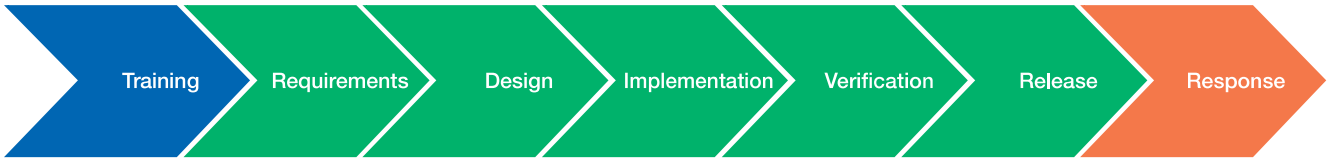


**FIGURE 1-7**
Microsoft's Security Development Lifecycle (SDL)
(*Source: http://www.microsoft.com/security/sdl/default.aspx.* Used by permission.)

the information systems that are procured are used inside the organization. Careful planning is required to determine which systems will be developed for an organization. Each system developed is an investment, and if the organization invests in a particular system, it cannot invest in some alternative system or in something else, such as a new store. Investment funds are limited, after all. Maintenance is also peculiar to an organizational context. Once systems go into general use, the organization needs to earn as much of a return on those investments as it can, so it is important that the systems run as long as possible. Companies such as Microsoft, which develop systems for others to use, do not need to worry about internal planning and maintenance in their product life cycles. Instead, as they have limited control over systems once they have been sold, they are concerned about selling a mature and reliable product. Therefore, they have two phases after implementation: verification and release. Verification involves quality assurance for products before they are released. Release involves all of the activities related to making the product available for general use. Next, note the two parts of the SDL that precede and follow the main development phases: training (in blue) and response (in orange). Two things make this particular SDLC specialized to security issues: the two unique phases that begin and end the life cycle (training and response), and the particular security activities associated with each phase in the development life cycle.

Training in Microsoft's SDL refers to the training each member of a development team receives about security basics and trends in security. The idea behind the training—indeed, the idea behind a specialized security development life cycle—is to have security become part of the development process from the beginning and not suddenly appear at the end of the SDLC. By training team members about security and how it can be addressed throughout the life cycle, security measures can be built into the system throughout its development. The response at the end of the SDL refers to a response plan developed during the release phase. If there is a security threat to a particular product, then the previously developed response plan is executed. The security-related activities that take place throughout the development life cycle vary by phase. Listing and explaining each activity is beyond the scope of this chapter, but we can provide some examples. One specialized activity performed during the requirements phase is a separate analysis of requirements related to both security and privacy. During design, developers can model threats to a system and consider how those threats differ with different design options. During implementation, project team members can conduct static analyses of source code, looking for security threats. During verification, they can conduct dynamic analyses. As part of the release phase, team members develop the incident response plan, mentioned previously, and they conduct a final security review. By adhering to a specialized life cycle devoted to security, project team members can ensure not only that security is addressed, but that it is addressed in a planned, systematic manner.

## THE HEART OF THE SYSTEMS DEVELOPMENT PROCESS

The SDLC provides a convenient way to think about the processes involved in systems development and the organization of this book. The different phases are clearly defined, their relationships to one another are well specified, and the sequencing of phases from one to the next, from beginning to end, has a compelling logic. In many

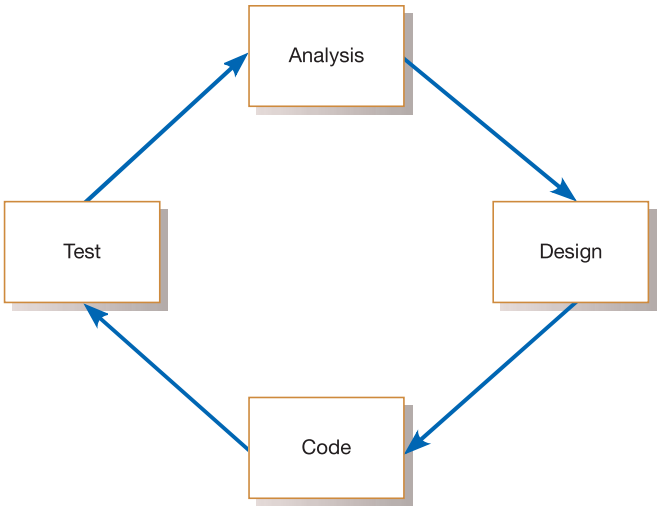


**FIGURE 1-8**
Analysis–design–code–test loop

ways, though, the SDLC is fiction. Although almost all systems development projects adhere to some type of life cycle, the exact location of activities and the specific sequencing of steps can vary greatly from one project to the next. Current practice combines the activities traditionally thought of as belonging to analysis, design, and implementation into a single process. Instead of systems requirements being produced in analysis, systems specifications being created in design, and coding and testing being done at the beginning of implementation, current practice combines all of these activities into a single analysis–design–code–test process (Figure 1-8). These activities are the heart of systems development, as we suggest in Figure 1-9. This combination of activities is typical of current practices in Agile Methodologies. A well-known instance of one of the Agile Methodologies is eXtreme Programming, although there are other variations. We will introduce you to Agile Methodologies and eXtreme Programming, but first it is important that you learn about the problems with the traditional SDLC. You will read about these problems next. Then you will read about CASE tools, Agile Methodologies, and eXtreme Programming.

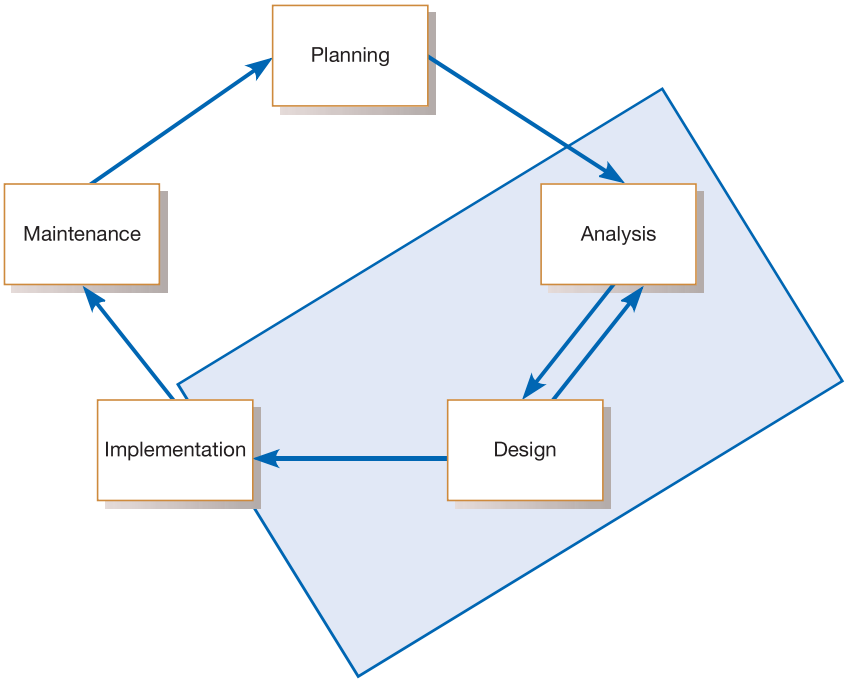


**FIGURE 1-9**
Heart of systems development

## The Traditional Waterfall SDLC

There are several criticisms of the traditional life-cycle approach to systems development; one relates to the way the life cycle is organized. To better understand these criticisms, it is best to see the form in which the life cycle has traditionally been portrayed, the so-called waterfall (Figure 1-10). Note how the flow of the project begins in the planning phase and from there runs "downhill" to each subsequent phase, just like a stream that runs off a cliff. Although the original developer of the waterfall model, W. W. Royce, called for feedback between phases in the waterfall, this feedback came to be ignored in implementation (Martin, 1999). It became too tempting to ignore the need for feedback and to treat each phase as complete unto itself, never to be revisited once finished.

Traditionally, one phase ended and another began once a milestone had been reached. The milestone usually took the form of some deliverable or prespecified output from the phase. For example, the design deliverable is the set of detailed physical design specifications. Once the milestone had been reached and the new phase initiated, it became difficult to go back. Even though business conditions continued to change during the development process and analysts were pressured by users and others to alter the design to match changing conditions, it was necessary for the analysts to freeze the design at a particular point and go forward. The enormous amount of effort and time necessary to implement a specific design meant that it would be very expensive to make changes in a system once it was developed. The traditional waterfall life cycle, then, had the property of locking users into requirements that had been previously determined, even though those requirements might have changed.

Yet another criticism of the traditional waterfall SDLC is that the role of system users or customers was narrowly defined (Kay, 2002). User roles were often relegated to the requirements determination or analysis phases of the project, where it was assumed that all of the requirements could be specified in advance. Such an assumption, coupled with limited user involvement, reinforced the tendency of the waterfall model to lock in requirements too early, even after business conditions had changed.

In addition, under the traditional waterfall approach, nebulous and intangible processes such as analysis and design are given hard-and-fast dates for completion,

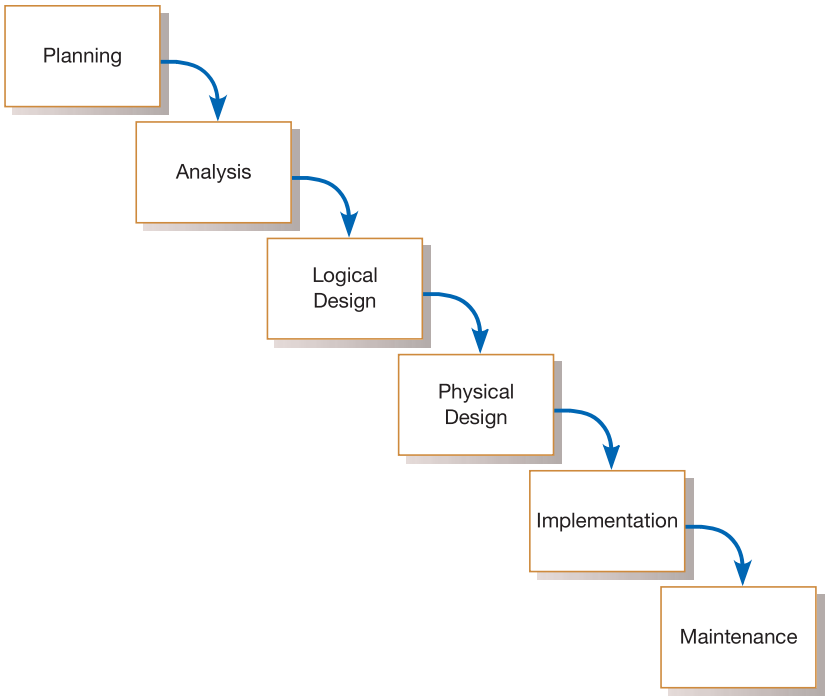


**FIGURE 1-10**
Traditional waterfall SDLC

and success is overwhelmingly measured by whether those dates are met. The focus on milestone deadlines, instead of on obtaining and interpreting feedback from the development process, leads to too little focus on doing good analysis and design. The focus on deadlines leads to systems that do not match users' needs and that require extensive maintenance, unnecessarily increasing development costs. Finding and fixing a software problem after the delivery of the system is often far more expensive than finding and fixing it during analysis and design (Griss, 2003). The result of focusing on deadlines rather than on good practice is unnecessary rework and maintenance effort, both of which are expensive. According to some estimates, maintenance costs account for 40 to 70 percent of systems development costs (Dorfman and Thayer, 1997). Given these problems, people working in systems development began to look for better ways to conduct systems analysis and design.

## DIFFERENT APPROACHES TO IMPROVING DEVELOPMENT

In the continuing effort to improve the systems analysis and design process, several different approaches have been developed. We will describe the more important approaches in more detail in later chapters. Attempts to make systems development less of an art and more of a science are usually referred to as *systems engineering* or *software engineering*. As the names indicate, rigorous engineering techniques have been applied to systems development. One manifestation of the engineering approach is CASE tools, which you will read about next.

### CASE Tools

**Computer-aided software engineering (CASE) tools**
Software tools that provide automated support for some portion of the systems development process.

Other efforts to improve the systems development process have taken advantage of the benefits offered by computing technology itself. The result has been the creation and fairly widespread use of **computer-aided software engineering (CASE) tools**. CASE tools have been developed for internal use and for sale by several leading firms, but the best known is the series of Rational tools made by IBM.

CASE tools are used to support a wide variety of SDLC activities. CASE tools can be used to help in multiple phases of the SDLC: project identification and selection, project initiation and planning, analysis, design, and implementation and maintenance. An integrated and standard database called a *repository* is the common method for providing product and tool integration, and has been a key factor in enabling CASE to more easily manage larger, more complex projects and to seamlessly integrate data across various tools and products. The idea of a central repository of information about a project is not new—the manual form of such a repository is called a *project dictionary* or *workbook*.

The general types of CASE tools are listed below:

- Diagramming tools enable system process, data, and control structures to be represented graphically.
- Computer display and report generators help prototype how systems "look and feel." Display (or form) and report generators make it easier for the systems analyst to identify data requirements and relationships.
- Analysis tools automatically check for incomplete, inconsistent, or incorrect specifications in diagrams, forms, and reports.
- A central repository enables the integrated storage of specifications, diagrams, reports, and project management information.
- Documentation generators produce technical and user documentation in standard formats.
- Code generators enable the automatic generation of program and database definition code directly from the design documents, diagrams, forms, and reports.

**TABLE 1-2 Examples of CASE Usage within the SDLC**

| SDLC Phase | Key Activities | CASE Tool Usage |
|---|---|---|
| Project identification and selection | Display and structure high-level organizational information | Diagramming and matrix tools to create and structure information |
| Project initiation and planning | Develop project scope and feasibility | Repository and documentation generators to develop project plans |
| Analysis | Determine and structure system requirements | Diagramming to create process, logic, and data models |
| Logical and physical design | Create new system designs | Form and report generators to prototype designs; analysis and documentation generators to define specifications |
| Implementation | Translate designs into an information system | Code generators and analysis, form and report generators to develop system; documentation generators to develop system and user documentation |
| Maintenance | Evolve information system | All tools are used (repeat life cycle) |

CASE helps programmers and analysts do their jobs more efficiently and more effectively by automating routine tasks. However, many organizations that use CASE tools do not use them to support all phases of the SDLC. Some organizations may extensively use the diagramming features but not the code generators. Table 1-2 summarizes how CASE is commonly used within each SDLC phase. There are a variety of reasons why organizations choose to adopt CASE partially or not use it at all. These reasons range from a lack of vision for applying CASE to all aspects of the SDLC, to the belief that CASE technology will fail to meet an organization's unique system development needs. In some organizations, CASE has been extremely successful, whereas in others it has not.

## AGILE METHODOLOGIES

Many approaches to systems analysis and design have been developed over the years. In February 2001, many of the proponents of these alternative approaches met in Utah and reached a consensus on several of the underlying principles their various approaches contained. This consensus turned into a document they called "The Agile Manifesto" (Table 1-3). According to Fowler (2003), the Agile Methodologies share three key principles: (1) a focus on adaptive rather than predictive methodologies, (2) a focus on people rather than roles, and (3) a focus on self-adaptive processes.

The Agile Methodologies group argues that software development methodologies adapted from engineering generally do not fit with real-world software development (Fowler, 2003). In engineering disciplines, such as civil engineering, requirements tend to be well understood. Once the creative and difficult work of design is completed, construction becomes very predictable. In addition, construction may account for as much as 90 percent of the total project effort. For software, on the other hand, requirements are rarely well understood, and they change continually during the lifetime of the project. Construction may account for as little as 15 percent of the total project effort, with design constituting as much as 50 percent. Applying techniques that work well for predictable, stable projects, such as bridge building, tend not to work well for fluid, design-heavy projects such as writing software, say the Agile Methodology proponents. What is needed are methodologies that embrace change and that are able to deal with a lack of predictability. One mechanism for dealing with a lack of predictability, which all Agile Methodologies share, is iterative development (Martin, 1999). Iterative development focuses on the frequent production of working versions of a system that have a subset of the total number of required features. Iterative development provides feedback to customers and developers alike.

The Agile Methodologies' focus on people is an emphasis on individuals rather than on the roles that people perform (Fowler, 2003). The roles that people fill, of

**TABLE 1-3 The Agile Manifesto**

**The Manifesto for Agile Software Development**

*Seventeen anarchists agree:*

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools.
- *Working software* over comprehensive documentation.
- *Customer collaboration* over contract negotiation.
- *Responding to change* over following a plan.

That is, while we value the items on the right, we value the items on the left more. We follow the following principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Businesspeople and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Continuous attention to technical excellence and good design enhances agility.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

—Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas (*www.agileAlliance.org*)

(*Source: http://agilemanifesto.org/* © 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice.)

systems analyst or tester or manager, are not as important as the individuals who fill those roles. Fowler argues that the application of engineering principles to systems development has resulted in a view of people as interchangeable units instead of a view of people as talented individuals, each bringing something unique to the development team.

The Agile Methodologies promote a self-adaptive software development process. As software is developed, the process used to develop it should be refined and improved. Development teams can do this through a review process, often associated with the completion of iterations. The implication is that, as processes are adapted, one would not expect to find a single monolithic methodology within a given corporation or enterprise. Instead, one would find many variations of the methodology, each of which reflects the particular talents and experience of the team using it.

Agile Methodologies are not for every project. Fowler (2003) recommends an agile or adaptive process if your project involves

- unpredictable or dynamic requirements,
- responsible and motivated developers, and
- customers who understand the process and will get involved.

**TABLE 1-4 Five Critical Factors That Distinguish Agile and Traditional Approaches to Systems Development**

| Factor | Agile Methods | Traditional Methods |
|---|---|---|
| Size | Well matched to small products and teams. Reliance on tacit knowledge limits scalability. | Methods evolved to handle large products and teams. Hard to tailor down to small projects. |
| Criticality | Untested on safety-critical products. Potential difficulties with simple design and lack of documentation. | Methods evolved to handle highly critical products. Hard to tailor down to products that are not critical. |
| Dynamism | Simple design and continuous refactoring are excellent for highly dynamic environments but a source of potentially expensive rework for highly stable environments. | Detailed plans and Big Design Up Front, excellent for highly stable environment but a source of expensive rework for highly dynamic environments. |
| Personnel | Requires continuous presence of a critical mass of scarce experts. Risky to use non-agile people. | Needs a critical mass of scarce experts during project definition but can work with fewer later in the project, unless the environment is highly dynamic. |
| Culture | Thrives in a culture where people feel comfortable and empowered by having many degrees of freedom (thriving on chaos). | Thrives in a culture where people feel comfortable and empowered by having their roles defined by clear practices and procedures (thriving on order). |

(*Source:* Boehm, Barry; Turner, Richard, *Balancing Agility and Discipline: A Guide for the Perplexed,* 1st Ed., © 2004. Reprinted and electronically reproduced by permission of Pearson Education, Inc. New York, NY.)

A more engineering-oriented, predictable process may be called for if the development team exceeds 100 people or if the project is operating under a fixed-price or fixed-scope contract. In fact, whether a systems development project is organized in terms of Agile or more traditional methodologies depends on many different considerations. If a project is considered to be high risk and highly complex, and has a development team made up of hundreds of people, then more traditional methods will apply. Less risky, smaller, and simpler development efforts lend themselves more to Agile methods. Other determining factors include organizational practice and standards, and the extent to which different parts of the system will be contracted out to others for development. Obviously, the larger the proportion of the system that will be outsourced, the more detailed the design specifications will need to be so that subcontractors can understand what is needed. Although not universally agreed upon, the key differences between these development approaches are listed in Table 1-4, which is based on work by Boehm and Turner (2004). These differences can be used to help determine which development approach would work best for a particular project.

Many different individual methodologies come under the umbrella of Agile Methodologies. Fowler (2003) lists the Crystal family of methodologies, Adaptive Software Development, Scrum, Feature Driven Development, and others as Agile Methodologies. Perhaps the best known of these methodologies, however, is eXtreme Programming, discussed next.

## eXtreme Programming

eXtreme Programming is an approach to software development put together by Beck and Andres (2004). It is distinguished by its short cycles, incremental planning approach, focus on automated tests written by programmers and customers to monitor the development process, and reliance on an evolutionary approach to development that lasts throughout the lifetime of the system. Key emphases of eXtreme Programming

are its use of two-person programming teams, described later, and having a customer on-site during the development process. The relevant parts of eXtreme Programming that relate to design specifications are (1) how planning, analysis, design, and construction are all fused into a single phase of activity; and (2) its unique way of capturing and presenting system requirements and design specifications. With eXtreme Programming, all phases of the life cycle converge into a series of activities based on the basic processes of coding, testing, listening, and designing.

Under this approach, coding and testing are intimately related parts of the same process. The programmers who write the code also develop the tests. The emphasis is on testing those things that can break or go wrong, not on testing everything. Code is tested very soon after it is written. The overall philosophy behind eXtreme Programming is that the code will be integrated into the system it is being developed for and tested within a few hours after it has been written. If all the tests run successfully, then development proceeds. If not, the code is reworked until the tests are successful.

Another part of eXtreme Programming that makes the code-and-test process work more smoothly is the practice of pair programming. All coding and testing is done by two people working together to write code and develop tests. Beck says that pair programming is not one person typing while the other one watches; rather, the two programmers work together on the problem they are trying to solve, exchanging information and insight and sharing skills. Compared to traditional coding practices, the advantages of pair programming include: (1) more (and better) communication among developers, (2) higher levels of productivity, (3) higher-quality code, and (4) reinforcement of the other practices in eXtreme Programming, such as the code-and-test discipline (Beck & Andres, 2004). Although the eXtreme Programming process has its advantages, just as with any other approach to systems development, it is not for everyone and is not applicable to every project.

## OBJECT-ORIENTED ANALYSIS AND DESIGN

**Object-oriented analysis and design (OOAD)**
Systems development methodologies and techniques based on objects rather than data or processes.

**Object**
A structure that encapsulates (or packages) attributes and methods that operate on those attributes. An object is an abstraction of a real-world thing in which data and processes are placed together to model the structure and behavior of the real-world object.

**Inheritance**
The property that occurs when entity types or object classes are arranged in a hierarchy and each entity type or object class assumes the attributes and methods of its ancestors, that is, those higher up in the hierarchy. Inheritance allows new but related classes to be derived from existing classes.

**Object class**
A logical grouping of objects that have the same (or similar) attributes and behaviors (methods).

There is no question that **object-oriented analysis and design (OOAD)** is becoming more and more popular (we elaborate on this approach later throughout the book). OOAD is often called the third approach to systems development, after the process-oriented and data-oriented approaches. The object-oriented approach combines data and processes (called *methods*) into single entities called **objects**. Objects usually correspond to the real things an information system deals with, such as customers, suppliers, contracts, and rental agreements. Putting data and processes together in one place recognizes the fact that there are a limited number of operations for any given data structure, and the object-oriented approach makes sense even though typical systems development keeps data and processes independent of each other. The goal of OOAD is to make systems elements more reusable, thus improving system quality and the productivity of systems analysis and design.

Another key idea behind object orientation is **inheritance**. Objects are organized into **object classes**, which are groups of objects sharing structural and behavioral characteristics. Inheritance allows the creation of new classes that share some of the characteristics of existing classes. For example, from a class of objects called "person," you can use inheritance to define another class of objects called "customer." Objects of the class "customer" would share certain characteristics with objects of the class "person": They would both have names, addresses, phone numbers, and so on. Because "person" is the more general class and "customer" is more specific, every customer is a person but not every person is a customer.

As you might expect, a computer programming language is required that can create and manipulate objects and classes of objects in order to create object-oriented information systems. Several object-oriented programming languages have been created (e.g., C++, Eiffel, and Java). In fact, object-oriented languages were developed

first, and object-oriented analysis and design techniques followed. Because OOAD is still relatively new, there is little consensus or standardization among the many OOAD techniques available. In general, the primary task of object-oriented analysis is identifying objects and defining their structure and behavior and their relationships. The primary tasks of object-oriented design are modeling the details of the objects' behavior and communication with other objects so that system requirements are met, and reexamining and redefining objects to better take advantage of inheritance and other benefits of object orientation.

The object-oriented approach to systems development shares the iterative development approach of the Agile Methodologies. Some say that the current focus on agility in systems development is nothing more than the mainstream acceptance of object-oriented approaches that have been germinating for years, so this similarity should come as no surprise (Fowler, 2003). One of the most popular realizations of the iterative approach for object-oriented development is the **Rational Unified Process (RUP)**, which is based on an iterative, incremental approach to systems development. RUP has four phases: inception, elaboration, construction, and transition (see Figure 1-11).

**Rational Unified Process (RUP)**
An object-oriented systems development methodology. RUP establishes four phases of development: inception, elaboration, construction, and transition. Each phase is organized into a number of separate iterations.

In the inception phase, analysts define the scope, determine the feasibility of the project, understand user requirements, and prepare a software development plan. In the elaboration phase, analysts detail user requirements and develop a baseline architecture. Analysis and design activities constitute the bulk of the elaboration phase. In the construction phase, the software is actually coded, tested, and documented. In the transition phase, the system is deployed, and the users are trained and supported. As is evident from Figure 1-11, the construction phase is generally the longest and the most resource intensive. The elaboration phase is also long, but less resource intensive. The transition phase is resource intensive but short. The inception phase is short and the least resource intensive. The areas of the rectangles in Figure 1-11 provide an estimate of the overall resources allocated to each phase.

Each phase can be further divided into iterations. The software is developed incrementally as a series of iterations. The inception phase will generally entail a single iteration. The scope and feasibility of the project is determined at this

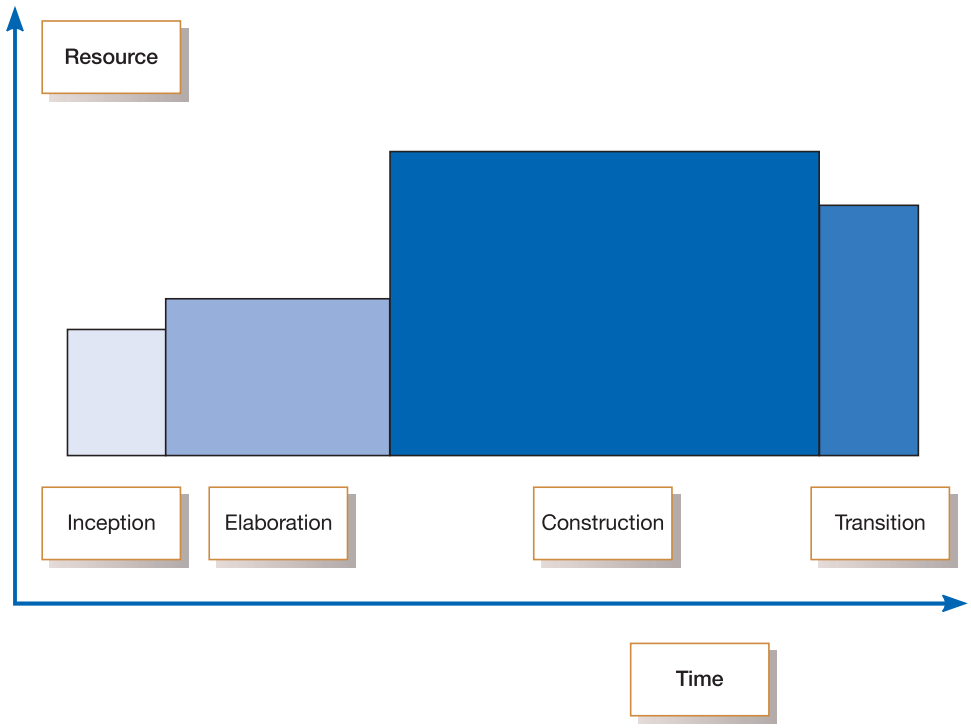


**FIGURE 1-11**
Phases of OOAD-based development