Unit-II

Understanding Requirements:

Understanding the requirements of a problem is among the most difficult tasks that face a software engineer. When you first think about it, developing a clear understanding of requirements doesn't seem that hard.

Requirements Engineering.

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction where business need is defined, user scenarios are described, functions and features are delineated, and project constraints are identified. The specific needs that design and construction must address; the priorities that guide the order in which work is to be completed; and the information, functions, and behaviors that will have a profound impact on the resultant design.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: inception, elicitation, elaboration, negotiation, specification, validation, and management. It is important to note that some of these tasks occur in parallel and all are adapted to the needs of the project.

Inception. Most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope. All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.

At project inception,3 you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.Christel and Kang identify a number of problems that are encountered as elicitation occurs.

Problems of scope. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objective

Problems of understanding. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

Problems of volatility. The requirements change over time

Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information**.**

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system. Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user. The attributes of each analysis class are defined, and the services that are required by each class are identified. The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

Negotiation. It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs.

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification. In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

Validation. The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification5 to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the technical review. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic (unachievable)requirements.

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents
Revision History

1. Introduction
1.1 Purpose
1.2 Document Conventions
1.3 Intended Audience and Reading Suggestions
1.4 Project Scope
1.5 References

2. Overall Description
2.1 Product Perspective
2.2 Product Features
2.3 User Classes and Characteristics
2.4 Operating Environment
2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies

3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)

4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces

5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes

6. Other Requirements

Appendix A: Glossary
Appendix B: Analysis Models
Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted earlier in this sidebar.

Eliciting requirements

Requirements elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification. In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements.

Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines**.**

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal. To better understand the flow of events as they occur,

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD"concentrates on maximizing customer satisfaction from the software engineering process". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.QFD identifies three types of requirements.

Normal requirements .The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the customer voice table—that is reviewed with the customer and other stakeholders. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements.

Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases, provide a description of how the system will be used.

Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include.

A statement of need and feasibility.

• A bounded statement of scope for the system or product.
• A list of customers, users, and other stakeholders who participated in requirements elicitation.
• A description of the system's technical environment.
• A list of requirements (preferably organized by function) and the domain constraints that apply to each.
• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
• Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

Developing use cases

The first step in writing a use case is to define the set of “actors” that will be involved in the story. Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system.

It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, where as an actor represents a class of external entities that play just one role in the context of the use case.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.

A number of questions12 that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor’s goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor’s interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

we define four actors: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the SafeHome home security function). For the purposes of this example, we consider only the homeowner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC.

Use case: InitiateMonitoring

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to “set” the system, i.e., to turn on the alarm functions.

Scenario:
1. Homeowner: observes control panel
2. Homeowner: enters password
3. Homeowner: selects “stay” or “away”
4. Homeowner: observes read alarm light to indicate that SafeHome has been armed
Exceptions:
1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.
2. Password is incorrect (control panel beeps once): homeowner reenters correct password.
3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.
4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.
5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

FIGURE 5.1

SafeHome control panel

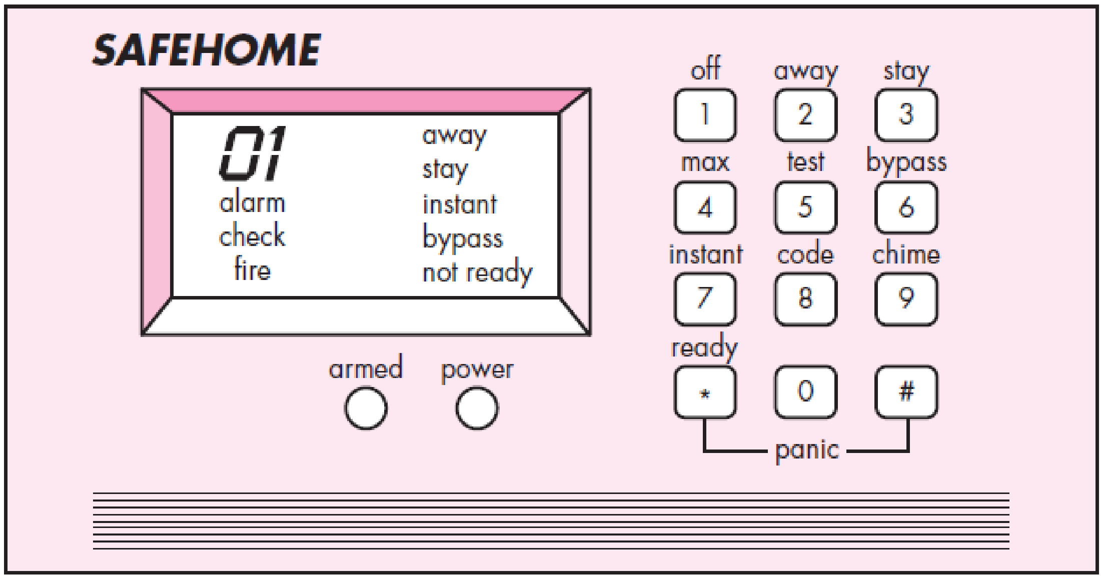
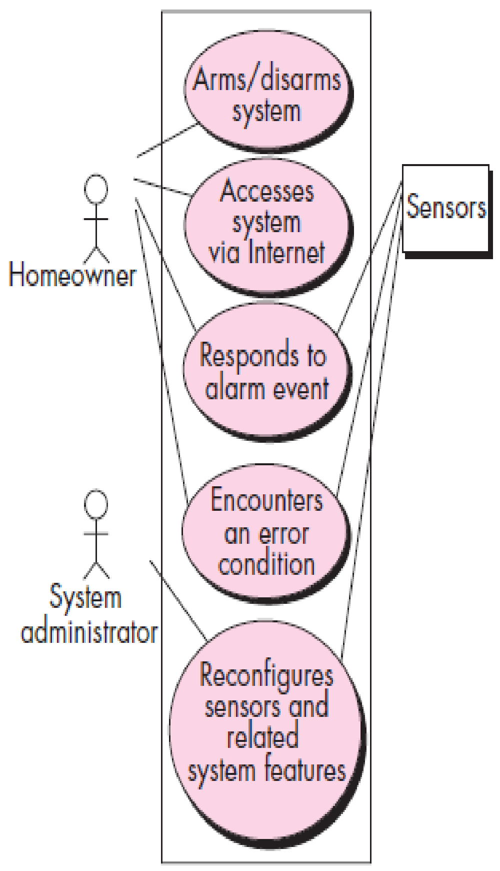


FIGURE 5.2

UML use case diagram for *SafeHome* home security function


Arms/disarms system
Accesses system via Internet
Responds to alarm event
Encounters an error condition
Reconfigures sensors and related system features
Homeowner
System administrator
Sensors

Negotiating requirements

The best negotiations strive for a "win-win" result.20 That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined.

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team)

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

Validating requirements

The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions.

Is each requirement consistent with the overall objectives for the system/product?

• Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
• Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
• Is each requirement bounded and unambiguous?
• Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
• Do any requirements conflict with other requirements?
• Is each requirement achievable in the technical environment that will house the system or product?
• Is each requirement testable, once implemented?
• Does the requirements model properly reflect the information, function, and behavior of the system to be built.
• Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
• Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Requirements Modeling:

Requirements Analysis

software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model actually a set of models—is the first technical representation of a system.

Requirements analysis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:
Scenario-based models of requirements from the point of view of various system "actors"
• Data models that depict the information domain for the problem
• Class-oriented models that represent object-oriented classes and the manner in which classes collaborate to achieve system requirements
• Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system
• Behavioral models that depict how the software behaves as a consequence of external "events".

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

scenario-based modeling—a technique that is growing increasingly popular throughout the software engineering community; data modeling—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and class modeling—a representation of the object-oriented classes and the resultant collaborations that allow a system to function. Flow-oriented models, behavioral models,

Overall Objectives and Philosophy

The requirements model must achieve three primary objectives:

(1) to describe what the customer requires,

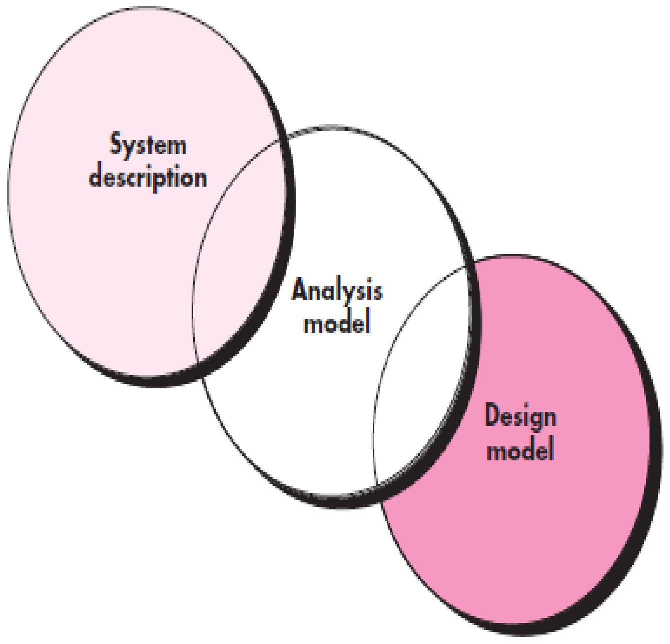
(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

FIGURE 6.1



The requirements model as a bridge between the system description and the design model

Analysis Rules of Thumb

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high. "Don't get bogged down in details" that try to explain how the system will work.

• Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.

• Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

• Minimize coupling throughout the system. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

• Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; acceptance tests.

• Keep the model as simple as it can be. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

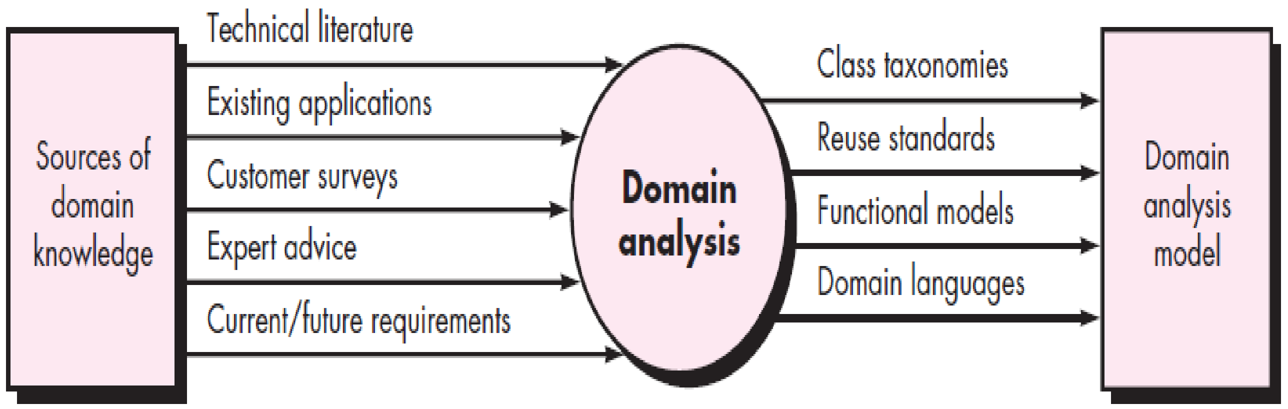
Domain Analysis

Analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

Domain analysis may be viewed as an umbrella activity for the software process. Domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master tool smith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst5 is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

Figure 6.2 illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

FIGURE 6.2 Input and output for domain analysis



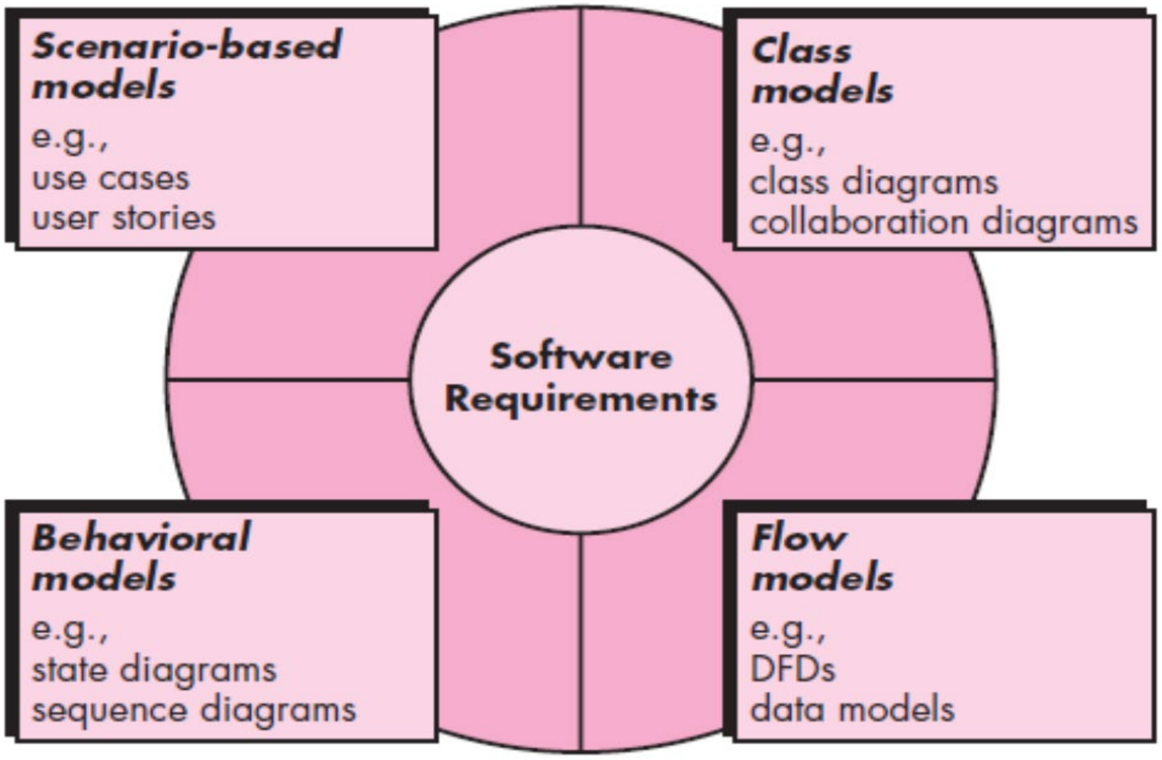
Requirements Modeling Approaches

One view of requirements modeling, called structured analysis, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element may differ from project to project.



Scenario-based modeling

Success of a computer-based system or product is measured in many ways; user satisfaction resides at the top of the list. If you understand how end users want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swimlane diagrams.

Creating a Preliminary Use Case

In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself. The following are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

(1) What to write about,
(2) How much to write about it,
(3) How detailed to make your description, and
(4) How to organize the description

What to write about?

The first two requirements engineering tasks—inception and elicitation—provide you with the information need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.

To begin developing a set of use cases, list the functions or activities performed by a specific actor. Obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.

The SafeHome home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) that are performed by the homeowner actor.

Select camera to view.

- Request thumbnails from all cameras.
- Display camera views in a PC window.
- Control pan and zoom for a specific camera.
- Selectively record camera output.
- Replay camera output.
- Access camera surveillance via the Internet.

As further conversations with the stakeholder (who plays the role of a homeowner)progress, the requirements gathering team develops use cases for each of the functions noted. A variation of a narrative use case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence.

Refining a Preliminary Use Case

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case. Therefore, each step in the primary scenario is evaluated by asking the following questions

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
- Is it possible that the actor will encounter some other behavior at this point? If so, what might it be?

Is it possible that the actor will encounter some other behavior at this point If so, what might it be.

The homeowner selects “pick a camera.”

The system displays the floor plan of the house.

Can the actor take some other action at this point

Is it possible that the actor will encounter some error condition at this point?

Is it possible that the actor will encounter some other behavior at this point?

Are there cases in which some “validation function” occurs during this use case?

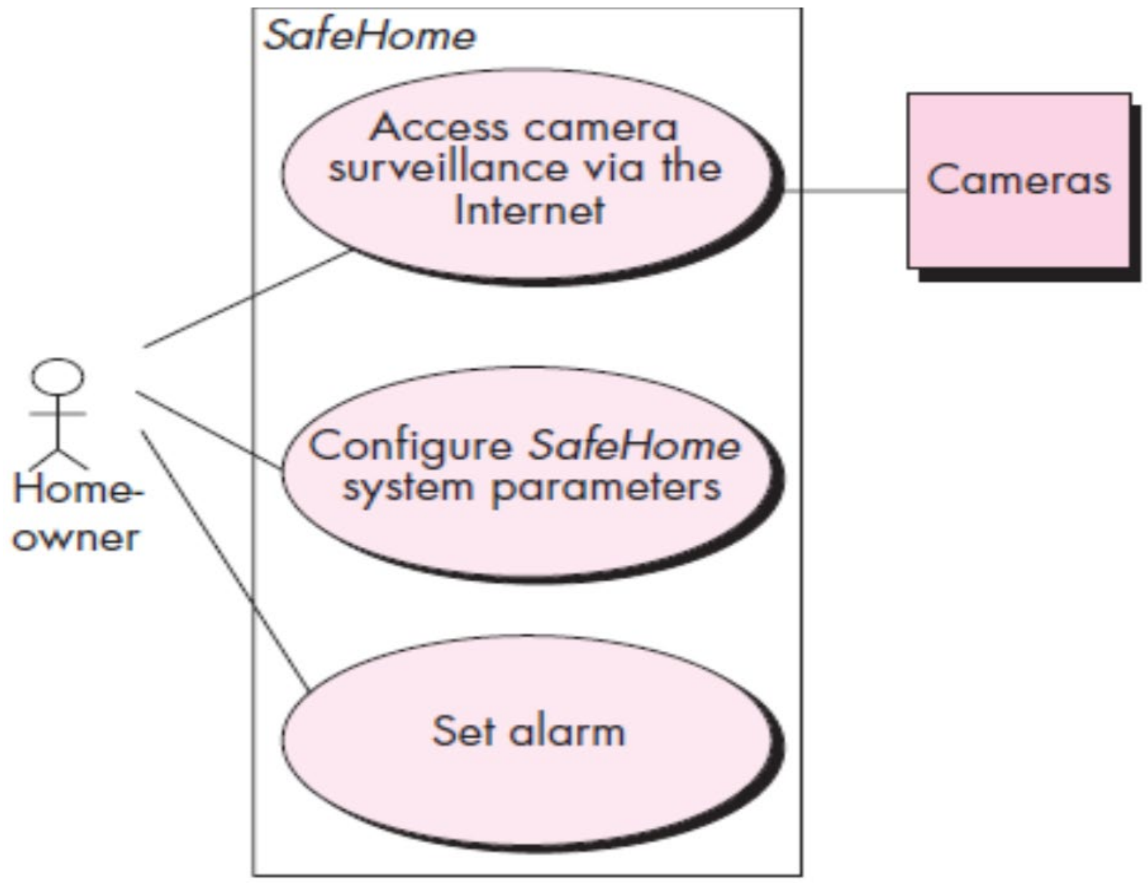
This implies that validation function is invoked and a potential error condition might occur.

Are there cases in which a supporting function (or actor) will fail to respond appropriately?
For example, a user action awaits a response but the function that is to respond times out.
Can poor system performance result in unexpected or improper user actions?

Writing a Formal Use Case

The goal in context identifies the overall scope of the use case. The precondition describes what is known to be true before the use case is initiated. The trigger identifies the event or condition that "gets the use case started". The scenario lists the specific actions that are required by the actor and the appropriate system responses. Exceptions identify the situations uncovered as the preliminary use case is refined. Additional headings may or may not be included and are reasonably self-explanatory.

Preliminary use-case diagram for the SafeHome system


SafeHome
Access camera surveillance via the Internet
Cameras
Configure SafeHome system parameters
Home-owner
Set alarm

UML model that supplement the Use-Case

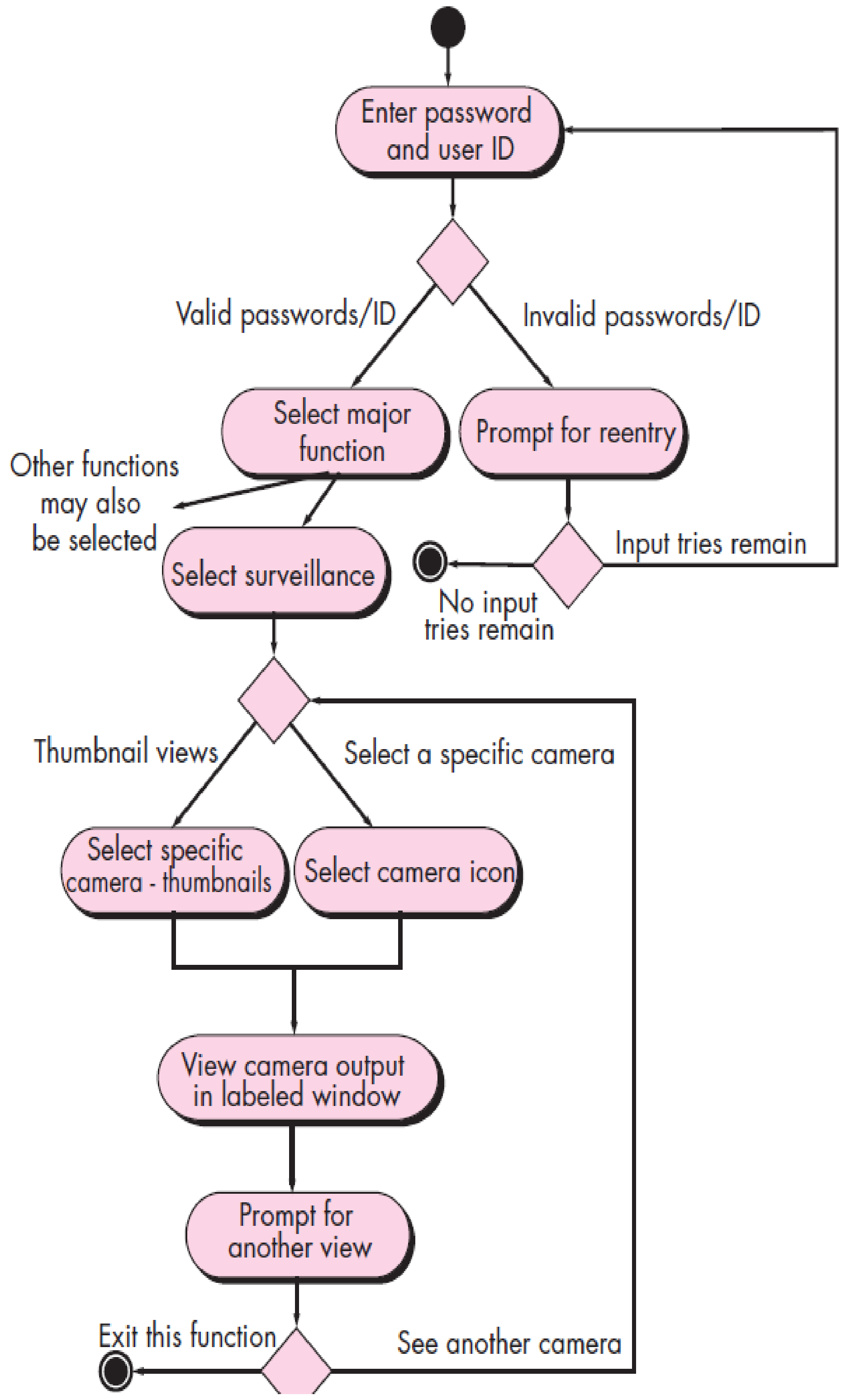
There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner.

Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function,arrows to represent flow through the system, decision diamonds to depict a branching decision and solid horizontal lines to indicate that parallel activities are occurring. An activity diagram for the ACS-DCV use case is shown in Figure 6.5. It should be noted that the activity diagram adds additional detail not directly mentioned by the use case.

FIGURE 6.5

Activity diagram for Access camera surveillance via the Internet—display camera views function.



Swimlane Diagrams

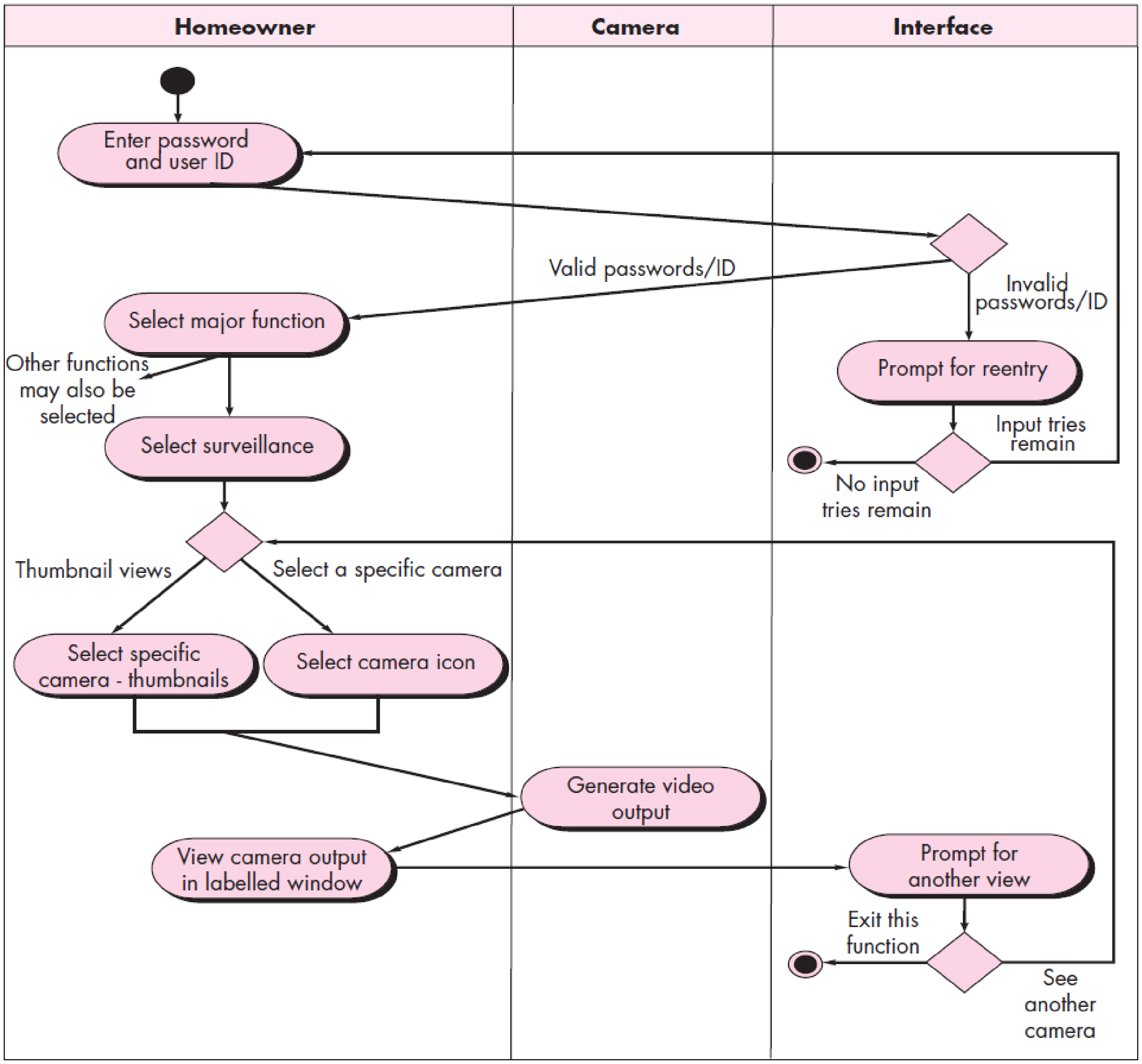
The UML swimlane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class (discussed later in this chapter) has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—Homeowner, Camera, and Interface—have direct or indirect responsibilities in the context of the activity diagram represented in Figure.

The activity diagram is rearranged so that activities associated with a particular analysis class fall inside the swimlane for that class The activity diagram notes two prompts that are the responsibility of the interface—"prompt for reentry" and "prompt for another view." These prompts and the decisions associated with them fall within the Interface swimlane.

Use cases, along with the activity and swimlane diagrams, are procedurally oriented. They represent the manner in which various actors invoke specific functions to meet the requirements of the system. But a procedural view of requirements represents only a single dimension of a system.

FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function



Data modeling concepts

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed and produced within an application.

Data Objects

A data object is a representation of composite information that must be understood by software. By composite information, Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g.,accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to

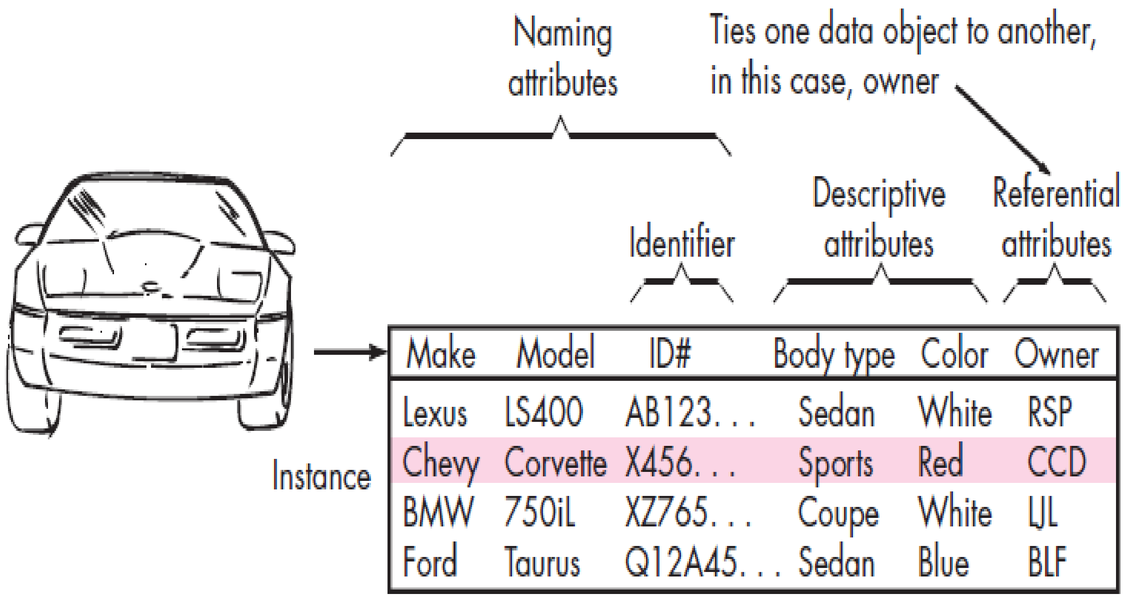
(1) Name an instance of the data object,

(2) Describe the instance, or

(3) Make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context.

FIGURE 6.7 Tabular representation of data objects



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Relationship

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. Establish a set of object /relationship pairs that define the relevant relationships. For example

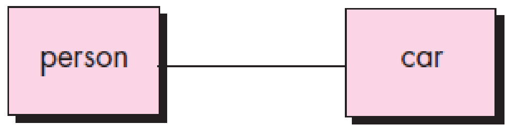
• A person owns a car.

• A person is insured to drive a car.

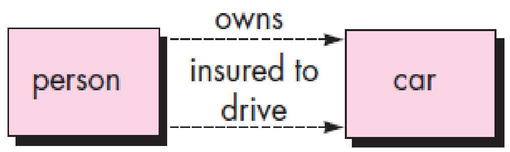
The relationships own and insured to drive define the relevant connections between person and car. Figure illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretation.

FIGURE 6.8

elationships
etween data
bjects



(a) A basic connection between data objects



(b) Relationships between data objects

Class-Responsibility-Collaborator (CRC) Modeling

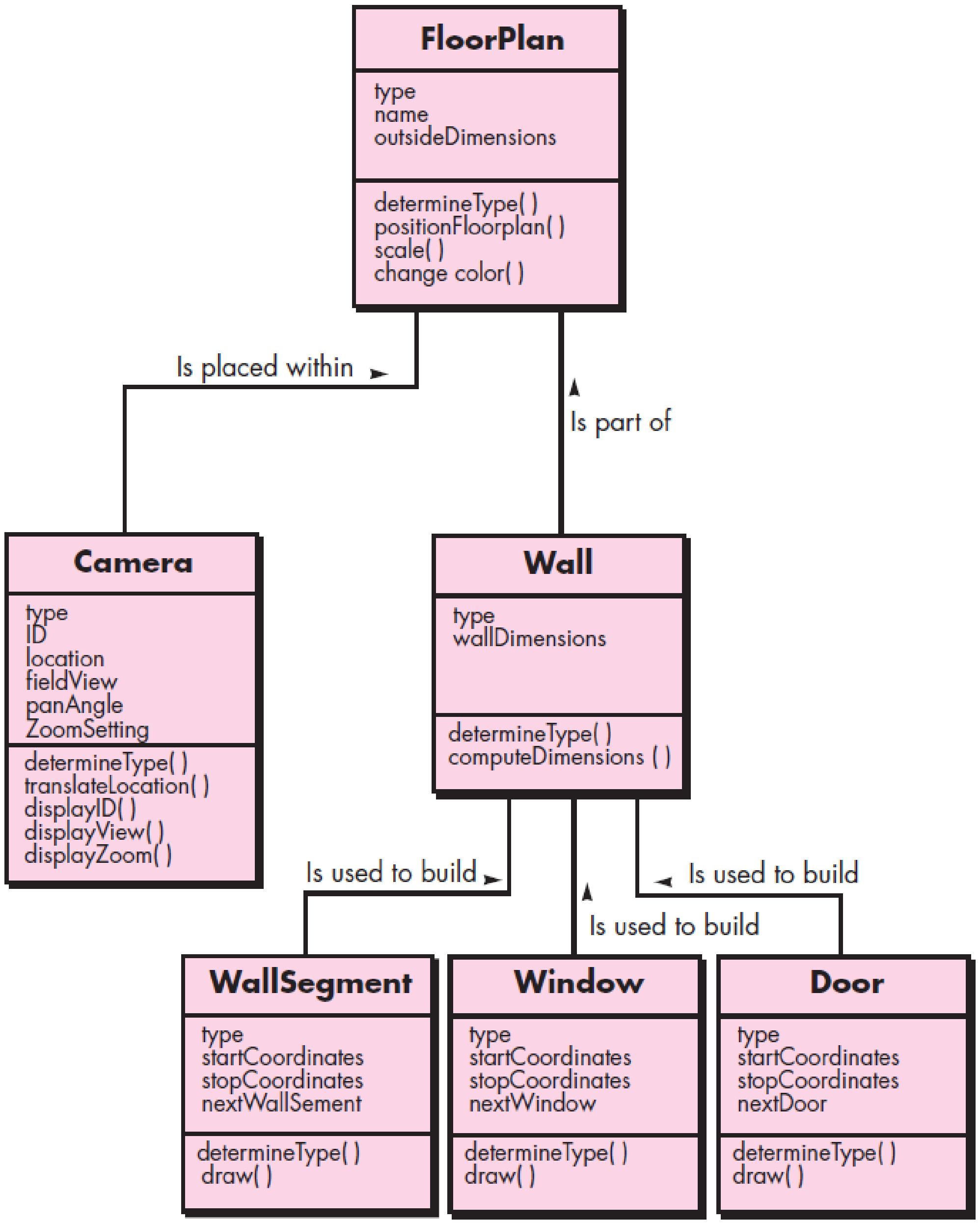
Class-responsibility-collaborator (CRC) modeling [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does". Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a collaboration implies either a request for information or a request for some action.

FIGURE 6.10

Class diagram for FloorPlan (see sidebar discussion)



A simple CRC index card for the FloorPlan class is illustrated in Figure.
The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification.

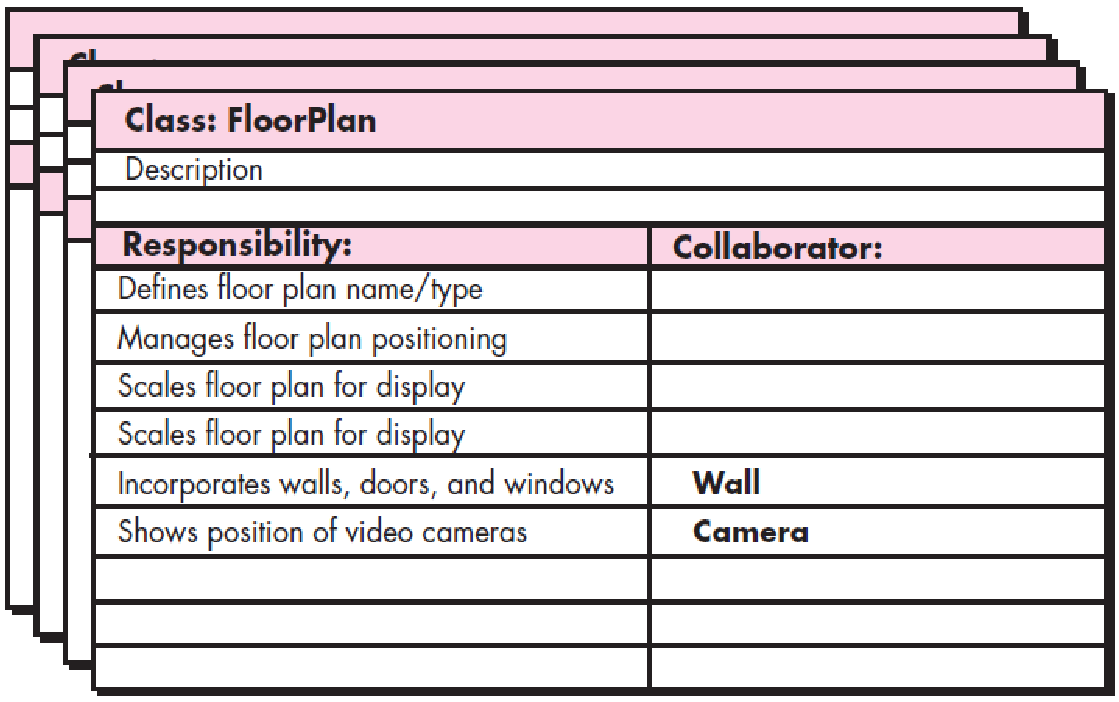
Classes. Basic guidelines for identifying classes and objects were presented earlier in this chapter. The taxonomy of class types extended by considering the following categories.

Entity classes, also called model or business classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor). These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.

Boundary classes are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. ntity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

FIGURE 6.11

CRC model
dex card



Class: FloorPlan

Description

Responsibility:	Collaborator:
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

Controller classes manage a "unit of work" from start to finish. That is, controller classes can be designed to manage
(1) the creation or update of entity objects,
(2) the instantiation of boundary objects as they obtain information from entity objects,
(3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application.
In general, controller classes are not considered until the design activity has begun.

Responsibilities.

Basic guidelines for identifying responsibilities (attributes and operations). five guidelines for allocating responsibilities to classes.

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways.
   To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence. In addition, the responsibilities for each class should exhibit the same level of abstraction.
2. **Each responsibility should be stated as generally as possible**. This guideline implies that general responsibilities should reside high in the class hierarchy.

3. **Information and the behavior related to it should reside within the same class**.This achieves the object-oriented principle called encapsulation. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes**. A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes**, when appropriate. There are many cases in which a variety of related objects must all exhibit the same behavior at the same time

Collaborations

Classes fulfill their responsibilities in one of two ways:

(1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or

(2) a class can collaborate with other classes.

Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A Collaboration is the embodiment of the contract between the client and the server. We say that an object collaborates with another object if, to fulfill a responsibility, it needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations is associated with a particular responsibility implemented by the server.

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

To help in the identification of collaborators, examine three different generic relationships between classes:

(1) the is-part-of relationship,

(2) the has-knowledge-of relationship, and

(3) the depends-upon relationship.

Each of the three generic relationships is considered briefly in the paragraphs that follow.

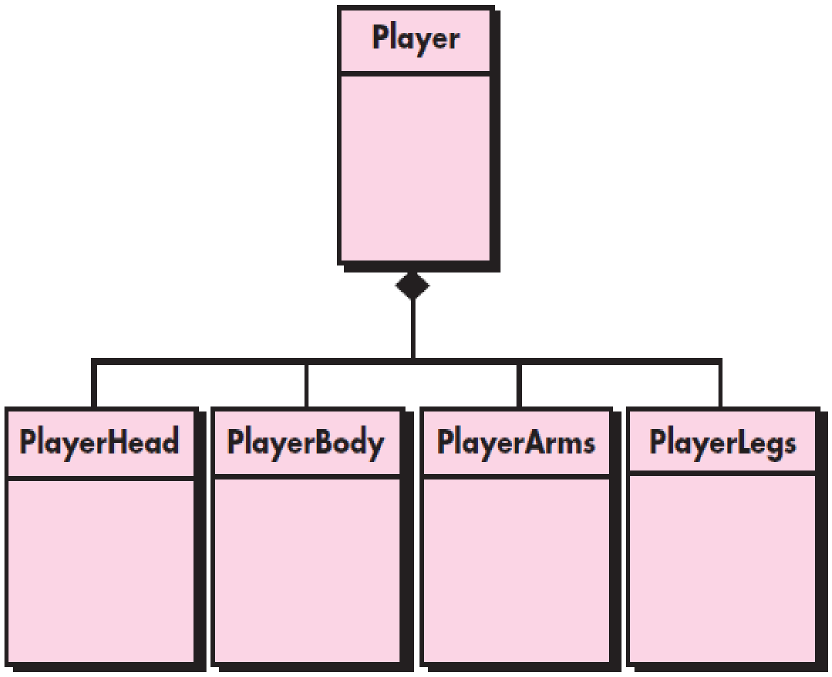
All classes that are part of an aggregate class are connected to the aggregate class
via an is-part-of relationship. When one class must acquire information from another class, the has-knowledgeof relationship is established. The determine-sensor-status() responsibility noted earlier is an example of a has-knowledge-of relationship.

The depends-upon relationship implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of.

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled.

FIGURE 6.12

A composite aggregate class



When a complete CRC model has been developed, stakeholders can review the model using the following approach.

1. All participants in the review are given a subset of the CRC model index cards. Cards that collaborate should be separated.
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
4. When the token is passed, the holder of the Sensor card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes or the specification of new or revised responsibilities or collaborations on existing cards.

Associations and Dependencies

In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another. In UML these relationships are called associations.

In some cases, an association may be further defined by indicating multiplicity. Referring to Figure. Where “one or more” is represented using 1. .*, and “0 or more” by 0 . .*. In UML, the asterisk indicates an unlimited upper bound on the range.1.

In many instances, a client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a dependency relationship is established. Dependencies are defined by a stereotype. A stereotype is an “extensibility mechanism” within UML that allows defining a special modeling element whose semantics are custom defined. In UML stereotypes are represented in double angle brackets (e.g., <<stereotype>>).

FIGURE 6.13

Multiplicity

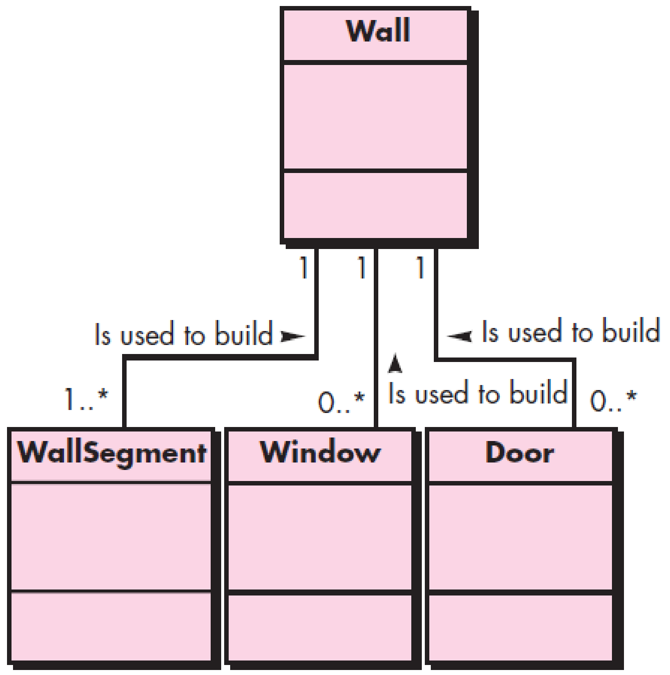
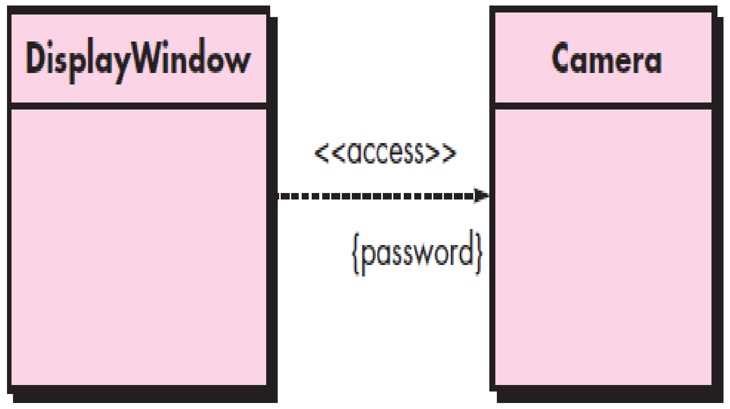


FIGURE 6.14

Dependencies



Analysis Packages

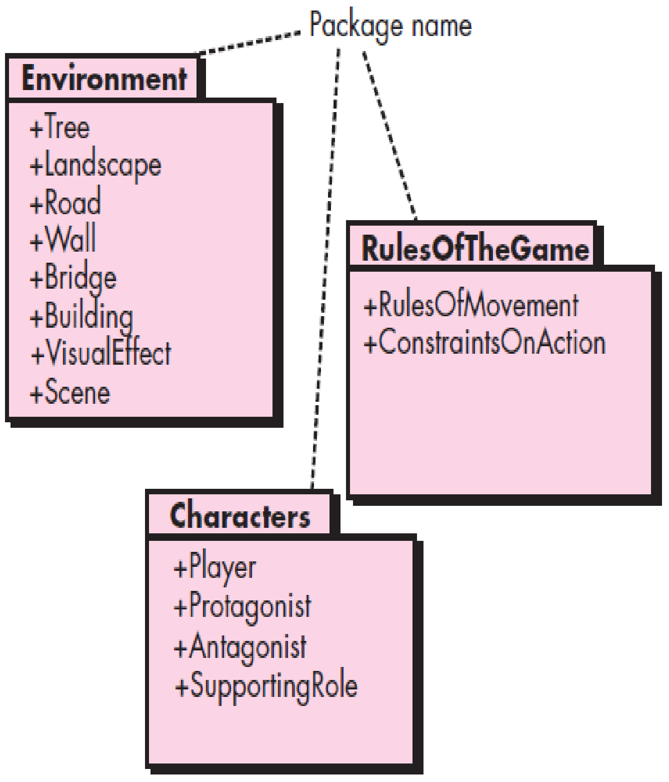
An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an analysis package—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that as the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and VisualEffect might fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints.

The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages. Although they are not shown in the figure, other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

FIGURE 6.15

Packages


Package name
Environment
+Tree
+Landscape
+Road
+Wall
+Bridge
+Building
+VisualEffect
+Scene
RulesOfTheGame
+RulesOfMovement
+ConstraintsOnAction
Characters
+Player
+Protagonist
+Antagonist
+SupportingRole