Module 2: Requirements Engineering and Modeling

Topics

---

1. Problem Analysis

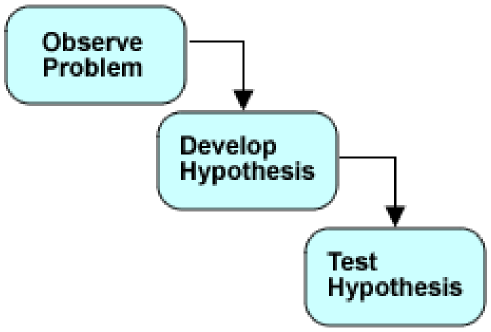
At the beginning of a project, the software engineer needs to define the problem before establishing the requirements for the software. In this section we will describe the scientific method that can be used to clarify the problem in order to develop a solution. In problem analysis, software engineers need to include all the stakeholders who have different perspectives on the proposed software. W e will discuss the role of stakeholders in requirements development and how their views of the proposed software will affect deriving the different categories of requirements.

Understanding the Problem

Defining requirements requires the software engineer to fully comprehend the problem before trying to solve it. As with other engineering disciplines, the software engineer can begin by following the scientific method to uncover the facts of the problem and to develop a hypothesis that can be used to solve the problem.

The scientific method is the standard way in which all scientists and engineers tackle problems, and the software engineer practices the scientific method in developing software. The solutions to the problems that the software engineer solves are the programs that he or she develops to execute on computers in a runtime environment.

There are three steps to the scientific method:



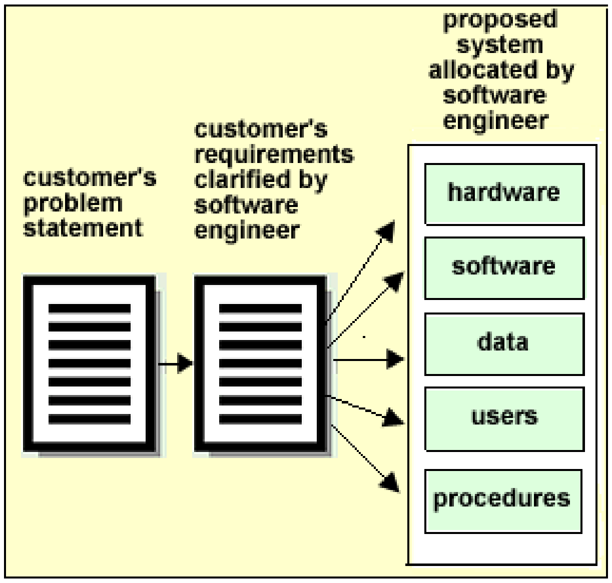
Analysis, the first task of software development, corresponds to the first step of the scientific method—observing the problem. The analysis step in software development involves partitioning the problem in order to understand its essential parts so that a solution can be proposed.

Analysis begins with a problem statement from the customer. Ideally, the problem statement will be an exhaustive laundry list of requirements that the customer has expressed for the solution to their problem. Customers, however, typically have difficulty expressing their requirements clearly. Clarifying customer requirements is one of the software engineer's most important tasks.

Here is an abbreviated example of a customer's problem statement:

- The self-service checkout system should ensure that store customers scan their products and pay their bill.
- All types of payment should be handled by the self-service checkout system.

Try to clarify this problem statement yourself and then look at ourclarified problem statement. The software engineer then allocates the clarified customer's requirements to a proposed system as indicated in the following illustration:


customer's problem statement
customer's requirements clarified by software engineer
proposed system allocated by software engineer
hardware
software
data
users
procedures

There are different categories of requirements that are needed for the varying components of a system. W e will discuss the different requirement categories in section 2 of this module. Once the allocation of requirements to system components is complete, the software engineer focuses on designing, building, and testing the software component of the system. As we already discussed in module 1, the system specification (SS) records the allocation of requirements to system components and forms the basis of all subsequent software engineering activities.

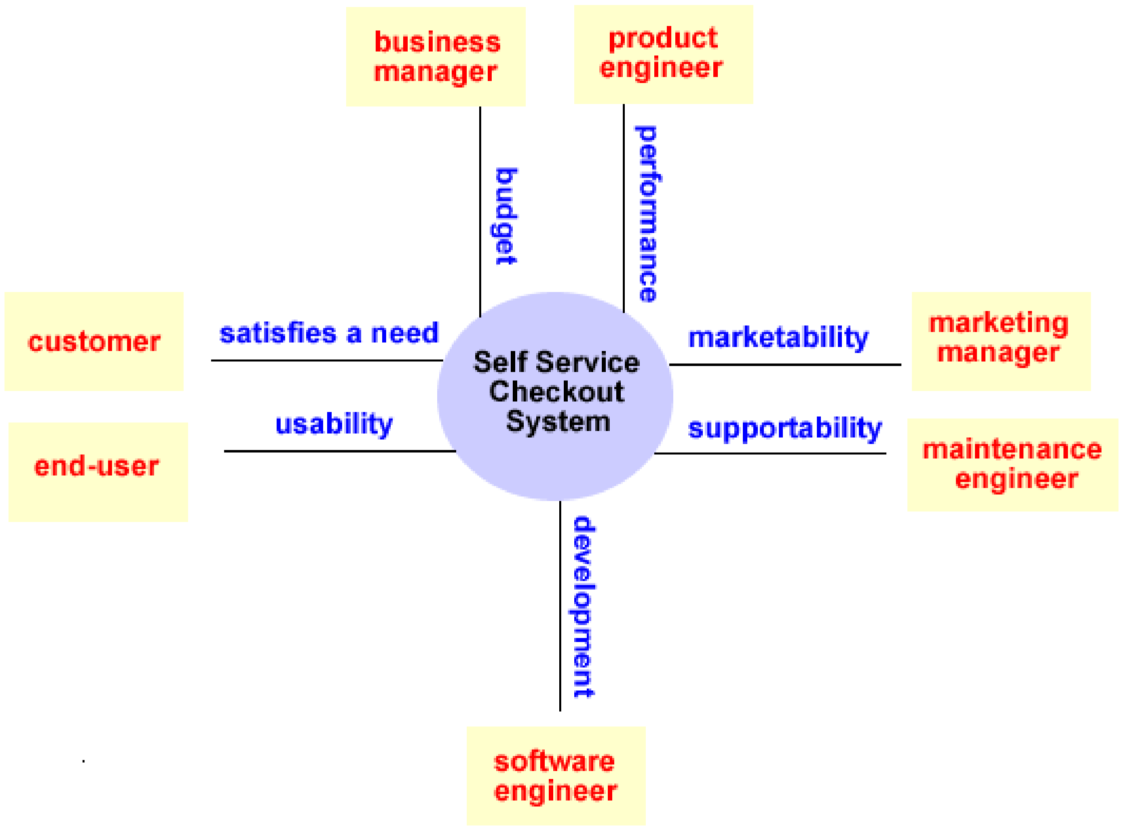
Software engineers use a model to verify that their comprehension of the software requirements is complete and coherent. These models typically are arrow-and-bubble diagrams, textual notations, or mathematical expressions, and are powerful tools for verifying completeness and communicating the required behavior of the software.

The software requirements analysis culminates with the composition of a requirement specifications document. For our discussion we will refer to the ANSI/IEEE Standard 830-1998 Recommended Practice for Software Requirements Specifications (SRS). The SRS is referred to as a *design-to* specification and is the definitive statement on the expected behavior of the software component of the system. W e will discuss the contents of various sections of the SRS throughout the remainder of this module.

Identifying the Stakeholders

Stakeholders play an important role in determining requirements for a computing system. Each stakeholder has a different viewpoint on what the software is supposed to do and the software engineer needs to involve them when deriving the requirements. A stakeholder can be defined as any individual, or group of individuals, "who will be affected by the system directly or indirectly" (Sommerville, 2004, p. 146). The effects of software can be described as any stakeholders who "request, pay for, select, specify, use, or receive the output generated by a software product" W eigers (2003, p. 29). In figure 2.1 below, we have outlined an example list of stakeholders and their potential interest in the product that will form the foundation for requirements elicitation.

Figure 2.1
Stakeholders and Their Viewpoints



The interest of each stakeholder will assist in defining categories of requirements for the proposed software. For example, the business manager may be concerned that the product is being developed with budget limitations, which restrict the number of features and the time allotted to develop the software. The end user is concerned with the ease of use of the software, which will affect the user interface requirements. The software engineer must consider each stakeholder's viewpoint when deriving a set of requirements. In section 2 we will discuss the different categories of requirements in more detail.

Pressman (2005, p. 102) categorizes stakeholders into two groups: customers and end users. W e have provided the outline of functions in table 2.1 that are used to distinguish the role of a customer and an end user. It is common for the customer to be the end user and to perform all the functions. The customers and end users have a significant impact on the technical aspects of the software. As we continue our discussion, we will use the terms customer and end user according to their functional role as outlined by Pressman.

Table 2.1
Role of Customer and End User in Software Development

Customer	End user
1) requests software	
2) defines overall business objectives	1) uses software to achieve business objectives
3) provides product requirements	2) defines operational details
4) coordinates funding	

Getting to Know the Business

In order to provide a solution to a customer's problem, the software engineer must become familiar with the customer's business and the objectives for the proposed software. Commonplace business problems can often be solved with commercial off-the-shelf products, but unique business problems require creative solutions. Understanding the nature of the customer's business will assist the software engineer in clarifying the requirements needed for the proposed software. For example, financial and medical institutions require secure, accurate, and responsive transaction processing software. The software engineer must research and become educated about the customer's business in order to form the foundation for the different categories of requirements that are needed to support the application domain.

2. The Software Requirement

To fully understand how to derive requirements from different perspectives at the software level, we need to define a requirement and discuss the various types of requirements. It is also important to understand that deriving complete and correct requirements is the driving force for the success of the subsequent development phases: design, code, and test.

Types of Requirements

Requirements can be classified into three basic types that exist for software. To read the definition of each type, move your mouse over the type.

- functional
- non-functional
- domain

Software engineers commonly use the acronym FURPS to describe these requirement categories: functionality, usability, reliability, performance, and supportability. In figure 2.2 we have extended the basic types of requirements for software to include data base, security, safety, and external interface. Additional software requirement categories can exist for a customer's problem in the areas of constraints and implementation (Grady, 1992).

Figure 2.2
Categories of Software Requirements

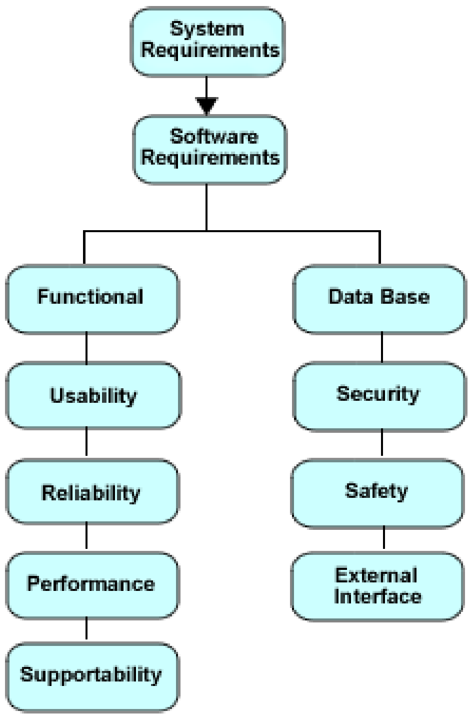


Figure 2.2 depicts that all the categories of software requirements are derived from the system requirements that form the basis for all subsequent development. As we mentioned in section 1, special consideration for the stakeholders' business objectives and their interests must exist in defining all the requirement categories for the proposed software product.

The primary focus of our requirements development discussion will be on deriving the functional requirements for software. A complete SRS will address all the categories that apply to the customer's problem. It is common for software engineers to create supplemental requirement documents and then reference them in the SRS. For example, if a system involves a safety-critical component, a separate specification containing only safety-related requirements may be compiled and used as a reference. After all the software requirements have been fully understood and elaborated for the business domain and the customer's problem, the analysis task is complete.

Requirement Definition

The Institute of Electrical and Electronics Engineers (IEEE) generically defines a software requirement as a condition or capability to which the system being built must conform to operate correctly.

A more refined definition of a software requirement is provided in the IEEE Standard 729 (1983) as:

- a software capability needed by the user to solve a problem or achieve an objective
- a software capability that must be met or processed by a system or system component to satisfy a contract, specification, standard, or other formally imposed documentation

Davis (1993) adds that software requirements are used to:

- define an object, function or state,
- limit or control actions associated with an object, function, or state
- define relationships among objects, functions, and states

Conflicting Requirements

W hen specifying requirements, there is a potential risk that one requirement may conflict with another in a different category. To avoid this conflict, it's necessary for the software engineer to cross-check requirements with those in other categories. For example, a functional requirement may specify that something should be done and the non-functional requirement may specify that it should *not*be done. Early identification of conflicting requirements will avoid rework and the introduction of software errors in later stages of the development life cycle.

Importance of Requirements

Requirements guide the planning, designing, coding, and testing of software, and therefore need to be correctly and completely defined by the software engineer. Correct and complete requirements will reduce misinterpretation that can cause errors in later stages of development. Latent errors in the requirements can have a negative impact in the subsequent phases of software development, in the following ways:

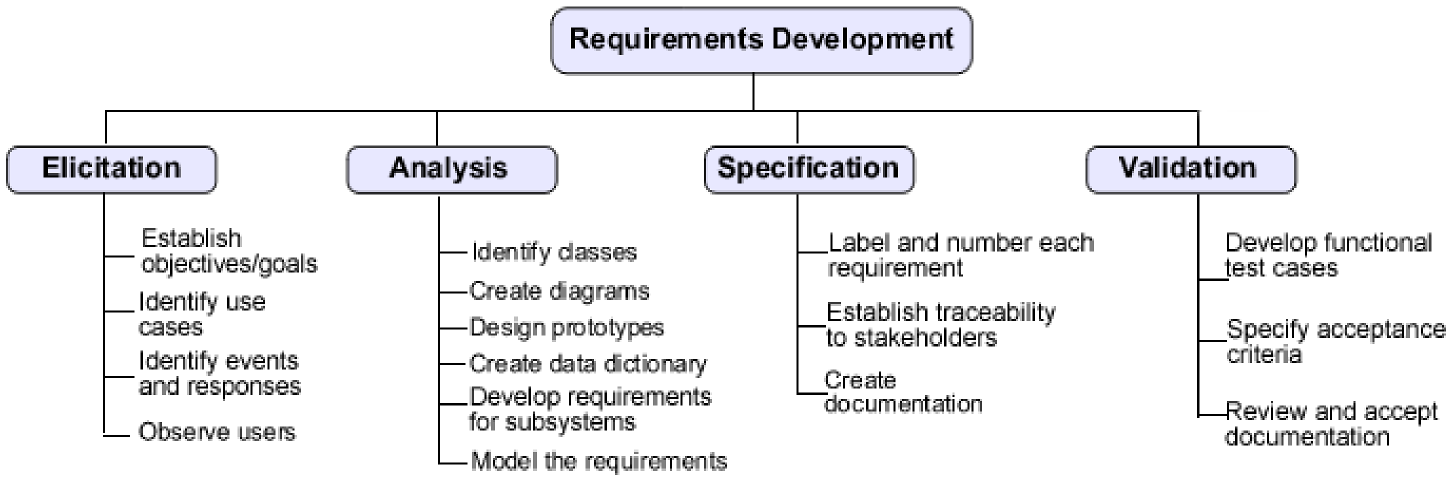
- expensive to fix or change
- does not satisfy the needs of the user(s)
- result in disagreements between the customer and the developer
- cannot derive test cases
- result in wrong system and customer dissatisfaction

Software engineers, customers, and end users should spend an adequate amount of time defining the requirements. Spending more time during requirements development has proven to be worth the investment. Blackburn et al (1996) reports in a study that improvement was seen in the development speed of companies when more time was spent during the software requirements stage. Companies that developed faster actually "spent less time and effort on average in all of the other stages of development and significantly less time in the final testing/integration stage." Blackburn et al concluded that the development speed was increased due to less rework involved in fixing errors discovered in the design, code, and test phases.

3. Requirements Development Model

In figure 2.3 we summarize the various phases and the activities that occur at each phase of the basic requirements development model (W iegers, 2003). In subsequent sections we will discuss several of the key activities contained in each phase.

Figure 2.3
Requirements Development Model


Requirements Development
Elicitation
Analysis
Specification
Validation
Establish objectives/goals
Identify use cases
Identify events and responses
Observe users
Identify classes
Create diagrams
Design prototypes
Create data dictionary
Develop requirements for subsystems
Model the requirements
Label and number each requirement
Establish traceability to stakeholders
Create documentation
Develop functional test cases
Specify acceptance criteria
Review and accept documentation

Requirements Elicitation

Requirements elicitation for a software product is the most difficult phase in development. Understanding the application domain, problem, and the needs of the customer and end user requires various means of oral and written communication to transfer this information among all parties. Any breakdown in this communication process can lead to ill-defined, missing, and incorrect requirements, which will result in the delivery of a low-quality product or even the wrong software product.

In section 1 we discussed the importance of understanding the problem, the stakeholders, and the business objectives. Requirements elicitation begins with the customer who initiates communication by contacting a software engineer with a business-related problem. The software engineer responds to the customer's problem by clarifying the needs by using effective means of communication in a written description.

Effective Communication

Communication among the software engineer, customer, and end user begins with a series of either informal or formal meetings, preferably face to face. The software engineer should be prepared with a list of well-formed questions before any meeting. The responses to these questions provided by the customer and end user will assist the software engineer in fully understanding what functions need to be satisfied by the proposed software. The following list outlines tips for effective communication among the software engineer, customer, and end user:

- listen carefully to the customers and end users of the proposed system
- keep the language simple and free of technical terms
- prepare well-formed questions in advance
- perform research and gain understanding of the application's domain
- draw diagrams or pictures to represent complex ideas
- take good notes
- use a facilitator, if necessary

The process of effective communication in translating the customer's and end user's needs into requirements can break down among all parties for a variety of reasons. The software engineer, customer, and end user often see a communication gap in the requirements phase, which has been commonly referred to as the "user and the developer" syndrome (Leffingwell et al, 2003, p. 92). The customer and end user will often know exactly what they want but have difficulty in articulating their needs in language that the software engineer understands. The software engineer must realize that the customer and end user are the domain experts. An effective communication approach must be established by the software engineer and be used during requirements elicitation.

Effective communication can sometimes be preempted by the different personalities among the involved parties. Conflicts may result from misunderstandings in the use of language, differing cultures, or personal agendas. The use of a facilitator can greatly mitigate conflicts to keep a positive and productive communication process flowing among all parties.

Use Case Model

Earlier in our discussion we stressed the importance of effective communication among the software engineer, customers, and end users in understanding the requirements of the proposed software. We also highlighted that conveying this information back to the customer and end user was important for directing subsequent phases of development. At this point in our discussion we will elaborate on a technique that can be used to specify and model the expected behavior for software. Software engineers refer to this behavioral model as the *use case model*. The use case model defines use cases and scenarios that can "easily capture a set of functional requirements" and "provide a vehicle for organizing the software requirements in an easy-to-manage way" (Bittner et al, 2003, p. 11). Following the use case model during requirements development can ease the transition from the analysis to the design phase.

About Use Cases

Our discussion on use cases will begin with a brief history. The use case was introduced as part of an object-oriented development methodology originated by Ivar Jacobson and described in his book called *Object-Oriented Software Engineering: A Use Case Driven Approach* (Addison-Wesley, 1992). Researchers Larry Constantine and others have extended this concept into a general technique for requirements analysis and user interface design. The Unified Modeling Language provides support for modeling a use case.

Leffingwell et al (2003, p. 149) defines a use case as "sequences of actions a system performs that yield an observable result of value to a particular actor." A sequence of actions refers to a "set of functions" or an "algorithmic procedure" that is initiated and used by an actor. The use case describes the input to the system and the response that is necessary to satisfy the function that produces the intended output.

A use case is best summarized as having the following characteristics (Leffingwell et al, 2003; Weigers, 2003):

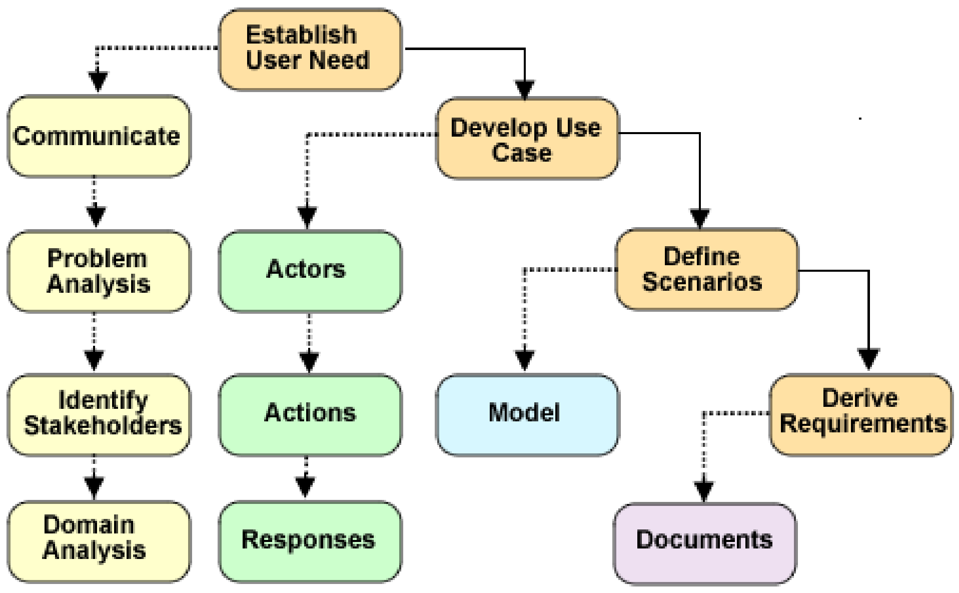
- provides the foundation for deriving scenarios in which an actor interacts with the system
- encompasses several tasks with a common goal
- achieves a specific goal or accomplishes a particular task or function
- describes an action in terminology relative to the actor's domain and not terminology relative to the computer
- focuses on the simplicity of the use or action being performed without regard to how it is implemented in the software or alternate ways of accomplishing the same task
- provides a means to derive detailed requirements for accomplishing the objective for each scenario by using a software application

When forming a use case, you should think of all the various external entities that interact with the system and the different ways in which they may possibly use the system. Specifying use cases is part of functional modeling of desired system behavior. By modeling the expected behavior with use cases, we can uncover required classes and associations that will be needed during the construction phase of the object model. We emphasize the necessary components for a complete use case in the following list:

- actor, who initiates an event or events
- specific interaction that takes place between this actor and the system
- expected response from the system

The steps to derive functional requirements following a use-case approach are summarized below in figure 2.4.

Figure 2.4
Use-case Approach to Derive Functional Requirements



In the list below we summarize the benefits derived from following a use case approach during requirements elicitation:

- determine system and software requirements
- determine actor profiles
- assist in identification of an initial set of classes and objects, at the system and software level
- assist in identification of software subsystems according to required functionality

W hen the software engineer meets with the potential end users of the desired system during requirements elicitation, the objective is to get answers to information-seeking questions (Leffingwell et al, 2003) to form the components of a use case. W e have provided a list of sample questions that the software engineer can use during meetings with the customers and end users:

- W hat is the expected behavior of the system?
- How is the user supposed to use the system?
- W hat types of services is the system supposed to provide?
- W hat is the expected result/output of the system?
- W ho will be using the system?

The software engineer composes a use case from the information and then uses the use case as the foundation for deriving one or more scenarios to carry out the functions.

Each use case scenario involves an actor, the system, and a single function to be performed. Once all the use cases are developed, they will assist you in identifying the data objects needed to create the proposed software system. W e will provide an example of a use case and model the possible scenarios with UML during our discussion on requirements analysis.

Requirements Analysis

During requirements elicitation, we discussed how to communicate and gather the requirements from the customers and end users for the proposed software. After we form a good idea of what the customers and end users want the software to do, we can expand our model of the expected behavior and develop profiles for the actors of the system. In this section we will discuss how to perform analysis that will result in detailed requirements that will form the bulk of the specifications document.

During requirements analysis, the software engineer elaborates on the information gathered during elicitation. At this stage the focus continues on the *whats*, rather than the *hows,* by defining the top-level objects and their attributes, their relationships with other objects, and the interfaces required with external entities. W e can begin to analyze and model the required flow and transformation of data between the identified entities (Pressman, 2005). There are several major data products that can result from requirements analysis; these are listed as:

- behavioral diagrams (use case scenario, data flow, and state transition)
- prototypes
- class identification
- data dictionary

Use Case and the Unified Modeling Language

The Unified Modeling Language (UML) provides a user model and notation for the specification of the different ways an end user will interact with the system, the use case. Models of scenarios are created from the use case to depict the expected behavior from the perspective of the actors. These scenarios assist the software engineer with elicitation and analysis of the requirements by describing the functionality that is required in the software. A functional requirement and the corresponding test case can be derived by the action that is modeled in the use case. A test case is used later in development to verify that a functional requirement has been met in the software product. W e will discuss scenario-based testing in module 4.

UML stresses the importance of textual documentation for each use case to accompany each scenario illustration. Following is a list of the items to be included in the use case documentation:

- unique name and number for the use case
- description of the role and purpose of the action
- description of the basic sequence of events
- description outlining any special requirements or conditions
- description stating the specific pre-conditions and post-conditions

A descriptive name is selected and assigned to each use case that best describes the function required by the software. Nomenclature for use cases is very important in software engineering, as it enhances documentation. W hen possible, similar functions should be categorized in one use case. A unique number or identifier is assigned to each use case to enhance the traceability to the requirement, the software component, and the test case. W e will discuss requirements traceability and its role in the development process later in this module. In module 5, we will discuss traceability through the development cycle as a project management activity.

Example Use Case

Let's take a look at a use case that we can define for the self-service checkout system. The use case for the customer in a retail store using the self-service checkout system to start a new purchase order is outlined in table 2.2.

Table 2.2
Use Case to Start a New Purchase Order

Use Case 1.0: Start new purchase order

A customer selects one or more items to purchase in a retail store. Once customers have completed selecting all the items, they will walk up to the checkout counter and start a new purchase order to prepare for scanning each item.

Precondition: The checkout system needs to be configured properly and have access to the local area network and store database. The system needs to be ready to accept a new purchase order.

Post-condition: The system is ready to scan items for a new purchase order.

Actor Profile: Customer shopping in a retail store who desires to purchase one or more items. A customer can be new to the system or be a frequent shopper familiar with the system. Frequent customers may have a discount card. Many customers will shop in the retail store.

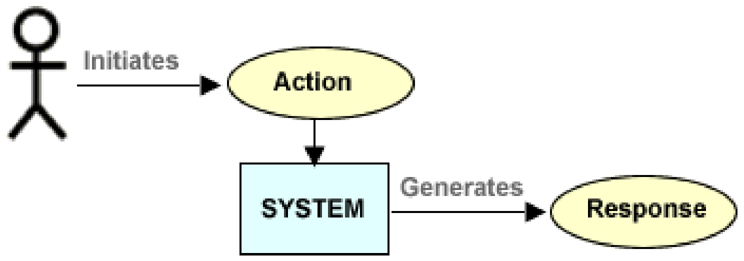
Sequence of events:

1. Customer walks up to an available self-service checkout.
2. Customer reads instructions and selects either Spanish or English language.
3. Customer selects the option to begin the order.
4. Customer scans frequent shopper discount card, if one exists.
5. Customer reads the directions for scanning an item.

User Scenarios

User scenarios, also referred to as user *stories*, describe a unique theme for interacting with the system. Scenarios are derived from the use case textual description. W hen creating scenarios, you select the actors that interact with the system and model different ways in which they will use the system to accomplish the required tasks. As we mentioned earlier, a user scenario consists of an actor, who initiates events, and an explanation of the interaction that takes place between this actor and the system. As a result of this interaction, the system will generate a response. Refer to figure 2.5 below for a representation of the various components of a user scenario.

Figure 2.5
Components of a User Scenario



It should be noted that an actor is any external entity that uses a system in a particular way to perform a desired function to accomplish a task. The actor can be a person, another software system, a hardware device, or an organization. For example, actors can be people who assume various roles (operator, customer, manager, system administrator) or an external entity (hardware device) that interacts with the system. The software engineer must have a good understanding of the characteristics of each actor or group of actors before composing a user scenario. These characteristics are documented and referred to as an actor profile.

For the self-service checkout system, we can write the following scenarios presented in table 2.3 from the use case presented in table 2.2.

Table 2.3
User Scenario for the Start of a New Purchase Order Use Case

User scenario 1: start the order

1. Customer selects either the English or Spanish language.
2. System displays the "Begin Order" option in the English or Spanish language.

User scenario 2: set up the order

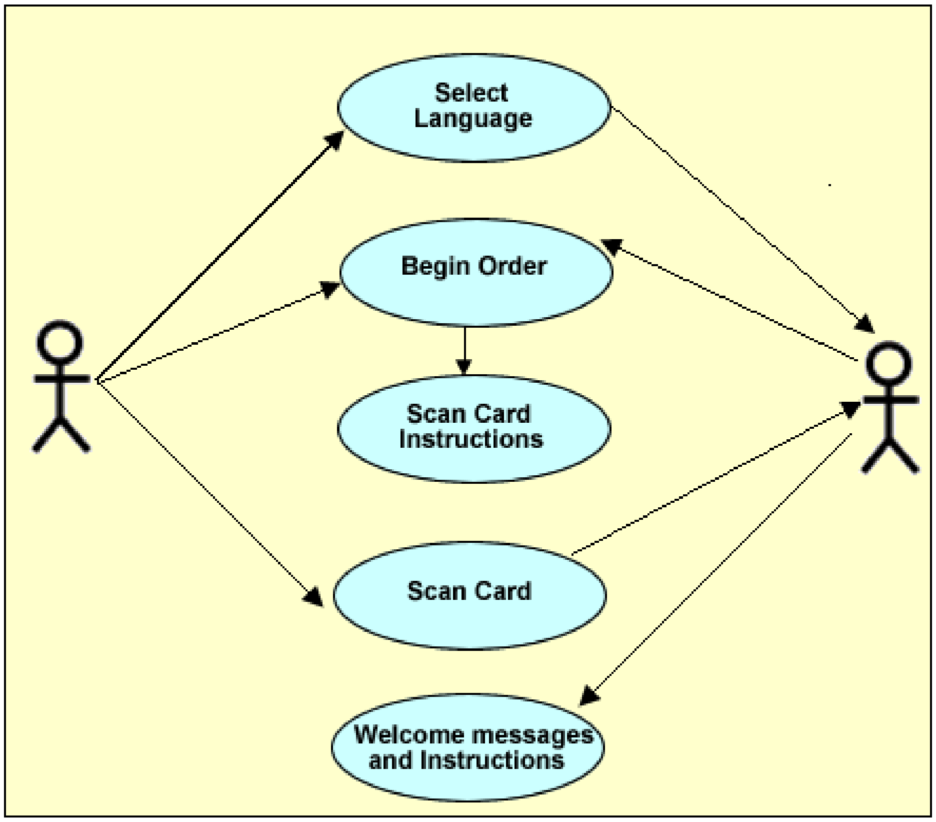
1. Customer selects the "Begin Order" option when ready to start the order.
2. System responds by displaying a message with instructions to scan frequent shopper discount card.

User scenario 3: scan discount card

1. Customer places the discount card over the scanner.
2. System responds by reading the barcode imprinted on the card.
3. System displays welcome message and instructions to scan an item.

Once the use case scenarios are written for the use case, we can model them with a use case diagram as shown in figure 2.6.

Figure 2.6
Start a New Purchase Order Use Case Diagram



Identification of Objects and Classes

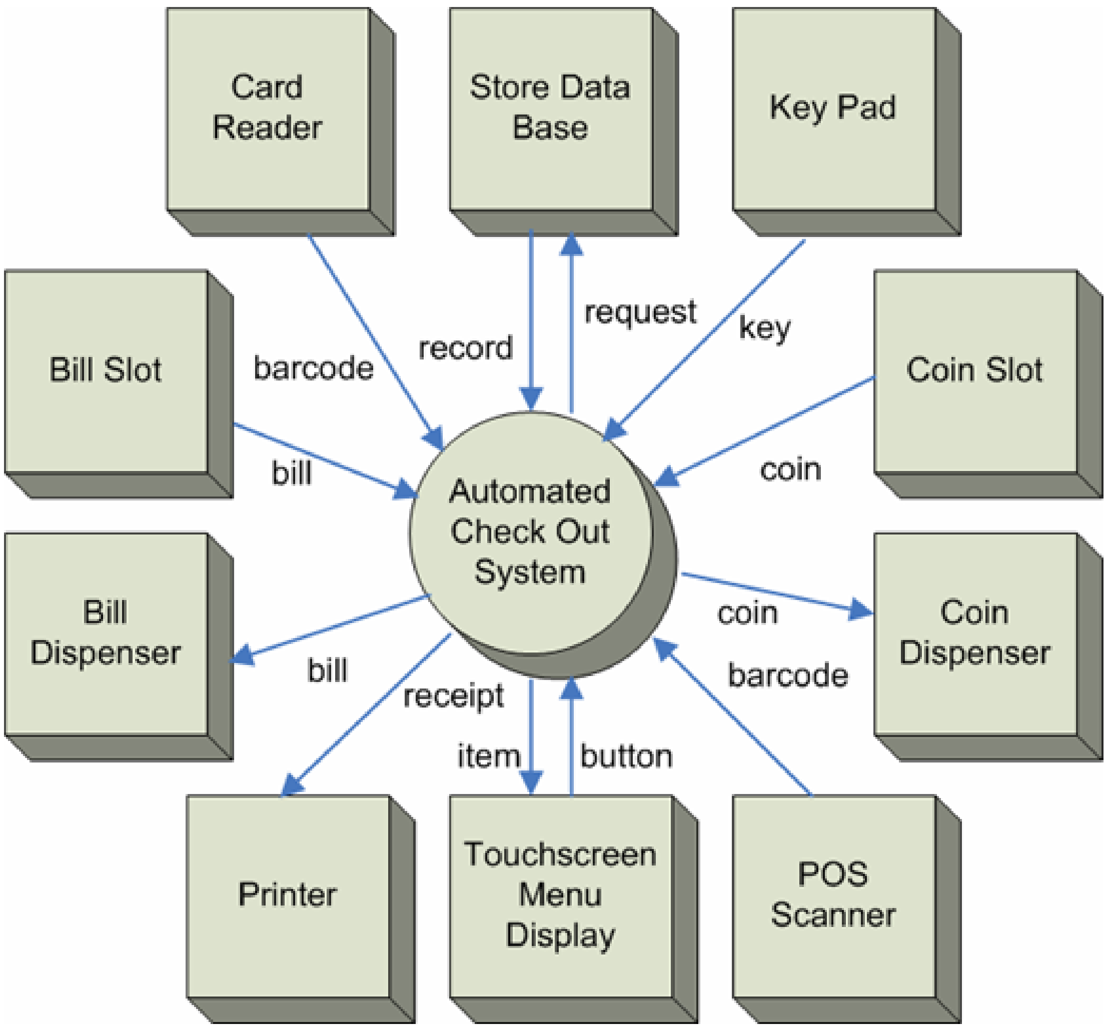
Data Objects

The use case can provide insight into the identification of objects that can be grouped by similarity to form classes for these objects. Objects are easily identifiable because the actors of a system manipulate objects when they use the software to perform a function to accomplish a task. Identifying the commonality among the objects will help software engineers to derive the base classes and establish attributes, needed functionality, and relationships. Software engineers model classes and their relationships with diagrams. UML is the common notation language being used in today's software industry for creating class diagrams.

Context Diagram

In module 1 we suggested the context diagram be used at the system level for specifying requirements. W e present the context diagram for the self-service checkout system again as a means of modeling the necessary flow of data items into and out of the software system.

Figure 2.7
Context Diagram for the Self-service Checkout System



The context diagram represented in figure 2.7 shows a sketch of the data items that are to be manipulated by the system's external entities and actors. At this point, the view of the software system is considered a black box and is represented by the circle. During the design phase, we will refine the software system to the process level to extend our view to the functional level, which will yield the subsystems and components.

If an object-oriented development approach is to be followed, each external entity and data item is a good candidate for an object. For example, we can easily derive several objects for the self-service checkout system by following the directional arrows that flow in and out of the external entities. Additionally, we can imagine objects for the external entities that represent the hardware devices.

Class Diagrams

Once the potential objects are identified, they can be grouped by similarity and used to determine base classes with a set of attributes. A class describes a group of objects with similar properties or attributes, common behaviors, relationships to other objects, and semantics. For example, think of a person as an object. Every person has the same set of common attributes, including a name, gender, height, weight, eye color, nationality, fingerprint, and social security number. These attributes are used to identify an individual and distinguish one person from another. A base class can be designed with these attributes and the class can be used to create instances of different individuals with a unique set of characteristics. Class instances are also referred to as objects. An object is defined as a concept, idea, or thing with crisp boundaries and meaning for the problem at hand.

Once the software engineer establishes all the base classes, the relationships between the classes are modeled using a class diagram. A class diagram is a schema, pattern, or template used for describing the class. Each class description contains a name, a set of attributes, and a set of operations.

Bennett et al (2001, p. 49) outlines how creating a UML class diagram can achieve several goals in modeling expected behavior. The class diagram can be used to:

- document classes within a system or subsystem
- describe association, generalization, and aggregation between classes
- show characteristics of the class
- show the operations and interfaces of objects
- direct activities throughout the development process

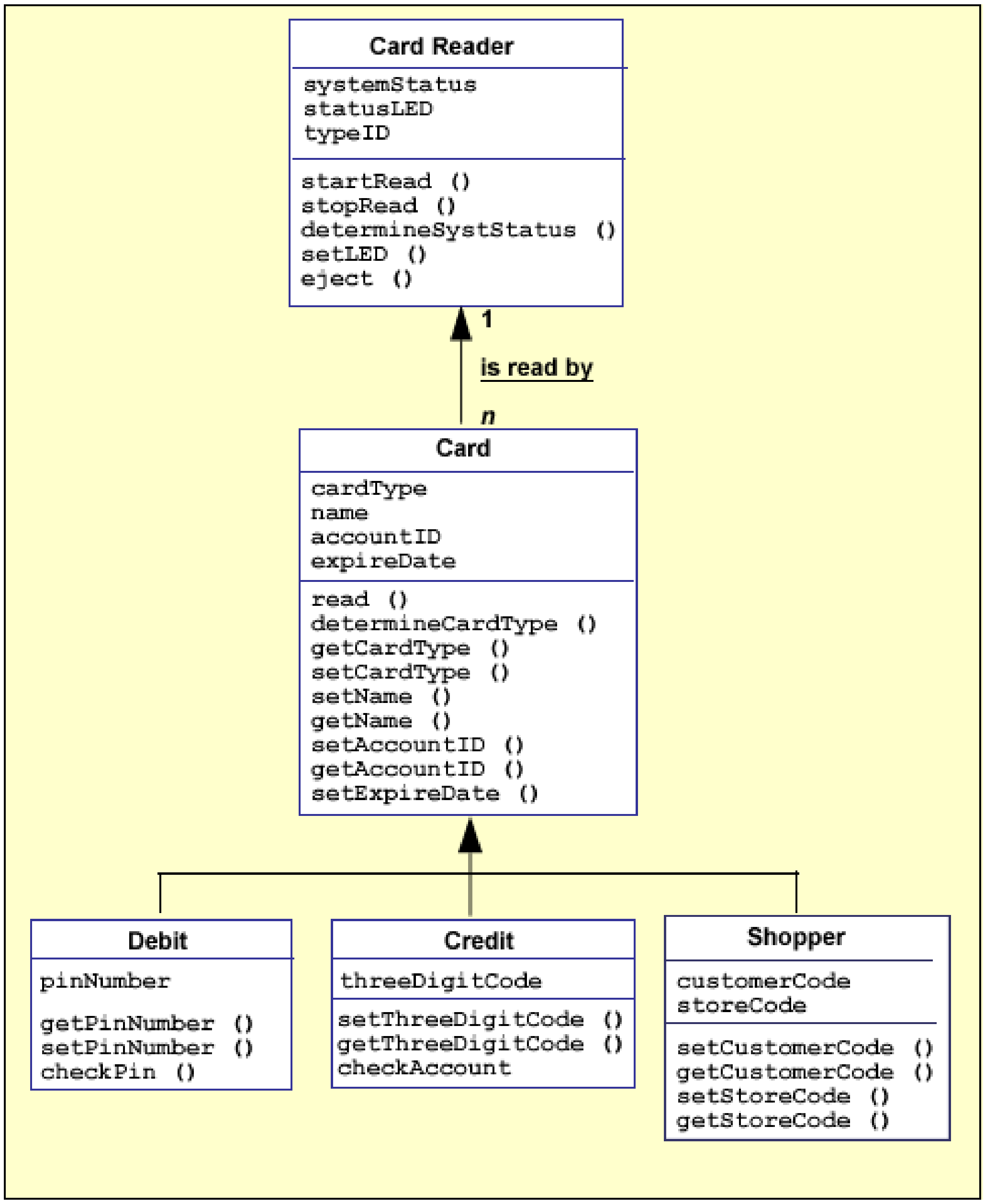
Class instances or objects collaborate with other objects through message passing. An association between two classes is described when a class instance or object passes a message to another object and that message is received by the recipient object. When drawing a class diagram, a connecting line is used to depict the association. The association is assigned a descriptive name. Software engineers can further define the association existing between the two class instances by adding its multiplicity, which indicates the possible number of instances for the class. In table 2.4 we summarize the indicators that can be placed at each end of the line in the class diagram to show multiplicity.

Table 2.4
Multiplicity Indicators for Class Diagrams

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n*, where *n* > 1
0..n	Zero to *n*, where *n* > 1
1..n	One to *n*, where *n* > 1

Refer to figure 2.8 for a class diagram that depicts the card reader and the card objects that are necessary for the payment transaction of the self-service checkout system.

Figure 2.8
Example Class Diagram



Data Dictionary

We will mention the data dictionary during requirements analysis because it can be used to store logical representations of data objects. It is important to adopt a notation for a data dictionary that provides for unambiguous and accurate descriptions of the data. Such notations facilitate interpretation and also allow for the use of CASE tools to assist in developing and maintaining dictionaries.

Modern systems may require extremely large data dictionaries, which may be practical only when a CASE tool is available to reduce the human effort and manage the complexity. Such tools, however, are useful for more than simple management of the data objects. If a formalized notation is used to describe the data, the tool can perform automated analyses of the data to ensure consistency across the system. Such analyses would be quite difficult if developers had to perform them by themselves.

We will extend our discussion on the data dictionary in module 3 when we look at the functional analysis model.

Requirement Specification

Software requirements are expressed in a simple and concise language and compiled into a formal document. The specification document should be easy for all the stakeholders to understand.

A simple set of guidelines (IEEE, 1998) is provided to assist in writing the software requirements specification. Each requirement should:

- consist of one sentence
- be free of technical jargon, abbreviations, and acronyms
- contain no conditional logic (if A then B)
- use active voice
- contain no references
- be grammatically correct

A software requirement can be composed by following either the "shall" or "shall not" format. Using the word *shall* in the requirement indicates that the requirement is mandatory. Using the self-service checkout system as an example, a requirement for processing a debit card payment transaction is a four-digit number. Here are two different and acceptable ways to write this requirement.

The software ***shall*** *require a four-digit code to be entered for a debit card transaction.*

The software ***shall not*** *process a debit card transaction without a four-digit code.*

The preferable way to write a software requirement is to use the "shall" format because the positive approach simplifies writing the software test case. The Boolean "not" can be somewhat tricky to code and test. The positive approach is commonly used in preparing the formal requirement document; however, you should be aware that both ways are acceptable to software engineers.

Other words, such as *should, will,* and *must,* can be used (Sommerville and Sawyer, 1997) if they are consistent with the meanings provided in table 2.5.

Table 2.5
Words and Their Meanings Used for Writing Software Requirements

Word	Meaning
shall	mandatory
should	desirable, but not mandatory
will	something that will be externally provided
must	best avoided; a synonym for "shall"

Writing a good set of requirements can be a challenging task because many software engineers find it difficult to write clear and concise natural language and have a tendency to write requirements in an unnecessarily verbose way. Table 2.6 contains a list of attributes of a well-written software requirement, and the do's and don'ts while writing the specifications document.

Table 2.6
Summary of Attributes for a Software Requirement

Attribute	Description	Do	Don't
correct	meets a user need	communicate with customer and user	add unnecessary features
unambiguous	no interpretation variations	choose simple words	use technical jargon
complete	include everything	be precise	embellish
verifiable	provide method to verify	write a test case	use words like *"usually," "often," "user friendly"*
consistent	no conflicts with other requirements	cross check with other categories	use the word *"don't"*

modifiable	easy to reference	provide index or table of contents	
traced	provide origin of each requirement	provide a traceability matrix	
traceable	identify each requirement	provide a unique number for each requirement	
design independent	no design constraints		imply a specific software architecture or algorithm
annotated	categories of requirements	group requirement by function or task	
concise	overall length and understandability	keep it short and as simple as possible	
organized	readability	use meaningful titles and category names	
understandable by the customer	meaningful content to both customer and developer	use simple language	use technical terms, unless simply defined

In table 2.7 we have provided a set of requirements for the start a new purchase order use case described in table 2.2 for our self-service checkout system.

Table 2.7
Requirements for the Start a New Purchase Order Use Case

Requirement 1.0: Start a new purchase order

1.1 The system shall display a message with an option to select the English or Spanish language.

1.1.1 If the English language option is selected, the system shall select the English language to display text.

1.1.2 The Spanish language option is selected, the system shall select the Spanish language to display text.

1.2 The system shall display an option to activate a new purchase order.

1.2.1 The option to activate the purchase order is detected, the system shall respond by displaying instructions to insert a frequent shopper discount card.

1.3 The system shall display an option to cancel the current session.

1.3.1 The cancel option is detected, the system shall respond by setting up for a new purchase order.

1.4 The system shall detect a frequent shopper discount card.

1.5 The system shall read an imprinted barcode from the frequent shopper discount card.

1.5.1 An error occurs while reading the barcode from a frequent shopper discount card, the system shall eject the card and display an error message.

1.5.2 The barcode is read without an error from the frequent shopper discount card, the system shall request the customer information from the store database.

1.6 The system shall use the barcode on the frequent shopper discount card to query the database for customer information.

1.6.1 The customer information is successfully queried from the database, the system shall display a welcome message with the customer's name.

1.6.2 The customer information was not successfully queried from the database, the system shall return a message "customer not located in database."

1.7 The system shall display instructions for scanning an item for purchase.

Traceability of Requirements

Each software requirement should be unique and traceable to the corresponding system requirement, use case, and software requirement. Traceability can be accomplished by creating atraceability matrix.

W e will extend the column headings in the traceability matrix as we discuss the various phases of development. The traceability matrix should be included as an appendix in the specifications document, updated at each phase in development, and included with each document created in that phase.

Specifications Document

The Software Requirements Specifications (SRS) document is used as a communication tool and serves as a contract between the customer and the software engineers. The SRS explains to the customer what the software engineer's perception is and what needs to be done in order to accomplish building the correct system. The requirements are outlined and categorized in simple language. In addition to satisfying communication needs, the SRS is the basis for integrated system testing and verification activities, and it also helps control the evolution of the software system.

The SRS is a deliverable document that is reviewed and accepted by the customer before design of the software actually begins. The SRS is developed after the system specifications and the system design documents are completed and accepted by the customer. The SRS must be complete, concise, and accurate to direct the subsequent phases of development. Roger Pressman (2005) proposes following a "use case driven" design. In this module, we discussed how to develop a use case and model scenarios with UML as a tool for requirements elicitation and analysis.

Once the SRS is approved by the customers, it serves as the baseline for all software requirements. Any change or addition to the requirements needs to be recorded, approved, and traced. We will discuss managing software changes as an activity of project management in module 5.

IEEE Recommended Practice for Software Requirements Specifications

The Institute of Electrical and Electronics Engineers has provided some guidelines for writing a Software Requirements Specification (SRS) in the IEEE Recommended Practice for Software Requirements Specifications, IEEE Standard 830-1998. Annex A of the IEEE Recommended Practice for Software Requirements Specifications provides guidelines for writing the specific requirements for a complete SRS. The templates outlined in Annex A organize specific requirements for a proposed software solution by describing:

- system modes
- user classes
- objects
- features
- stimulus
- response
- functional hierarchy

Software engineers choose the best template, or combinations of templates, that best describe the requirements for the proposed software solution. Combinations of templates can also be used to organize requirements.

In module 1 and in section 3 of this module, we have discussed several diagrams that can be used to graphically represent requirements for the SRS. The use of diagrams is recommended in technical documents and all diagrams should be supplemented with a textual description.

Requirements Validation

The final phase of the requirements development model is validation. Validating the requirements is necessary to ensure that the requirements are complete and unambiguous to build a high-quality and correct product. Early in module 1 we discussed how software engineering emerged in the computer industry to address the legacy problems of poor quality and cost overruns and to reduce maintenance costs after deployment of software products. Many studies have been done by Boehm, Grady, Kelly, Sherif, and Hops (Weigers, 2003) that have supported that early discovery of errors during the requirements development phase will reduce the cost and time to correct them. Errors in requirements can lay dormant in software and may not be discovered until system testing or by the customer after delivery of the product. Correcting latent errors in requirements requires significant rework by the software engineers, which takes time and money. Software engineers can validate the requirements by developing a set of preliminary test cases and participating in a formal peer review or inspection. Time spent validating the requirements can actually shorten the delivery schedule because less rework will be required (Blackburn et al, 1996) during this phase.

A preliminary set of test cases can easily be developed when the use-case model is followed during requirements elicitation and analysis. Writing test cases during the requirements validation phase follows a scenario-based approach that will uncover ambiguities in the functional requirements. These test cases are extended during the test phase of development. We will discuss how to write scenario-based test cases in module 4.

Another requirements validation method commonly used by software engineers is to conduct and participate in a formal review or inspection. A review committee is formed from the stakeholders who represent varying perspectives and interests in the proposed software product. A small team consisting of six or fewer participants is formed. During the review, the software engineer presents the categories of requirements to the team for discussion. Any deficiency in a requirement identified by the participants is included in a summary portfolio that consists of one or more "requests for action." The software engineer addresses each request and takes corrective action to revise or clarify the requirement. The revisions to the requirements are resubmitted to the reviewers for final approval.

References

Bennett, S., J. Skelton, and K. Lunn.*Schaum's Outlines, UML*. Italy: L.E.G.O. Spa, Vincenza, 2001.

Blackburn, Joseph D., Gary D. Scudder, and Van Wassenhove.*Improving Speed and Productivity of Software Development: A Global Survey of Software Developers*. IEEE Transactions on Software Engineering 22(12): 875-885. 1996.

Brittner, Kurt, and Ian Spence. *Use Case Modeling*. Boston, MA: Pearson Education, Addison-Wesley, 2003.

Davis, Alan M. *Software requirements: Objects, functions, and states.*Englewood Cliffs, CA: Prentice Hall Professional, 1993.

Fowler, M., and K. Scott. *UML Distilled. Applying the Standard Object Modeling Language*. Reading, Mass.: Addison W esley, 1997.

Grady, Robert B. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

Institute of Electrical and Electronics Engineers. *IEEE standard glossary of software engineering terminology.* (ANSI/IEEE Standard 729-1983.) New York: IEEE, 1983.

Institute of Electrical and Electronics Engineers. Recommended Practice for Software Requirements Specifications, (ANSI IEEE Standard 830-1998.) New York: IEEE, 1998.

Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-oriented software engineering: A use case driven approach.* New York: Addison-W esley Professional, 1992.

Leffingwell, D., and D. W idrig. *Managing Software Requirements: A Use Case Approach, (second edition)*. Boston, MA: Pearson Education, Inc., 2003.

Pressman, Roger S. *Software engineering: A practitioner's approach, (sixth edition).* New York: McGraw-Hill, 2005.

Sommerville, Ian. *Software Engineering, (seventh edition)*. England: Pearson Education Addison W esley, 2004.

Sommerville, Ian, and Peter Sawyer. *Requirements engineering: A good practice guide*. Chichester, W est Sussex, England: W iley, 1997.

W eigers, K.E. *Software Requirements, (second edition)*. Redmond, W ashington: Microsoft Press, 2003.