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 practices are becoming accepted, increasing in popularity, and reportedly working. 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.

Project management is important (as we saw in Chapter 3), so the agile approach tries to define an overall system plan quickly, develop and release software quickly, and then continuously revise the software to add additional features. Ordinary programmers, analysts, and designers who work independently and then integrate their work achieve solid results; agile programmers who work in pairs can be outstanding. But agile the approach is not based just on results. It is based on values, principles, and practices. We will now examine how the values and principles of agile modeling, including extreme programming (XP), shape the development of agile systems.
VALUES AND PRINCIPLES OF AGILE MODELING

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.”

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 6.7.

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. An agile coach, as described in Chapter 3, is present to observe whether anyone has stopped communicating, and to reunite them.

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.

Simplicity takes practice, and is something that the agile coach may have to help with. 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 to 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 a teammate’s 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.

Although there are about a dozen principles that we can usefully derive from our values, the basic principles that we describe are providing rapid feedback,
assuming simplicity, changing incrementally, embracing change, and encouraging quality work. These principles are illustrated in Figure 6.8.

The organizing principle to remember regarding rapid feedback is that in order for humans or the system to make a connection between a stimulus and reaction, the feedback must occur at a reasonable interval. If a printer runs out of paper, it must display an “out of paper” message instantly as feedback to the user, so that the situation can be remedied and the printing can continue. Rapid feedback for the development team means that the closer to the time of an action (coding a feature derived from a user story) to the time of the testing, the more meaningful the feedback (test results) will be. The earlier in the life of a system it is brought into production (rather than just being in development), the more value the feedback has to the business for gauging whether the system is meeting its goals.

The next basic principle is that the development team must assume simplicity. The premise is that over 90 percent of problems can be solved with utter simplicity. Notice that this flies in the face of most traditional training, which asks developers to plan for the future, figure out all of the interfaces, and so on, before beginning. Extreme programming (XP), an agile method that takes principles and practices to the extreme, says that “Simplicity rules the day.” Complexity can be added later. This is a very difficult principle for many developers to master.

Accepting incremental change is the third basic principle that we examine. This means that you are constantly making the smallest change possible that still results in a difference in the development effort. No sweeping changes. This goes for the code, the team, and the business requirements. They will all change incrementally, even after the product is released. This fits well with the agile idea of evolution.

A fourth basic principle that we can derive from the agile values is that of embracing change. We want to keep all of our options open, but we want to be able to simultaneously solve whatever presents the biggest obstacle. Though there are always trade-offs involved, we will know for sure that change is welcomed. That dynamism keeps the project moving forward and animates the spirit of the project team. Change is good.

The last principle is the notion of performing quality work. The principle stems from the idea that all participants want to do quality work. Otherwise, why would they even be involved in an agile effort? The point is to make work enjoyable, work well with the team, and keep the project alive and well.

There are a handful of other principles that will help developers know how to proceed when a certain situation comes up. Briefly, they include the mandate to teach learning; encouragement to make a small initial investment so that good, but
not extravagant, work is done; play to win, don’t play to avoid losing; and use concrete experiments to test the work that is being done.

Other important concepts that support the agile approach are the idea of using open and honest communication without fear; working with people’s natural tendencies (to want to succeed, interact with others, have autonomy in their work, be part of a winning team, be trusted, have their software work); claiming responsibility for a task rather than ordering others to do something; locally adapting the approach you are learning for agile development and seeking to use honest measurement that doesn’t pretend a preciseness that doesn’t exist. Agile modeling adds other principles such as “model with a purpose,” “software is your primary goal,” and “travel light,” a way of saying a little documentation is good enough.

**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 extreme programming.

**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 basically 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. Extreme programming 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 of 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**

In order to accomplish the activities described above, agile analysts need resources. Four resources can be adjusted to complete the project by its due date: time, cost, quality, and scope. When these four control variables are properly included in the planning, there is a state of balance between the resources and the activities needed to complete the project. A complete discussion of these resources and how they can be adjusted can be found in Chapter 3.

**Four Core Agile Practices**

Four core practices markedly distinguish the agile approach from other approaches: short releases; the 40-hour work week; 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 work week means that agile development teams purposely endorse a cultural core practice in which the team works intensely together during a typical 40-hour work week. 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 they are 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 6.9.

**THE AGILE DEVELOPMENT PROCESS AND TOOLS**

Now that you have learned about the activities, resources, and core practices of agile modeling, we can put that knowledge about agile modeling to work. This
The section describes the agile programming development process, explains the details involved in writing user stories, and examines some of the tools currently available for developing systems with an agile approach.

**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.
4. Develop some display prototypes. In doing so, show the customer 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, not dissimilar to extreme programming.

In Chapter 1 you learned about the SDLC and its many phases. Extreme programming also possesses a development process, but it is much more interactive, iterative, and integrative than the SDLC. However, the agile approach does not guide the developer through phases. Rather it is incremental and the activities are often done concurrently. Notice that many of the steps of the cycle for agile development are done every day. This is clearly in contrast to the SDLC, which proceeds at a much slower pace, and for which some activities (requirements analysis, testing, and so on) would be completed in distinct phases.

The five phases of the agile development process are exploration, planning, iterations to the first release, productionizing, and maintenance (for a detailed description of the stages, see Chapter 3). The following section describes specifically how a typical agile work session unfolds during the development process. For instance, the typical process would be to take on a task that is directly related to a system feature that a customer desires, test it, implement it into the existing design, and integrate it, all during one development episode. The day might begin by scrutinizing a user story task card, on which a specific task is written. During a brief, so-called stand-up meeting, you make a few inquiries about work done the previous day that could help in this task. You then ask another programmer to help on the task. Other onsite experts who might know answers to specific questions are quickly consulted. Then the existing group of test cases is consulted. There are probably some that will apply and some that must be written.

A next step would be to write down the next task on the to-do list. Then you might write a test case for whatever you are trying to find out. You finish and run it. It probably will fail. Together you and your partner look at other test cases and debug what you wrote. You continue on to the next test case and the next. Eventually, you are down to one item on the to-do list, which would be restructuring the other test cases. You do so.

You load the updated release and the changes. You then run all the test cases, debug any that are not running, and fix the code. When you rerun it and it works, you’re done. The code can then be released.

You may still be wondering just how to begin an agile development task. One wise author getting directly to the heart of the matter and being only slightly facetious, wrote, “Pick your worst problem, solve it the XP way. When it is no longer your worst problem, repeat” (Wells, as quoted in Beck, 2000, p. 123). In this way, you are showing great courage. You are daring to focus on solving your most pressing problem first, and you are applying the agile development strategies to work through that problem by testing, coding, listening, designing, and integrating. You are completing all the tasks of agile development in each daily programming assignment, and you are recognizing that the process of improving the system and addressing the hard problems simply and directly are keys to success.

**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 of 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 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 home page.**
*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 their 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 of 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 6.10. 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.

**Development Tools for Agile Modeling**  There are several tools that are favored by agile developers. The original creators of the XP approach were working in
SmallTalk, and eventually ported their unit testing framework (SUnit) to Java, which is now called JUnit. There are many resources on the Web that allow you to download xUnit testing frameworks for whatever software development language you are using.

The creators of XP and other agile approaches were careful not to saddle their principles with any particular development tool. This means that the agile approach can be as flexible as needed over time, and can also evolve with new tools that become available. Tools come and go, but the principles should remain intact regardless of that fluctuation.

Many of the tools used in agile development are inexpensive or entirely free. There is an excellent Web site, SourceForge.net, at which most of the software development tools can be found. As you will see from examining the site, extreme programming is used across many languages and software platforms, but the two clear leaders in terms of widespread use and popularity are Java and Microsoft .NET.

There are many different types of tools available that support the activities you would need to accomplish when doing agile development. These include tools that facilitate collaboration such as Wiki Wiki, Whiteboard, Project Web, NetMeeting, and IBM’s Rational ProjectConsole. There are also tools such as IBM’s Rational ClearCase, Visual Intercept, Compuware Track Record, and Blippt that support defect management.

Automated unit testers, acceptance testers, and GUI testers include JUnit, ComUnit, VBUnit, Nunit, httpUnit, and Rational Visual Test Tools. DevPartner Code Review helps with quality assurance. In addition there are tools that help with measuring system and component performance such as Jmeter, JUnitPerf, PerfMon, TrueTime, RealTime, and Microsoft Visual Studio Analyzer. There are also tools that assist with source code configuration management, including CVS, Visual Source Safe, and PVCS. Finally, there is a class of tools that you are probably already familiar with, the development environments of IBM VisualAge, Microsoft Visual Studio .NET, and JBuilder.

**LESSONS LEARNED FROM AGILE MODELING**

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 6.11 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 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 work week 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 of 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.

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.

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