maintenance on desktop computers.
2. Normally we go from room to room.
3. When a room is completed, we write it on a list.

Anna starts by developing the PREVENTIVE MAINTENANCE REPORT prototype. Based on agile stories, she sets to work creating the prototype of the report she feels Mike Crowe will need.

"This report should be used to predict when machines should have preventive maintenance," Anna thinks. "It seems to me that Mike would need to know *which* machine needs work performed as well as *when* the work should be scheduled. Now let's see, what information would identify the machine clearly? The inventory number, brand name, and model would identify the machine. I imagine the room and campus should be included to quickly locate the machine. A calculated maintenance date would tell Mike when the work should be completed. What sequence should the report be in? Probably the most useful would be by location."

The PREVENTIVE MAINTENANCE REPORT prototype showing the completed report is shown in Figure E6.1. Notice that Xxxxxxx's and generic dates are used to indicate where data should be printed. Realistic campus and room locations as well as inventory numbers are included. They are necessary for Microsoft Access to accomplish group printing.

The report prototype is soon finished. After printing the final copy, Anna takes the report to both Mike Crowe and Dot Matricks. Mike Crowe is enthusiastic about the project and wants to know when the report will be in production. Dot is similarly impressed.

Several changes come up. Mike wants an area to write in the completion date of the preventive maintenance so the report can be used to reenter the dates into the computer. She also suggests that the report title be changed to WEEKLY PREVENTIVE MAINTENANCE REPORT. The next steps are to modify the prototype report to reflect the recommended changes and then have both Mike and Dot review the result.

The report is easily modified and printed. Dot is pleased with the final result. "This is really a fine method for designing the system," she comments. "It's so nice to feel that we are a part of the development process and that our opinions count. I'm starting to feel quite confident that the final system will be just what we've always wanted."

Mike has similar praise, observing, "This will make our work so much smoother. It eliminates the guesswork about which machines need to be maintained. And sequencing them by room is a fine idea. We won't have to spend so much time returning to rooms to work on machines."

Chip makes a note about each of these modifications on a Prototype Evaluation Form (like Figure 6.3 in the chapter). This form gets Chip organized and documents the prototyping process.

Chip and Anna next turn their attention to creating screen prototypes. "Because I like the hardware aspect of the system, why don't I start working on the ADD NEW COMPUTER screen design?" asks Chip.

"Sounds good to me," Anna replies. "I'll focus on the software aspects."

Chip analyzes the results of detailed interviews with Dot and Mike. He compiles a list of elements that each user would need when adding a computer. Other elements, such as location and maintenance information, would update the COMPUTER MASTER later, after the machine was installed.

"Having the database tables defined sure helps to make quick prototypes," Chip comments. "It didn't take very long to complete the screen. Would you like to watch me test the prototype?"

"Sure," replies Anna. "This is my favorite part of prototyping."

Preventive Maintenance Report

Week of 6/1/10

6/1/10

Page 1 of 1

Campus Location	Room Location	Inventory Number	Brand Name	Model	Last Preventive Maintenance Date	Done
Central Administration	11111	84004782	Xxxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxxxx	4/4/10	___
Central Administration	11111	90875039	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	3/24/10	___
Central Administration	11111	93955411	Xxxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxxxx	4/4/10	___
Central Administration	11111	99381373	Xxxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxxxx	3/24/10	___
Central Administration	22222	10220129	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxx	3/24/10	___
Central Administration	99999	22838234	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	3/24/10	___
Central Administration	99999	24720952	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxx	3/24/10	___
Central Administration	99999	33453403	Xxxxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	4/4/10	___
Central Administration	99999	34044449	Xxxxxxxxxxx	Xxxxxxxxxxxxxxxx	4/4/10	___
Central Administration	99999	40030303	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxxxx	4/4/10	___
Central Administration	99999	47403948	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	3/24/10	___
Central Administration	99999	56620548	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	4/4/10	___
Central Computer Science	22222	34589349	Xxxxxxxxxxx	Xxxxxxxxxxxxxxxx	3/24/10	___
Central Computer Science	22222	38376910	Xxxxxxxxxxx	Xxxxxxxxxxxxxxxx	3/24/10	___
Central Computer Science	22222	94842282	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxxx	3/24/10	___
Central Computer Science	99999	339393	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	4/4/10	___
Central Zoology	22222	11398423	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxx	3/24/10	___
Central Zoology	22222	28387465	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxx	4/4/10	___
Central Zoology	99999	70722533	Xxxxxxxxxxxxx	Xxxxxxxxxxxxxxx	3/24/10	___
Central Zoology	99999	99481102	Xxxxxxxxxxxxxx	Xxxxxxxxxxxxxxxxx	3/24/10	___

FIGURE E6.1

Prototype for PREVENTIVE MAINTENANCE REPORT. This report needs to be revised.

Chip executes the screen design as Anna, Mike, and Dot watch. The drop-down lists and check boxes make it easy to enter accurate data.

"I really like this," Dot says. "May I try adding some data?"

"Be my guest," replies Chip. "Try to add both invalid and valid data. And notice the help messages that appear at the bottom of the screen to indicate what should be entered. Why don't you try out this prototype for a day or two and get back to me? Then I'll make the changes that you recommend and have you review the revised prototype."

Anna returns to her desk and creates the ADD SOFTWARE RECORD screen design.

When Anna completes the screen design, she asks Cher to test the prototype. Cher keys information in, checks the drop-down list values, and views help messages.

"I really like the design of this screen and how it looks," remarks Cher. "It lacks some of the fields that would normally be included when a software package is entered, though, like the computer type that the software runs on, the memory required, and the processor speed. I would also like buttons to save the record and exit."

"Those are all doable. I'll make the changes and get back to you," replies Anna, making some notes to herself.

A short time later, Cher again tests the ADD SOFTWARE RECORD screen. It includes all the features that she requires. The completed screen design may be viewed using Microsoft Access. Notice that there is a line separating the software information from the hardware entries.

A few days later Dot visits Anna with suggested changes for the ADD NEW COMPUTER prototype. "I reviewed this with Mike and we like what we have seen, but we have some suggestions," remarks Dot. "One of the things that is missing is the operating system. We have a number of technical people that have multiple operating systems. Many of the Mac users have Windows installed on their Macs, and some of the

Windows users have Linux-type operating systems installed. We need to include multiple operating systems on the ADD NEW COMPUTER screen."

"This will take some time to update the tables and prototypes," comments Chip after a lengthy pause. "But this is why we use short cycles to develop, test, and get feedback. I'll modify the prototype and get back to you."

After some thought and reworking of database tables, the prototype is modified and sent to Mike for approval. After a few days, Mike stops in with some feedback. "This looks great!" exclaims Mike. "However, I realized that there are some additional requirements that I forgot to tell you about. We have a refresh program that replaces computers after an interval of time. The time period varies by type of machine and when it was purchased. When we purchase the machine, we estimate the refresh interval. Can we have the refresh interval added to the database and the prototype? We could use it to calculate the refresh date, and periodically scan the Computer Master for all computers that need to be refreshed."

Chip starts working on the modifications. "This agile development is interesting," he says, grinning at Anna. "I can see why it's used to discover the requirements."

The final version of the ADD NEW COMPUTER prototype screen created with Access is shown in Figure E6.2. Placed on the top of the screen are the current date and time as well as a centered screen title. Field captions are placed on the screen, with the characters left aligned. Check boxes are included for the warranty field, as well as a drop-down list for the type of optical drive. An operating system subform is included to select multiple operating systems in the lower right portion of the screen. "Add Record" and "Print" buttons are included.

"Chip, I was speaking with Dot and she mentioned that there has been funding for putting some of the information on the Web, as part of the Web site for technology support at CPU," comments Anna, looking up from her computer. "I have been busy creating a prototype for the Web page menus and the first screen, one to report technology problems. Because solving problems is Mike's area, I have invited him and Dot to review the prototype. Care to join the session?"

"Sure," replies Chip. "I am interested in working on the design of some of the Web pages."

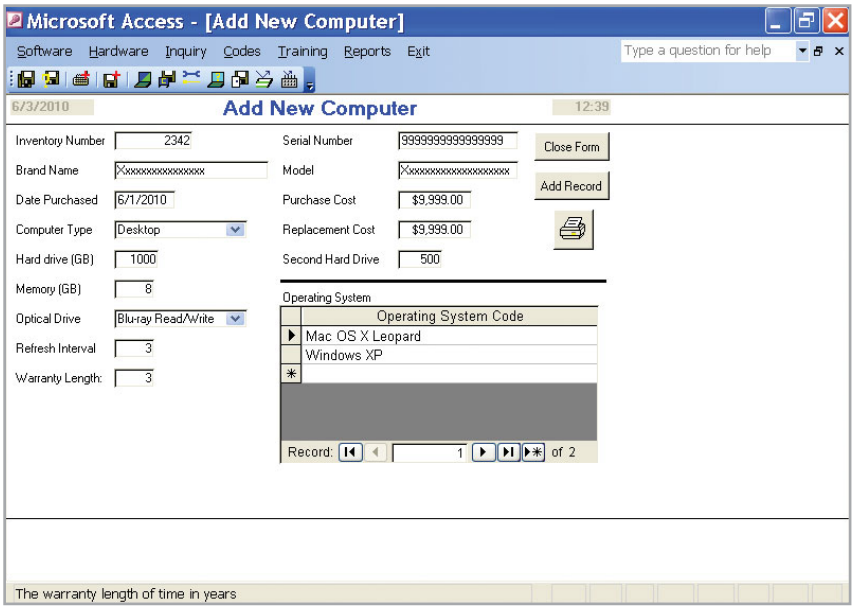
A short time later Mike, Dot, and Chip are gathered around Anna as she demonstrates the Web page, illustrated in Figure E6.3.

"I really like the menu style," comments Dot. "The main menu features drop-down submenus on the top that are easy to use, and I like the way they drop down and the menu items change color when the mouse moves over them."

"Yes, and having submenus drop down below the main one for the features of each choice makes it easy to find what you are looking for," adds Mike. "I do have some suggestions for the Web page for reporting problems, though. It would be more useful if the Problem Category selection area were moved to the top of the page. Each problem type is assigned to a different technician, one who is more or less an expert in that area. We need an additional check box to identify if it is a Macintosh or a Windows machine or software we are working with. The Tag Number help is a great idea. Many people do not realize that each piece of equipment has a small metal identifying tag on it with a unique inventory number. Hmmm. . . . That large blue area seems to stand out too much. After all, it is just help. I think that it would be better to replace it with a small graphic image."

FIGURE E6.2

Prototype for the ADD NEW COMPUTER screen. Microsoft Access was used as the prototyping tool. Improvements can be made at this stage.



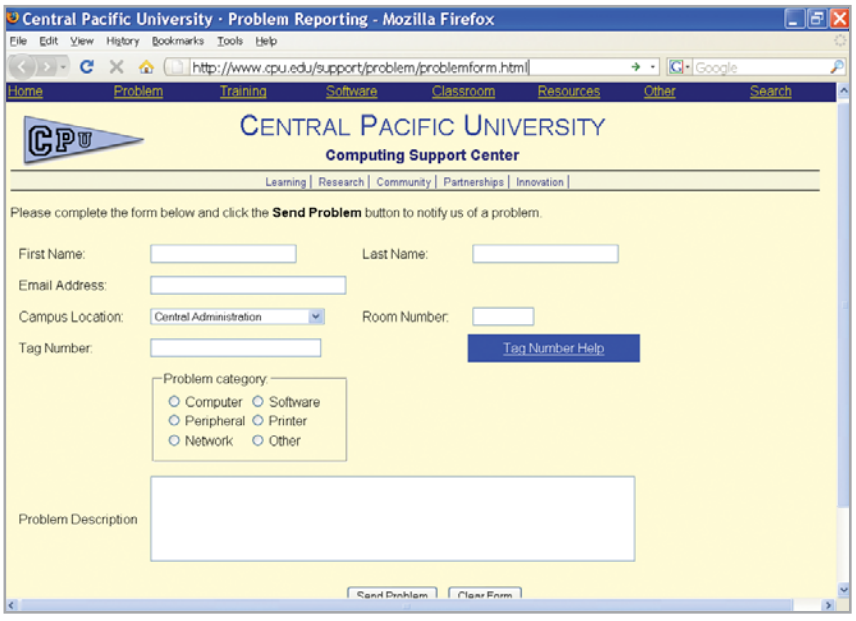


FIGURE E6.3

Prototype for the PROBLEM REPORTING SYSTEM Web page. This Web page needs some improvement.

"I think that these changes will be easy to do," remarks Anna.

"Great," replies Mike. "It would also be useful to include the tech support hotline phone number on the Web page. If it's a real emergency, it might speed up our resolution to the problem. We should add an entry field for their phone number as well. Of course, we could always look it up, but the person reporting the problem may be in a computer lab or another location away from his office."

"Good idea!" exclaims Dot. "This is going to be extremely helpful to the faculty and staff. I think that we should prototype all the Web pages for the site. I realize that Web pages are supposed to change from time to time, but let's get these as good as possible from the start! Why don't you look these over and give us feedback in a few days?"

Anna glances at Chip and grins. "I guess you'll be working on Web page design sooner than you think!"

A few days later Mike stops back with additional feedback on the Web design. "This Web page looks good," grins Mike, "But it got me thinking. We have an image of all the software on each lab computer. When there is a problem, such as a virus or bad hard drive, we fix it and re-image the machine. However each lab has different requirements for the software that should be on the machine. Additionally, we ask the faculty if the image needs updating. This usually happens at the end of the spring semester and we work on it over the summer. Can you whip up a couple of Web page prototypes for us to review? One should have a list of all the software, including browsers, virus detection, and other standard packages, that are included for each machine in a given room. Another Web page would be used for faculty to update the image list."

"Whew, that's a tall order," replies Anna with a thoughtful look. "We'll work on it."

Anna and Chip continued to work on prototypes by designing, obtaining user feedback, and modifying the design to accommodate user changes. Now that the work is complete, they have a solid sense of the requirements of the system.

"This is becoming a large project," comments Anna as she looks at the large amount of prototypes that have been assembled. "I don't think that we can develop all this software in the allotted amount of time."

"I agree with you—you seem to have a good sense about this," replies Chip thoughtfully. "We only have six months of development time to complete the project, including the Web pages. There's a lot of server code and JavaScript to write."

Anna puts down a stack of prototype evaluation forms and looks directly at Chip. "What are our options?"

Chip takes a moment to reflect and replies, "Well, compromising quality is not an option, and the due date is inflexible. That leaves cost and scope as trade-offs."

"Cost is somewhat fixed," replies Anna. "Dot and Paige have said in no uncertain terms that there are so many software development projects that we have to stay within our budget."

"Well, that means that we will have to reduce the scope of the project," says Chip after a moment. "We will work on the high priority items first."

"What about sacrificing a 40 hour workweek?" chides Anna.

"Not an option," grins Chip. "After all, it's a core value!"

EXERCISES

Critique the report and screen prototypes for the exercises below (E-1 through E-10). Record the changes on a copy of the Prototype Evaluation Form. Use Microsoft Access to view the prototypes, then modify the report and screen prototypes with the suggested changes. Print the final prototypes.

Use the following guidelines to help in your analysis:

1. ***Alignment of fields on reports.*** Are the fields aligned correctly? Are report column headers aligned correctly over the columns? If the report has captions to the left of data fields, are they aligned correctly (usually on the left)? Are the data aligned correctly *within* each entry field?
2. ***Report content.*** Does the report contain all the necessary data? Are appropriate and useful totals and subtotals present? Are there extra totals or data that should not be on the report? Are codes or the meaning of the codes printed on the report (codes should be avoided because they may not clearly present the user with information)?
3. ***Check the visual appearance of the report.*** Does it look pleasing? Are repeating fields group printed (that is, the data should print only once, at the beginning of the group)? Are there enough blank lines between groups to easily identify them?
4. ***Screen data and caption alignment.*** Are the captions correctly aligned on the screens? Are the data fields correctly aligned? Are the data *within* a field correctly aligned?
5. ***Screen visual appearance.*** Does the screen have a pleasing appearance? Is there enough vertical spacing between fields? Is there enough horizontal spacing between columns? Are the fields logically grouped together? Are features, such as buttons and check boxes, grouped together?
6. ***Does the screen contain all the necessary functional elements?*** Look for missing buttons that would help the user work smoothly with the screen; also look for missing data, extra unnecessary data, or fields that should be replaced with a check box or drop-down list.

E-1. The HARDWARE INVENTORY LISTING shows all personal computers, sorted by campus and room.

E-2. The SOFTWARE INVESTMENT REPORT is used to calculate the total amount invested in software.

E-3. The INSTALLED COMPUTER REPORT shows the information for installed machines.

E-4. The prototype for the COMPUTER PROBLEM REPORT lists all machines sorted by the total cost of repairs and includes the number of repairs (some machines do not have a high cost, because they are still under warranty). This prototype is used to calculate the total cost of repairs for the entire university, as well as to identify the problem machines.

E-5. The NEW SOFTWARE INSTALLED REPORT shows the number of machines with each software package that are installed in each room of each campus.

E-6. The SOFTWARE CROSS-REFERENCE REPORT lists all locations for each version of each software package.

E-7. The DELETE COMPUTER RECORD screen is used to select computers to remove from the system. The entry area is the Hardware Inventory Number field. The other fields are for display only, to identify the machine. The users would like the ability to print each record before they delete it. They also want to scroll to the next and previous records. *Hint:* Examine the fields shown in the HARDWARE INVENTORY LISTING report.

E-8. An UPDATE MAINTENANCE INFORMATION screen enables Mike Crowe to change maintenance information about personal computers. Sometimes these are routine changes, such as the LAST PREVENTIVE MAINTENANCE DATE or the NUMBER OF REPAIRS, but other changes may occur only sporadically, such as the expiration of a warranty. The HARDWARE INVENTORY NUMBER is entered, and the matching COMPUTER RECORD is found. The BRAND and MODEL are displayed for feedback. The operator may then change the WARRANTY, MAINTENANCE INTERVAL, NUMBER OF REPAIRS, LAST PREVENTIVE MAINTENANCE DATE, and TOTAL COST OF REPAIRS fields. Mike would like to print the screen information, as well as undo any changes, easily.

E-9. The SOFTWARE LOCATION INQUIRY displays information about rooms and machines containing selected software. The TITLE, VERSION NUMBER, and OPERATING SYSTEM are entered. The output portion of the screen should show the CAMPUS LOCATION, ROOM LOCATION, HARDWARE INVENTORY NUMBER, BRAND NAME, and MODEL. Buttons allow the user to move to the next record, the previous record, and to close and exit the screen.

E-10. The UPDATE LAB IMAGE Web page prototype is shown in Figure E6.4. Review this Web page and suggest changes.

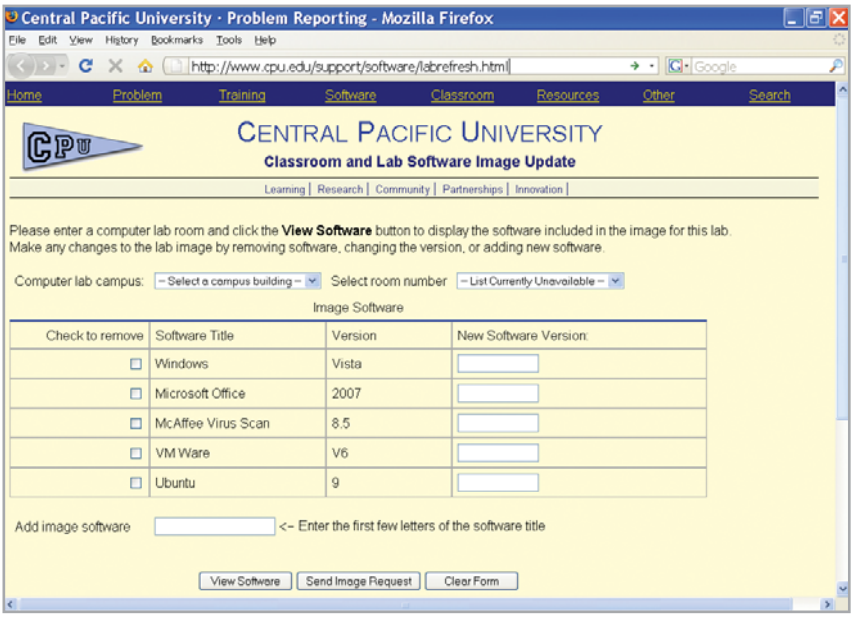


FIGURE E6.4

Prototype for the UPDATE LAB IMAGE Web page. This Web page needs some improvement.

The exercises preceded by a Web-icon indicate value-added material is available from the Web site at www.pearsonhighered.com/kendall. Students can download a sample Visible Analyst Project and a Microsoft Access database that can be used to complete the exercises.

This page intentionally left blank

CHAPTER 7

Using Data Flow Diagrams

LEARNING OBJECTIVES

Once you have mastered the material in this chapter you will be able to:

1. **Comprehend the importance of using logical and physical data flow diagrams (DFDs) to graphically depict data movement for humans and systems in an organization.**
2. **Create, use, and explode logical DFDs to capture and analyze the current system through parent and child levels.**
3. **Develop and explode logical DFDs that illustrate the proposed system.**
4. **Produce physical DFDs based on logical DFDs you have developed.**
5. **Understand and apply the concept of partitioning of physical DFDs.**

The systems analyst needs to make use of the conceptual freedom afforded by data flow diagrams, which graphically characterize data processes and flows in a business system. In their original state, data flow diagrams depict the broadest possible overview of system inputs, processes, and outputs, which correspond to those of the general systems model discussed in Chapter 2. A series of layered data flow diagrams may be used to represent and analyze detailed procedures in the larger system.

THE DATA FLOW APPROACH TO HUMAN REQUIREMENTS DETERMINATION

When systems analysts attempt to understand the information requirements of users, they must be able to conceptualize how data move through the organization, the processes or transformation that the data undergo, and what the outputs are. Although interviews and the investigation of hard data provide a verbal narrative of the system, a visual depiction can crystallize this information for users and analysts in a useful way.

Through a structured analysis technique called data flow diagrams (DFDs), the systems analyst can put together a graphical representation of data processes throughout the organization. By using combinations of only four symbols, the systems analyst can create a pictorial depiction of processes that will eventually provide solid system documentation.

Advantages of the Data Flow Approach

The data flow approach has four chief advantages over narrative explanations of the way data move through the system:

1. Freedom from committing to the technical implementation of the system too early.
2. Further understanding of the interrelatedness of systems and subsystems.
3. Communicating current system knowledge to users through data flow diagrams.
4. Analysis of a proposed system to determine if the necessary data and processes have been defined.