

Chapter 5

Understanding the requirements

5.1 Requirements engineering

- Requirements engineering is the wide range of activities and methods that result in a comprehension of requirements.
- Requirements engineering is a significant software engineering action that starts during the communication activity and extends into the modelling activity from the standpoint of the software process. It needs to be modified to meet the requirements of the project, the product, the workers, and the procedure.
- Requirements engineering builds a bridge to design and construction.
- Requirements engineering offers an effective framework for comprehending the customer's desires, evaluating necessity, determining feasibility, discussing a viable solution, articulating the solution clearly, confirming the accuracy of the specifications, and overseeing the evolution of requirements into a functional system.

It involves seven specific tasks:

Inception, elicitation, elaboration, negotiation, specification, validation, and management.

1. Inception

- Just chatting informally can unexpectedly lead to a significant software engineering project. However, typically, projects start when a business requirement emerges or a potential new market or service is uncovered.
- Business stakeholders such as business managers, marketing professionals, and product managers craft a business justification for the concept. They aim to gauge the extent and potential of the market, conduct a preliminary feasibility assessment, and outline a functional overview of the project's scope.
- At this stage, it's crucial to establish a foundational understanding of the problem at hand, the individuals seeking a solution, the type of solution desired, and the efficacy of initial communication and collaboration among stakeholders and the software team.

2. Elicitation

Several challenges arise during the process of requirements elicitation.

1. Problems of scope: The system's boundary is unclear, or customers/users provide excessive technical details that might complicate rather than clarify the overall objectives of the system.
2. Problems of understanding: The customers/users are uncertain about their exact needs, lack a thorough understanding of their computing environment's capabilities and constraints, and have incomplete knowledge of the problem domain.
3. Problems of volatility: The requirements change over time.

3. Elaboration

- Elaboration is guided by crafting and enhancing use case scenarios, detailing how the end user (and other actors in the scene) will engage with the system.
- Each user scenario is dissected to extract analysis classes, which are business domain entities observable to the end user. Attributes of each analysis class are specified, and the necessary services for each class are determined. Relationships and cooperation among classes are outlined, and various additional diagrams are generated.

4. Negotiation

- To address the conflicts in requirements gathering, negotiation becomes essential. Customers, users, and other stakeholders are prompted to prioritize requirements and engage in discussions to resolve conflicts.
- Employing an iterative method that focuses on prioritizing requirements, evaluating their costs and risks, and managing internal conflicts, adjustments are made. This may involve eliminating, combining, or modifying requirements to ensure that each party attains some level of satisfaction.

5. Specification

A specification may take various forms, including a written document, a series of graphical representations, and a formal mathematical model, a collection of use case scenarios, a prototype, or a blend of these methods.

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.

6. Validation

The main method for validating requirements is through technical review. This review involves a team comprising software engineers, customers, users, and other stakeholders who scrutinize the specification for errors in content or interpretation, areas necessitating clarification, omissions, inconsistencies (especially prevalent in large-scale products or systems), conflicting requirements, or requirements that are impractical or unattainable.



Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

INFO

7. Requirements management

Requirements for computer-based systems evolve over time, and the need for modifications in requirements persists throughout the system's lifecycle. Requirements management encompasses a range of tasks aimed at enabling the project team to identify, control and requirements and any alterations to them as the project progresses.

5.2 Establishing the groundwork

Steps required to establish the groundwork for the understanding of software requirements

5.2.1 Identifying Stakeholders

A stakeholder is

- Anyone who has a direct interest in or benefits from the system that is to be developed.
- Anyone who benefits in a direct or indirect way from the system which is being developed.
- Identified stakeholders are: Business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, and support and maintenance engineers.

5.2.2 Recognizing Multiple Viewpoints

- The requirements of the system will be examined from various perspectives.
- For instance, the marketing team focuses on functions and features that will attract the potential market, making the new system easy to sell.
- Business managers are interested in a feature set that can be developed within budget and ready to meet defined market windows.
- End users prefer features that are familiar and easy to learn and use.
- Software engineers are concerned with functions that may be invisible to non-technical stakeholders but that provide an infrastructure supporting more marketable functions and features.
- Support engineers prioritize the maintainability of the software.
- All stakeholder information, including any inconsistent and conflicting requirements are organised in a manner that enables decision-makers to select a set of internally consistent requirements for the system.

5.2.3 Working toward Collaboration

- The role of a requirements engineer is to pinpoint areas of commonality (i.e., requirements that all stakeholders agree on) and areas of conflict or inconsistency (i.e., requirements desired by one stakeholder that conflict with the needs of another).
- Collaboration doesn't always mean that requirements are decided by committee. Often, stakeholders contribute their perspectives on requirements, but a decisive "project champion" (such as a business manager or senior technologist) ultimately determines which requirements are accepted.



Using "Priority Points"

One way of resolving conflicting requirements and at the same time better understanding the relative importance of all requirements is to use a "voting" scheme based on *priority points*. All stakeholders are provided with some number of priority points that can be "spent" on any number of requirements. A list of requirements is presented, and each stakeholder indicates the relative importance of

each (from his or her viewpoint) by spending one or more priority points on it. Points spent cannot be reused. Once a stakeholder's priority points are exhausted, no further action on requirements can be taken by that person. Overall points spent on each requirement by all stakeholders provide an indication of the overall importance of each requirement.

INFO

5.2.4 Asking the First Questions

The first set of context-free questions focuses on the customer and other stakeholders. For example:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

- How would you characterize “good” output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication

- Are you the right person to answer these questions? Are your answers “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

5.3 Eliciting Requirements

Requirements elicitation (or requirements gathering) involves aspects of problem-solving, elaboration, negotiation, and specification. To promote a collaborative, team-oriented approach, stakeholders work together to identify the problem, propose solution elements, negotiate various approaches, and specify an initial set of solution requirements.

5.3.1 Collaborative Requirements Gathering

The objective is to recognize the problem, suggest components of the solution, discuss various strategies, and outline an initial set of solution requirements, all within an environment that supports achieving the objective.

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 objective is to recognize the problem, suggest components of the solution, discuss various strategies, and outline an initial set of solution requirements, all within an environment that supports achieving the objective.

During inception, the developer and customers write “product request”. A meeting location, time, and date are determined; a facilitator is appointed; and participants from the software team and other stakeholder groups are invited to join. The product request is shared with all attendees prior to the meeting.

A narrative about the home security function that is to be part of SafeHome:

Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with “alarm systems” so this would be an easy sell.

The home security function would protect against and/or recognize a variety of undesirable “situations” such as illegal entry, fire, flooding, carbon monoxide levels, and others. It’ll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

Prior to the meeting, each participant is asked to review the product request and create several lists: one of objects within the environment surrounding the system, another of objects the system will produce, and a third of objects the system will use to carry out its functions. Additionally, participants should

compile a list of services (processes or functions) that interact with or manipulate these objects. Finally, they need to develop lists of constraints (such as cost, size, and business rules) and performance criteria (such as speed and accuracy).

The goal is to create an agreed-upon list of objects, services, constraints, and performance criteria for the system that will be developed.

Each mini-specification is an elaboration of an object or service. For example, the mini-spec for the SafeHome object Control Panel might be:

The control panel is a wall-mounted unit that is approximately 9 × 5 inches in size. The control panel has wireless connectivity to sensors and a PC. User interaction occurs through a keypad containing 12 keys. A 3 × 3 inch LCD color display provides user feedback. Software provides interactive prompts, echo, and similar functions.

The mini-specs are shared with all stakeholders for discussion, where additions, deletions, and further details are made. This process may reveal new objects, services, constraints, or performance requirements that will be added to the initial lists.

SAFEHOME



Conducting a Requirements Gathering Meeting

The scene: A meeting room. The first requirements gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all SafeHome functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

5.3.2 Quality Function Deployment

QFD defines requirements in a way that maximizes customer satisfaction.

To achieve this, QFD focuses on understanding what is valuable to the customer and integrating these values throughout the engineering process. QFD identifies three types of requirements.

1. Normal requirements

The objectives and goals outlined for a product or system during meetings with the customer. If these requirements are met, the customer is satisfied. Examples of normal requirements include graphical displays, specific system functions, and defined levels of performance.

2. Expected requirements

These requirements are inherent to the product or system and may be so basic that the customer does not explicitly mention them. Their absence will lead to significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

3. Exciting requirements

These features exceed the customer's expectations and are highly satisfying when included. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multi-touch screen, visual voice mail) that delight every user of the product.

QFD gathers requirements through customer interviews and observations, surveys, and analysis of historical data (such as problem reports). This information is compiled into a **customer voice table**, which is reviewed with the customer and other stakeholders. Various diagrams, matrices, and evaluation methods are then employed to identify expected requirements and try to uncover exciting requirements.

5.3.3 Usage Scenarios

As requirements are collected, a comprehensive vision of the system's functions and features starts to take shape. However, it is challenging to advance to more technical software engineering tasks without understanding how these functions and features will be utilized by various types 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.

5.3.4 Elicitation Work Products

The work products generated from requirements elicitation will differ based on the system's or product's size. For most systems, these work products typically 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.

5.4 Developing use cases

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

- The initial step in crafting a use case involves identifying the group of "actors" participating in the scenario.
- Actors encompass the diverse individuals or devices utilizing the system or product within the context of the function and behavior being depicted.
- They embody the roles individuals (or devices) assume during the system's operation. More precisely, an actor is any entity external to the system that interacts with it. Each actor possesses one or more objectives while engaging with the system.
- An Actor represents a class of external entities that play just one role in the context of the use case
- After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter.

- Since requirements elicitation is an iterative process, not all actors are necessarily identified in the initial iteration. Primary actors can be identified in the first iteration, while secondary actors may become apparent as more is learned about the system.
- Primary actors are those directly involved in achieving the required system function and deriving its intended benefits, interacting frequently with the software.
- Secondary actors, on the other hand, support the system, enabling primary actors to carry out their tasks.
- Once actors have been identified, use cases can be developed.

Number of questions that should be answered by a use case:

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

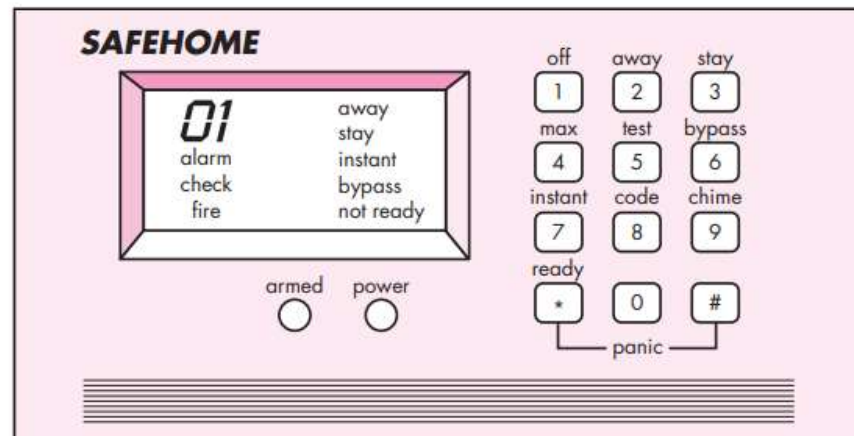
Referring back to the fundamental **SafeHome** requirements, we establish four actors: **the homeowner** (a user), **the setup manager** (potentially the same individual as the homeowner but fulfilling a distinct role), **sensors** (devices integrated into the system), and **the monitoring and response subsystem** (the central station overseeing the SafeHome home security operations). For this illustration, we focus solely on the **homeowner actor**. The **homeowner actor** engages with the home security function through various means, utilizing either the alarm control panel or a PC:

- Enters a password to allow all other interactions.
- Inquires about the status of a security zone.
- Inquires about the status of a sensor.
- Presses the panic button in an emergency.
- Activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation follows:

1. The homeowner observes the *SafeHome* control panel (Figure 5.1) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]

FIGURE 5.1
SafeHome
control panel



2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.
3. The homeowner selects and keys in *stay* or *away* (see Figure 5.1) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

Use cases are often written informally. However, use the template shown here to ensure that you've addressed all key issues.

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.

Priority: Essential, must be implemented

When available: First increment

Frequency of use: Many times per day

Channel to actor: Via control panel interface

Secondary actors: Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?
2. Should the control panel display additional text messages?
3. How much time does the homeowner have to enter the password from the time the first key is pressed?
4. Is there a way to deactivate the system before it actually activates?

SAFEHOME



Developing a High-Level Use-Case Diagram

The scene: A meeting room, continuing the requirements gathering meeting

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've spent a fair amount of time talking about *SafeHome* home security functionality. During the break I sketched a use case diagram to summarize the important scenarios that are part of this function. Take a look.

(All attendees look at Figure 5.2.)

Jamie: I'm just beginning to learn UML notation.¹⁴ So the home security function is represented by the big box with the ovals inside it? And the ovals represent use cases that we've written in text?

Facilitator: Yep. And the stick figures represent actors—the people or things that interact with the system as described by the use case . . . oh, I use the labeled square to represent an actor that's not a person . . . in this case, sensors.

Doug: Is that legal in UML?

Facilitator: Legality isn't the issue. The point is to communicate information. I view the use of a humanlike stick figure for representing a device to be misleading. So I've adapted things a bit. I don't think it creates a problem.

Vinod: Okay, so we have use-case narratives for each of the ovals. Do we need to develop the more detailed template-based narratives I've read about?

Facilitator: Probably, but that can wait until we've considered other *SafeHome* functions.

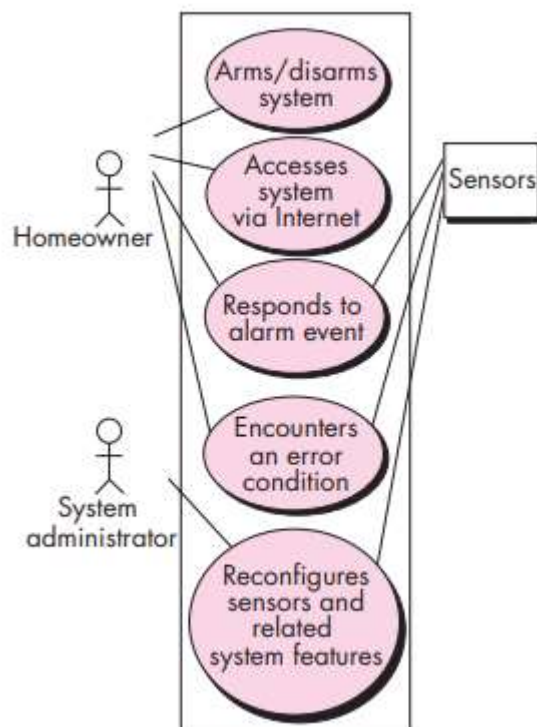
Marketing person: Wait, I've been looking at this diagram and all of a sudden I realize we missed something.

Facilitator: Oh really. Tell me what we've missed.

(The meeting continues.)

FIGURE 5.2

UML use case diagram for *SafeHome* home security function



SOFTWARE TOOLS

**Use-Case Development**

Objective: Assist in the development of use cases by providing automated templates and mechanisms for assessing clarity and consistency.

Mechanics: Tool mechanics vary. In general, use-case tools provide fill-in-the-blank templates for creating effective use cases. Most use-case functionality is embedded into a set of broader requirements engineering functions.

Representative Tools:¹⁵

The vast majority of UML-based analysis modeling tools provide both text and graphical support for use-case development and modeling.

Objects by Design

(www.objectsbydesign.com/tools/umltools_byCompany.html) provides comprehensive links to tools of this type.

5.5 Building the Requirements Model

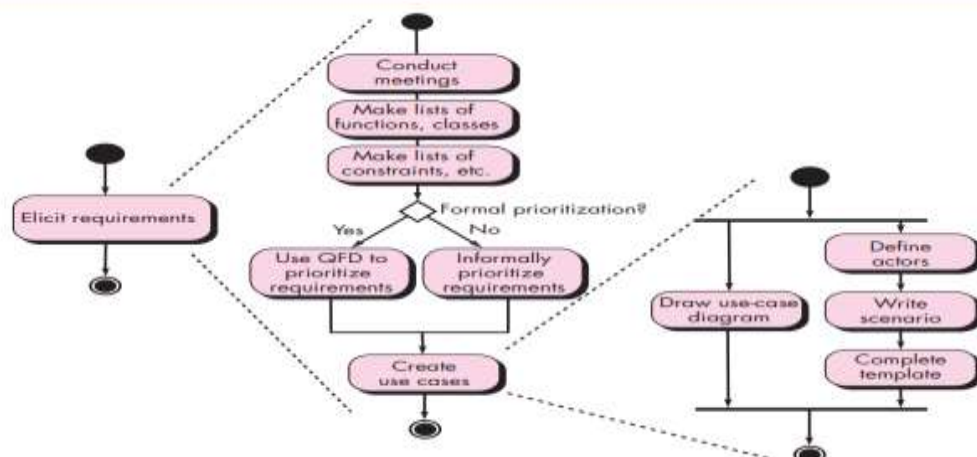
The purpose of the analysis model is to describe the necessary informational, functional, and behavioral domains of a computer-based system. This model evolves dynamically as you gain more insights into the system being developed and as stakeholders better comprehend their actual needs. Therefore, the analysis model represents a snapshot of the requirements at any given moment.

5.5.1 Elements of the Requirements Model

The specific components of the requirements model are determined by the chosen analysis modeling method.

1. Scenario-based elements: The system is depicted from the user's perspective using a scenario-based approach. For instance, basic use cases and their associated use-case diagrams evolve into more detailed template-based use cases. Scenario-based components of the requirements model are typically the first parts to be developed. Figure 5.3 depicts a UML activity diagram¹⁷ for eliciting requirements and representing them using use cases.

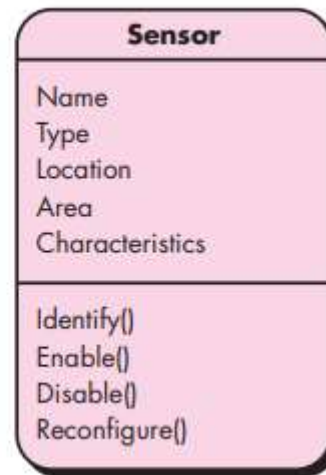
FIGURE 5.3
UML activity diagrams for eliciting requirements



2. Class-based elements: Each usage scenario involves a group of objects that an actor interacts with within the system. These objects are organized into classes, which represent collections of entities sharing similar attributes and behaviours. For instance, a UML class diagram can illustrate a Sensor class for the SafeHome security feature. In addition to class diagrams, other analysis modelling elements illustrate how classes collaborate with each other, as well as the relationships and interactions between them.

FIGURE 5.4

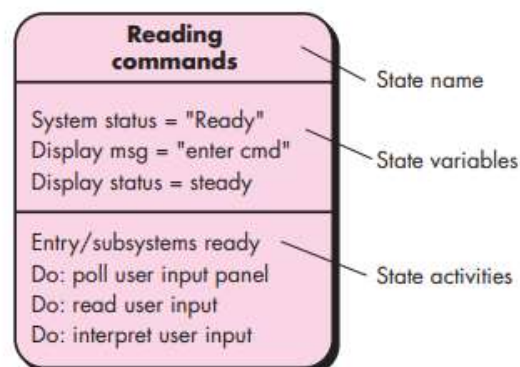
Class diagram
for sensor



3. Behavioural elements: Requirements model must provide modeling elements that depict behaviours. The state diagram is a technique used to represent a system's behaviour by showing its various states and the events that trigger state changes. A state represents any externally observable behaviour mode. Additionally, the state diagram highlights actions (such as process activation) that occur as a result of specific events. A simplified UML state diagram is shown in Figure 5.5.

FIGURE 5.5

UML state
diagram
notation



4. Flow-oriented elements: As information moves through a computer-based system, it undergoes transformation. The system receives input in various forms, processes it through different functions, and generates output in multiple formats.

5.5.2 Analysis Patterns

These analysis patterns propose solutions (such as a class, a function, or a behaviour) within the application domain that can be reused when modelling various applications.

Two advantages of using analysis patterns:

1. Analysis patterns accelerate the creation of abstract analysis models that capture the primary requirements of the specific problem by offering reusable models, complete with examples and a description of their benefits and limitations.
2. Analysis patterns aid in converting the analysis model into a design model by recommending design patterns and dependable solutions for common issues.

5.6 Negotiating Requirements

The goal of negotiation is to create a project plan that fulfils stakeholder' requirements while considering the real-world constraints (such as time, personnel, and budget) imposed on the software team.

Successful negotiations aim for a "win-win" outcome, where stakeholders receive a system or product that meets most of their needs, and the software team works within realistic and achievable budgets and deadlines.

Following activities are involved:

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)

Successfully completing these initial steps results in a win-win outcome, which becomes the primary criterion for advancing to the next stages of software engineering activities.



The Art of Negotiation

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve; what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen to her. It's likely you'll gain knowledge that will help you to better negotiate your position.
4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle; commit to it and move on.

INFO

5.7 Validating Requirements

Each element of the requirements model undergoes validation for inconsistencies, omissions, and ambiguities as it is developed. Stakeholders prioritize the requirements depicted in the model and organize them into packages that will be implemented as software increments.

A review of the requirements model evaluates the following questions:

1. Is each requirement consistent with the overall objectives for the system/product?
2. Have all requirements been specified at the proper level of abstraction? That
3. Is, do some requirements provide a level of technical detail that is inappropriate at this stage?
4. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
5. Is each requirement bounded and unambiguous?
6. Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
7. Do any requirements conflict with other requirements?
8. Is each requirement achievable in the technical environment that will house the system or product?
9. Is each requirement testable, once implemented?
10. Does the requirements model properly reflect the information, function, and behaviour of the system to be built?
11. Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

12. Have requirements patterns been used to simplify the requirements model?
Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Chapter 6

Requirements modelling: Scenarios, Information and Analysis Classes

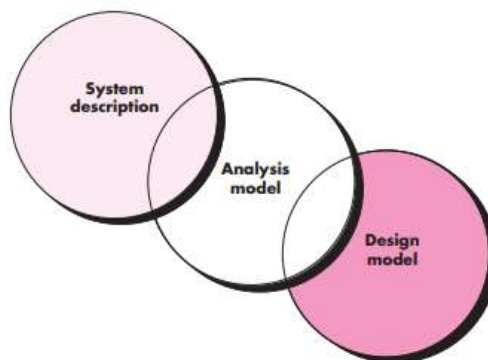
6.1 Requirements analysis

Requirements analysis leads to the detailed description of a software's operational features, defines how it will interact with other system components, and sets the limitations the software must adhere to.

The analysis model and requirements specification provide a means for assessing quality once the software is built.

FIGURE 6.1

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



6.1.1 Overall Objectives and Philosophy

The analysis model should describe what the customer wants, establish a basis for design, and establish a target for validation.

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.
- (3) The goal is to establish a set of requirements that can be verified after the software is developed. The analysis model serves as a bridge between a system-level overview, which outlines the overall functionality achieved through the integration of software, hardware, data, human, and other system components, and a software design that details the application's architecture, user interface, and component-level structure.

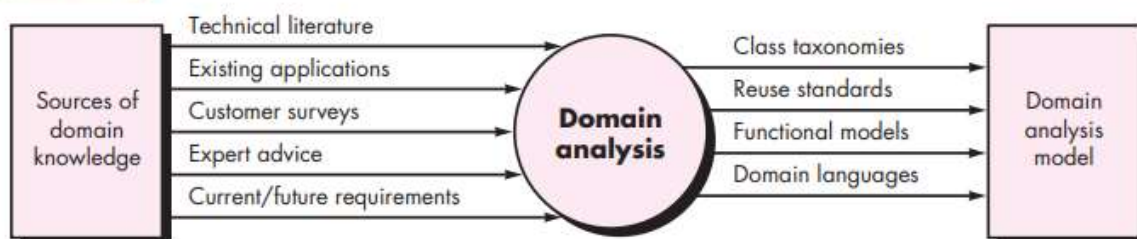
6.1.2 Analysis Rules of Thumb

1. The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
2. Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behaviour of the system.
3. Delay consideration of infrastructure and other non-functional models until design.
4. Minimize coupling throughout the system.
5. Be certain that the requirements model provides value to all stakeholders.
6. Keep the model as simple as it can be.

6.1.3 Domain Analysis

Software domain analysis involves identifying, analysing, and specifying common requirements within a particular application domain, usually for reuse in multiple projects within that domain. Object-oriented domain analysis, on the other hand, focuses on identifying, analysing, and specifying common, reusable capabilities within a specific application domain, described in terms of common objects, classes, subassemblies, and frameworks.

FIGURE 6.2 Input and output for domain analysis



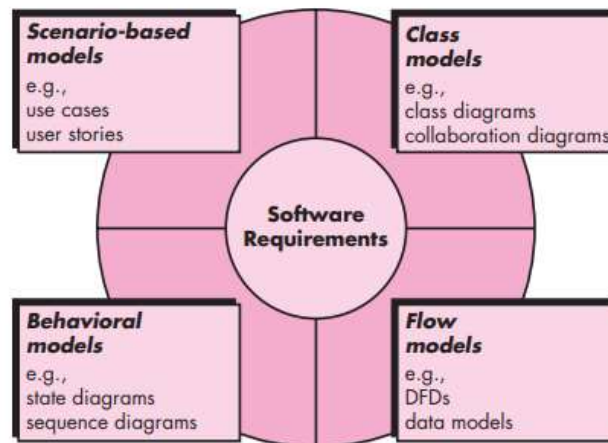
The "specific application domain" can vary widely, from avionics to banking, multimedia video games to software embedded in medical devices. The objective of domain analysis is clear: to identify or develop analysis classes and/or analysis patterns that are broadly applicable, enabling their reuse.

6.1.4 Requirements Modelling Approaches

- One perspective on requirements modelling, known as structured analysis, treats data and the processes that transform it as separate entities. In this approach, data objects are modelled to define their attributes and relationships, while processes are modelled to illustrate how they transform data as it flows through the system.
- Another approach, called object-oriented analysis, emphasizes defining classes and how they interact with each other to meet customer requirements.

FIGURE 6.3
Elements of
the analysis
model

? What
different
points of view
can be used to
describe the
requirements
model?



- Each component of the requirements model (Figure 6.3) represents the problem from a different perspective.
- Scenario-based elements illustrate user interactions with the system and the specific sequence of activities that occur during software use.
- Class-based elements model the objects the system will manipulate, the operations applied to these objects, the relationships (some hierarchical) between them, and the interactions between the defined classes.
- Behavioural elements show how external events alter the state of the system or its classes.
- Finally, flow-oriented elements represent the system as an information transformer, depicting how data objects are transformed as they pass through various system functions.

6.2 Scenario-Based Modelling

6.2.1 Creating a Preliminary Use Case

A use case describes a specific usage scenario in straightforward language from the perspective of a defined actor.

However, to effectively use cases as a requirements modelling tool, you need to determine: (1) what to write about, (2) how much to write, (3) the level of detail to include, and (4) how to organize the description. Answering these questions is crucial for the use cases to be valuable.

What to write about?

- The first two requirements engineering tasks—inception and elicitation—provide the information needed to start writing use cases.
- Requirements gathering meetings, QFD, and other requirements engineering techniques help identify stakeholders, define the problem's scope, specify overall operational goals, set priorities, outline all known functional requirements, and describe the objects the system will manipulate.
- To begin creating use cases, list the functions or activities performed by a specific actor. These can be derived from a list of required system functions, discussions with stakeholders, or an evaluation of activity diagrams developed during requirements modelling.

The SafeHome home surveillance function (subsystem) discussed in the sidebar identifies the following functions (an abbreviated list) 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.

For example, consider the function "access camera surveillance via the Internet—display camera views (ACS-DCV)." The stakeholder acting as the homeowner might provide the following narrative:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)**Actor: homeowner**

If I'm at a remote location, I can use any PC with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera" and the original viewing window disappears and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

A variation of a narrative use case presents the interaction as a sequence of user actions in a specific order, with each action described in a declarative sentence. Revisiting the ACS-DCV function, you would write:

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)**Actor: homeowner**

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

6.2.2 Refining a Preliminary Use Case

Describing alternative interactions is crucial for fully understanding the function outlined in a use case.

Each step in the primary scenario is assessed by asking the following questions:

1. Can the actor take some other action at this point?
2. Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
3. Is it possible that the actor will encounter some other behaviour at this point (e.g., behaviour that is invoked by some event outside the actor's control)? If so, what might it be?

Answers to these questions lead to the creation of secondary scenarios that are part of the original use case but represent alternative behaviours. For example, consider steps 6 and 7 in the primary scenario mentioned earlier:

6. The homeowner selects "pick a camera."

7. The system displays the floor plan of the house.

- Can the actor take a different action at this point? The answer is "yes." Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Therefore, one secondary scenario might be "View thumbnail snapshots for all cameras."
- Is it possible the actor will encounter an error condition at this point? Numerous error conditions can occur in a computer-based system. Here, we consider only those errors likely as a direct result of the actions in steps 6 or 7. Again, the answer is "yes." A floor plan with camera icons might not have been configured. Thus, selecting "pick a camera" could result in an error condition: "No floor plan configured for this house." This error condition becomes a secondary scenario.
- Is it possible the actor will encounter other behaviours at this point? The answer is "yes" once more. As steps 6 and 7 occur, the system might encounter an alarm condition, leading to a special alarm notification (type, location, system action) and presenting the actor with several options relevant to the alarm. Since this secondary scenario can occur at any time during virtually all interactions, it will not be included in the ACS-DCV use

case. Instead, a separate use case—"Alarm condition encountered"—would be developed and referenced from other use cases as needed.

- Each situation described in the preceding paragraphs is characterized as a use-case exception. An exception outlines a scenario (either a failure condition or an alternative action chosen by the actor) that causes the system to display different behavior.
- "Brainstorming" session is recommended to derive a reasonably complete set of exceptions for each use case.

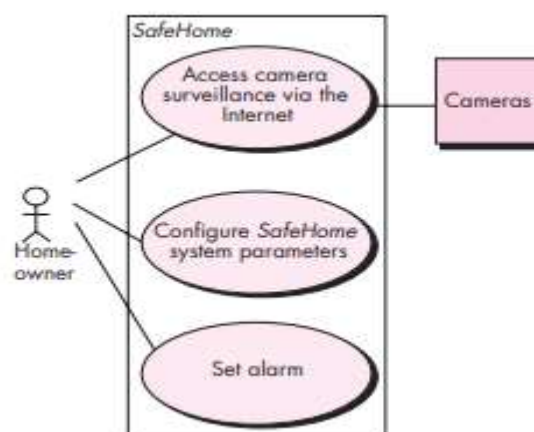
The following issues should also be explored:


- Are there cases in which some "validation function" occurs during this use case?
- Are there cases in which a supporting function (or actor) will fail to respond appropriately?
- Can poor system performance result in unexpected or improper user actions?

6.2.3 Writing a Formal Use Case

Often, there is no need to create a graphical representation of a usage scenario. However, a diagram can enhance understanding, especially when the scenario is complex. As mentioned earlier in this book, UML offers use-case diagramming capabilities. Figure 6.4 shows a preliminary use-case diagram for the SafeHome product, where each use case is represented by an oval. Only the ACS-DCV use case has been discussed in this section.

FIGURE 6.4
Preliminary
use-case
diagram for
the SafeHome
system




SAFEHOME

Use Case Template for Surveillance

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of camera placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the “surveillance” from the major function buttons.
6. The homeowner selects “pick a camera.”
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the “view” button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects “View thumbnail snapshots for all camera”—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Moderate frequency.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

Channels to secondary actors:

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

Every modelling notation has its limitations, and use cases are no exception. Like any written description, a use case is only as effective as its author(s). If the description is unclear, the use case can be misleading or ambiguous. Use cases focus on functional and behavioural requirements and are generally unsuitable for non-functional requirements. For situations requiring significant detail and precision, such as safety-critical systems, a use case may not be sufficient.

However, scenario-based modelling is appropriate for the majority of situations encountered as a software engineer. When properly developed, use cases can be highly beneficial as a modelling tool.

6.3 UML models that supplement the use case

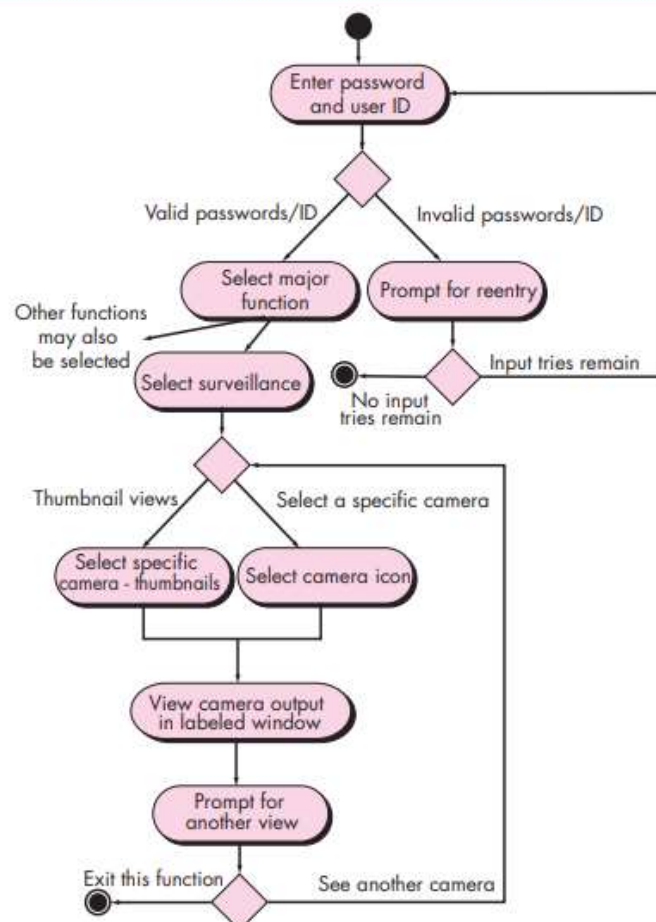
6.3.1 Developing an Activity Diagram

A UML activity diagram represents the actions and decisions that occur as some function is performed.

- The UML activity diagram complements the use case by offering a visual representation of the interaction flow within a particular scenario.
- Similar to a flowchart, an activity diagram utilizes rounded rectangles to denote specific system functions, arrows to illustrate flow through the system, decision diamonds to indicate branching decisions (each arrow originating from the diamond is labeled), and solid horizontal lines to signify parallel activities.
- Figure 6.5 presents an activity diagram for the ACS-DCV use case. It's important to note that the activity diagram provides additional detailed insights that are implied but not explicitly mentioned in the use case. For example, a user may only attempt to enter userID and password a limited number of times. This is represented by a decision diamond below “Prompt for re-entry.”

FIGURE 6.5

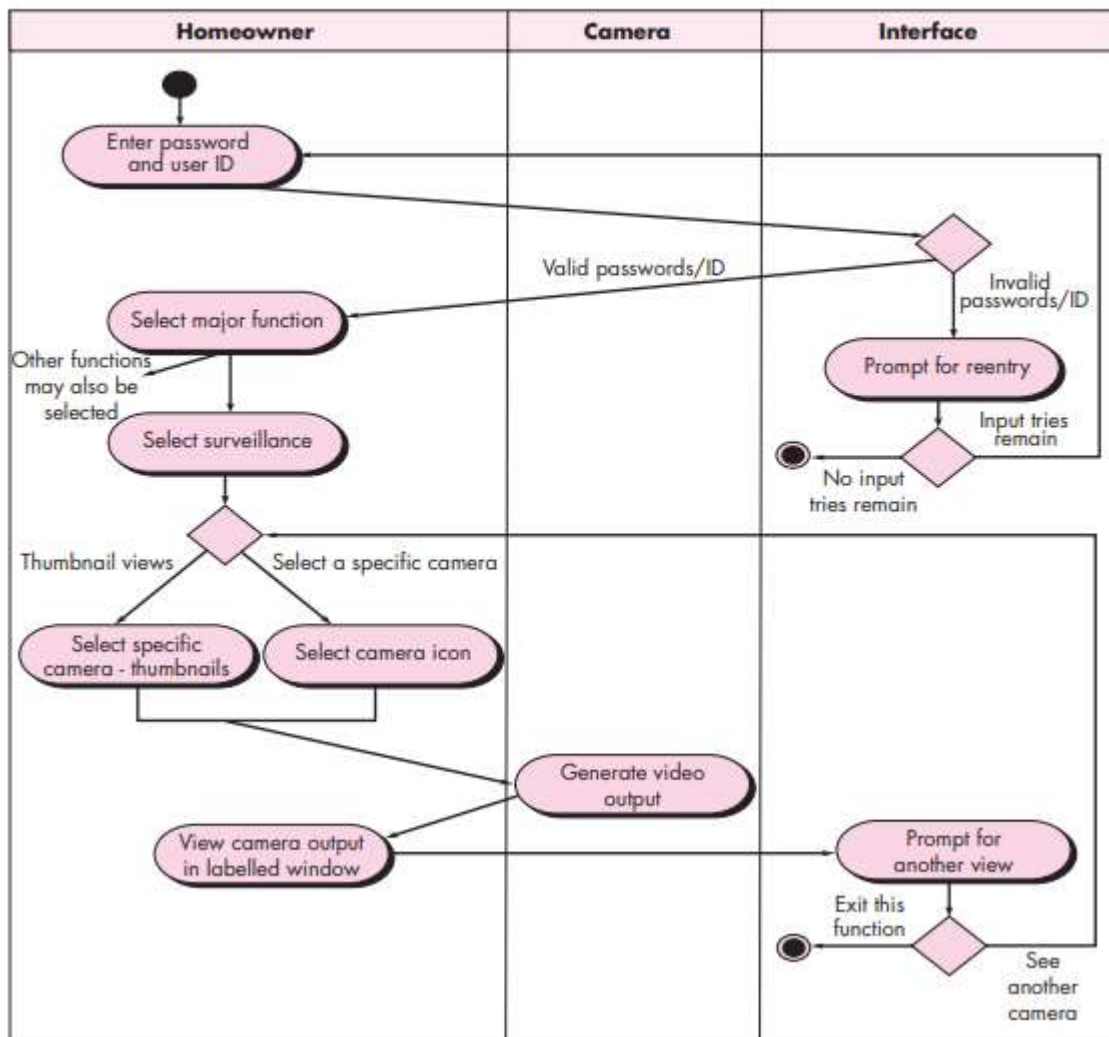
Activity diagram for Access camera surveillance via the Internet—display camera views function.



6.3.2 Swim lane Diagrams

- A UML swim lane diagram represents the flow of actions and decisions and indicates which actors perform each.
- The UML swim lane diagram provides a valuable variation of the activity diagram.
- It allows for the representation of activity flows described by the use case while indicating which actor (in cases involving multiple actors) or analysis class (discussed later in this chapter) is responsible for each action depicted by an activity rectangle.
- Responsibilities are depicted as parallel segments that divide the diagram vertically, resembling lanes in a swimming pool.
- In the context of the activity diagram shown in Figure 6.5, three analysis classes—Homeowner, Camera, and Interface—have direct or indirect responsibilities.
- Referring to Figure 6.6, the activity diagram is structured so that activities related to a specific analysis class are contained within the swim lane assigned to that class. For instance, the Interface class represents the user interface viewed by the homeowner.
- The activity diagram indicates two prompts handled by the interface—"prompt for re-entry" and "prompt for another view." These prompts and their associated decisions are situated within the Interface swim lane.
- Arrows extend from this swim lane back to the Homeowner swim lane, where homeowner actions take place.
- Use cases, alongside activity and swim lane diagrams, are oriented towards procedures. They illustrate how various actors initiate specific functions (or other procedural steps) to fulfil system requirements. However, a procedural perspective of requirements only captures a single dimension of a system.

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

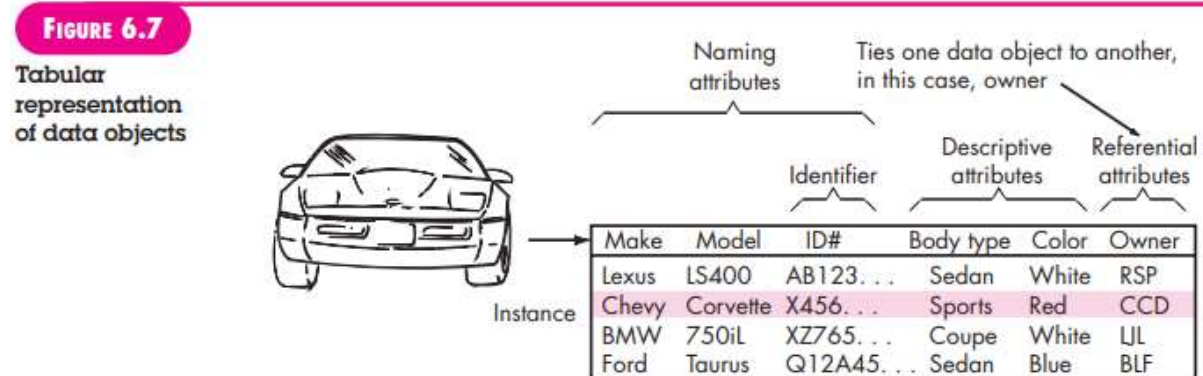


6.4 Data Modelling Concepts

A software engineer or analyst specifies all data objects processed within the system, their relationships, and other relevant information about these relationships. The entity-relationship diagram (ERD) is used to address these aspects and visually represents all data objects that are input, stored, transformed, and output within an application.

6.4.1 Data Objects

- A data object in software represents composite information, defined by multiple attributes. For example, "dimensions" (including height, width, and depth) qualifies as a data object, whereas "width" alone does not.
- Data objects can take various forms: they can represent external entities, entities like reports or displays, occurrences such as telephone calls or events like alarms, roles like salespeople, organizational units such as accounting departments, places like warehouses, or structures like files. For instance, both a person and a car can be considered data objects due to their attributes.
- A data object's description includes the object itself and all its attributes. Importantly, a data object encapsulates only data and does not include operations that manipulate it. It can be represented as a table (see Figure 6.7), where headings denote attributes such as make, model, ID number, body type, colour, and owner for a car. Rows in the table represent specific instances of the data object; for instance, a Chevy Corvette would be one such instance.



6.4.2 Data Attributes

Attributes name a data object, describe its characteristics, and in some cases, make reference to another object.

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.

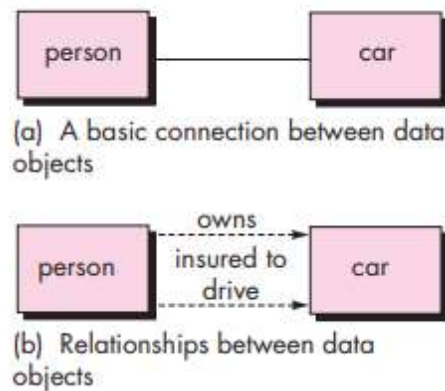
The selection of attributes for a given data object depends on the specific context of the problem. For example, attributes such as ***make, model, ID number, body type,*** and ***colour*** might be suitable for an application used by a department of motor vehicles.

However, these attributes would be inadequate for an automobile manufacturing control software. In the manufacturing context, additional attributes such as ***interior code, drivetrain type, trim package designator,*** and ***transmission type*** would be essential to fully define the “car” object and make it meaningful within that domain.

6.4.3 Relationships

- Relationships indicate the manner in which data objects are connected to one another.
- Data objects can be interconnected in various ways. Take, for example, the two data objects, person and car. These objects are linked as shown in the simplified notation in Figure 6.8a because they share a relationship.
- But what exactly are these relationships? To determine this, one must understand how people (specifically owners in this case) and cars function within the software context being developed.
- By establishing a set of object/relationship pairs, one can define these connections. For instance:
 - A person owns a car.
 - A person is insured to drive a car.
- These relationships—owns and insured to drive—define the relevant connections between person and car. Figure 6.8b depicts these object-relationship pairs graphically. The arrows in Figure 6.8b indicate the directionality of these relationships, which helps clarify their meaning and reduces potential ambiguity or misinterpretation.

FIGURE 6.8
Relationships
between data
objects



6.5 Class-based modelling

Class-based modeling describes the objects the system will handle, the operations (or methods/services) applied to manipulate these objects, relationships (which can be hierarchical) between objects, and collaborations among defined classes.

Elements of a class-based model encompass classes and objects, attributes, operations, Class-Responsibility-Collaborator (CRC) models, collaboration diagrams, and packages. The following sections provide informal guidelines to help identify and represent these elements effectively.

6.5.1 Identifying Analysis Classes

Analysis classes manifest themselves in one of the following ways:

- **External entities** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units** (e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.

- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a Class of objects or related classes of objects.

To demonstrate how analysis classes could be defined in the initial stages of modelling, let's examine a grammatical parse (with nouns underlined and verbs italicized) of a processing narrative for the SafeHome security function.

The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed*, *monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is *programmed* for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides information* about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained*.

The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

Extracting the nouns, we can propose a number of potential classes:

| Potential Class | General Classification |
|--------------------------------|--|
| homeowner | role or external entity |
| sensor | external entity |
| control panel | external entity |
| installation | occurrence |
| system (alias security system) | thing |
| number, type | not objects, attributes of sensor |
| master password | thing |
| telephone number | thing |
| sensor event | occurrence |
| audible alarm | external entity |
| monitoring service | organizational unit or external entity |

Here are six selection criteria that should be applied when considering each potential class for inclusion in the analysis model:

1. **Retained information.** The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. **Common operations.** A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

| Potential Class | Characteristic Number That Applies |
|----------------------------------|---|
| homeowner | rejected: 1, 2 fail even though 6 applies |
| sensor | accepted: all apply |
| control panel | accepted: all apply |
| installation | rejected |
| system (alias security function) | accepted: all apply |
| number, type | rejected: 3 fails, attributes of sensor |
| master password | rejected: 3 fails |
| telephone number | rejected: 3 fails |
| sensor event | accepted: all apply |
| audible alarm | accepted: 2, 3, 4, 5, 6 apply |
| monitoring service | rejected: 1, 2 fail even though 6 applies |

6.5.2 Specifying Attributes

- Attributes are the set of data objects that fully define the class within the context of the problem.
- Attributes define a class that has been chosen for inclusion in the requirements model. They specify what the class represents within the problem space. For instance, if we consider a system tracking baseball statistics for professional players, the attributes of the class "Player" would differ significantly from those used in a professional baseball pension system.
- In the former, relevant attributes might include name, position, batting average, fielding percentage, years played, and games played.
- Conversely, attributes in the latter context would focus on different aspects such as average salary, credit towards full vesting, chosen pension plan options, and mailing address.

For example, let's examine the System class defined for SafeHome. A homeowner has the ability to configure the security function, encompassing sensor information, alarm response details, activation/deactivation settings, identification data, and other related information. These composite data items can be represented as follows:

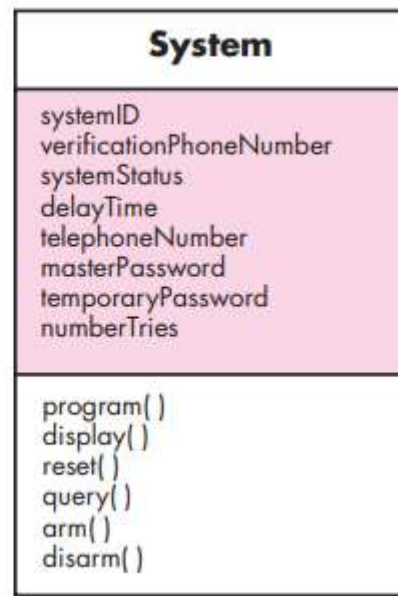
```
identification information = system ID + verification phone number + system status  
alarm response information = delay time + telephone number  
activation/deactivation information = master password + number of allowable tries +  
temporary password
```

6.5.3 Defining Operations

Operations define how an object behaves. Although operations come in various types, they can generally be categorized into four broad groups: (1) operations that manipulate data (e.g., adding, deleting, reformatting, selecting), (2) operations that perform computations, (3) operations that query the state of an object, and (4) operations that monitor an object for specific events.

FIGURE 6.9

Class diagram
for the system
class



For instance, in the SafeHome processing narrative discussed earlier in this chapter, statements such as "sensor is assigned a number and type" or "a master password is programmed for arming and disarming the system" indicate several points:

- The Sensor class is relevant for an assign() operation.
- The System class will involve a program() operation.
- Operations like arm() and disarm() are applicable to the System class.

6.5.4 Class-Responsibility-Collaborator (CRC) Modelling

A CRC (Class-Responsibility-Collaborator) model consists of standard index cards representing classes. Each card is divided into three sections: at the top, you write the name of the class. In the body of the card, responsibilities of the class are listed on the left, and collaborators are listed on the right.

- The goal is to create a structured representation of classes. Responsibilities refer to the attributes and operations that are pertinent to a class.
- Put simply, a responsibility encompasses "anything the class knows or does".
- Collaborators, on the other hand, are classes that need to provide information to fulfil a responsibility. Generally, collaboration involves either requesting information or requesting action.

An example of a CRC index card for the FloorPlan class is depicted in Figure 6.11. The list of responsibilities on the CRC card is preliminary and may be expanded

or revised. The classes Wall and Camera are identified alongside responsibilities that require their collaboration.

FIGURE 6.10

Class diagram
for FloorPlan
(see sidebar
discussion)

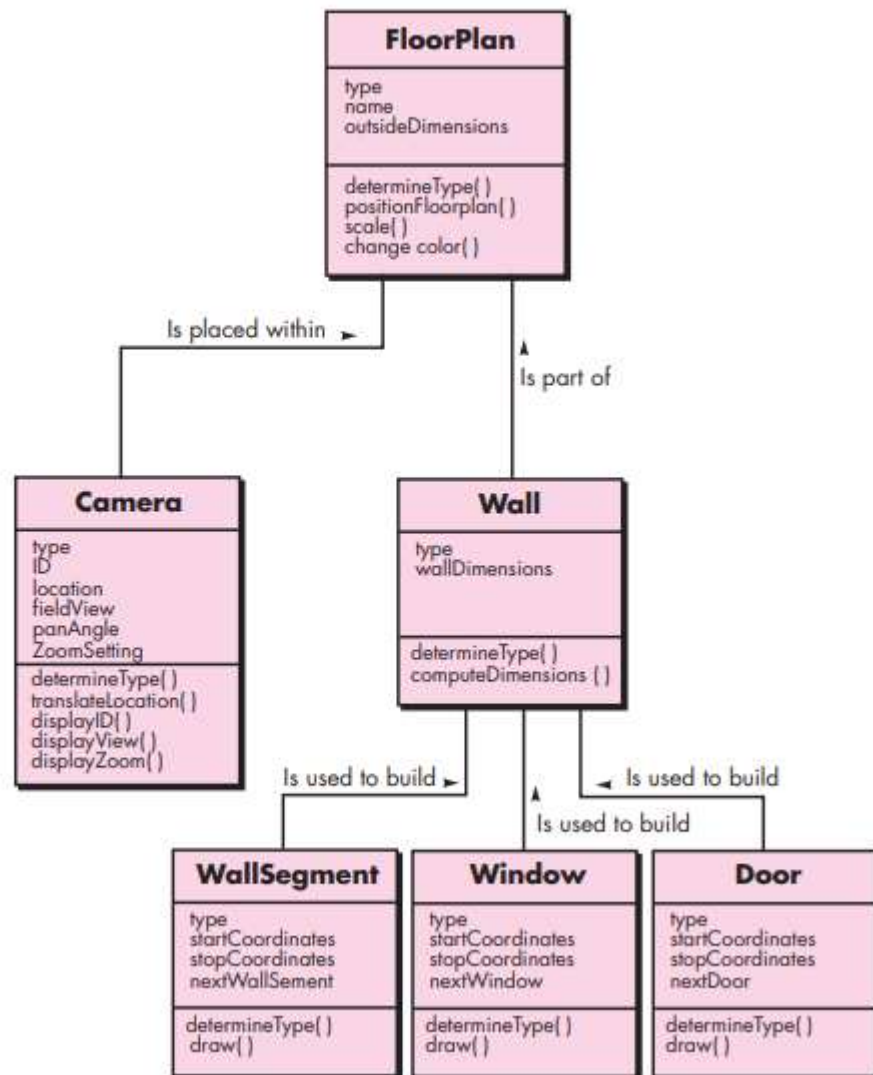


FIGURE 6.11

A CRC model
index 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 |
| | |
| | |
| | |

Classes

- **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 (unless they are specifically deleted).
- **Boundary classes** are responsible for creating the interface (e.g., interactive screens or printed reports) through which users see and interact with the software during its use. These classes manage how entity objects, which contain crucial information for users, are presented but do not display themselves. For instance, a boundary class like CameraWindow would be tasked with showing surveillance camera output in the SafeHome system.
- **Controller classes** oversee a "unit of work" from start to finish. They can handle tasks such as creating or updating entity objects, coordinating the instantiation of boundary objects to gather information from entity objects, managing complex interactions between sets of objects, and validating data exchanged between objects or between the user and the application.

Responsibilities

Five guidelines for allocating responsibilities to classes:

1. System intelligence should be distributed across classes to best address the needs of the problem.
2. Each responsibility should be stated as generally as possible
3. Information and the behavior related to it should reside within the same class
4. Information about one thing should be localized with a single class, not distributed across multiple classes
5. Responsibilities should be shared among related classes, when appropriate

Collaborations

Collaborations depict the interactions where a client requests a server to fulfil a client responsibility. It represents the agreement or contract between the client and the server. When we say an object collaborates with another object, it means that to fulfil a responsibility, the object needs to send messages to the other object.

Each collaboration flows in a specific direction—reflecting a request from the client to the server. From the client's perspective, each collaboration is linked to a specific responsibility that the server implements.

For instance, let's look at the SafeHome security function. During the activation process, the ControlPanel object needs to verify if any sensors are open. A responsibility named `determine-sensor-status()` is defined for this purpose. If sensors are found to be open, ControlPanel must set a status attribute to "not ready." To gather sensor information, ControlPanel collaborates closely with each Sensor object.

To identify collaborators effectively, three generic relationships between classes can be examined: (1) the is-part-of relationship, (2) the has-knowledge-of relationship, and (3) the depends-upon relationship. Each relationship type is briefly explained in the following paragraphs.

All classes that are part of an aggregate class are linked to the aggregate class through an is-part-of relationship. For example, in the context of a video game, classes like PlayerBody, PlayerArms, PlayerLegs, and PlayerHead are all part of the Player class. In UML notation, these relationships are depicted as aggregations, as shown in Figure 6.12.

When one class needs to obtain information from another class, a has-knowledge-of relationship is established. For instance, the `determine-sensor-status()` responsibility mentioned earlier exemplifies a has-knowledge-of relationship.

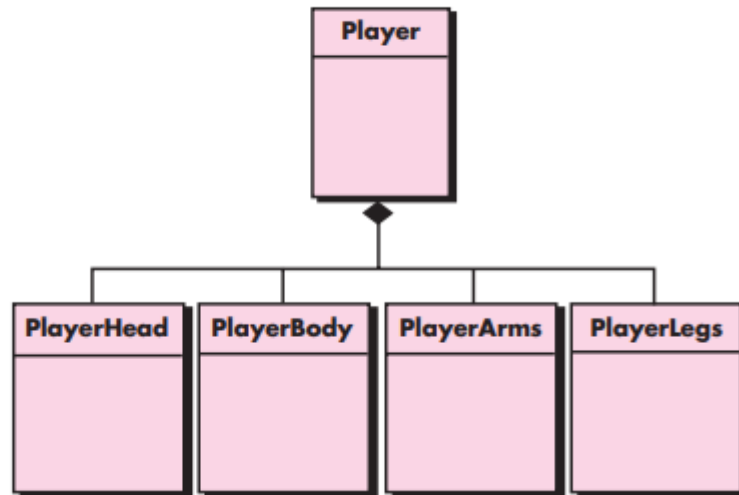
On the other hand, the depends-upon relationship indicates that two classes have a dependency that isn't covered by has-knowledge-of or is-part-of relationships. For example, in a video game scenario, PlayerHead must always be connected to PlayerBody (except in specific contexts like violent games), yet each object could exist without direct awareness of the other. An attribute of PlayerHead, such as center-position, relies on information derived from PlayerBody, which is obtained through a third object, Player. Therefore, PlayerHead depends-upon PlayerBody.

In all instances, the name of the collaborator class is recorded on the CRC (Class-Responsibility-Collaborator) model index card alongside the responsibility that

necessitates the collaboration. This index card serves to list responsibilities and their corresponding collaborations, facilitating the fulfilment of responsibilities (see Figure 6.11).

FIGURE 6.12

A composite
aggregate
class



Stakeholders can review the model using the following approach:

1. Each participant in the review session receives a subset of CRC model index cards. Cards that collaborate should be distributed so that no reviewer holds two collaborating cards simultaneously.
2. All use-case scenarios and their corresponding diagrams should be categorized and organized systematically.
3. The review leader reads each use case methodically. When encountering a named object in the use case, the leader passes a token to the individual holding the corresponding class index card.
4. Upon receiving the token, the holder of the Sensor card is tasked with describing the responsibilities listed on their card. The group assesses whether any of these responsibilities meet the requirements outlined in the use case.
5. If the responsibilities and collaborations documented on the index cards do not align with the use case, adjustments are made. This may involve creating new classes and corresponding CRC index cards, or refining existing cards by specifying new or revised responsibilities and collaborations.

6.5.5 Associations and Dependencies

An association defines a relationship between classes. Multiplicity defines how many of one class are related to how many of another class.

FIGURE 6.13

Multiplicity

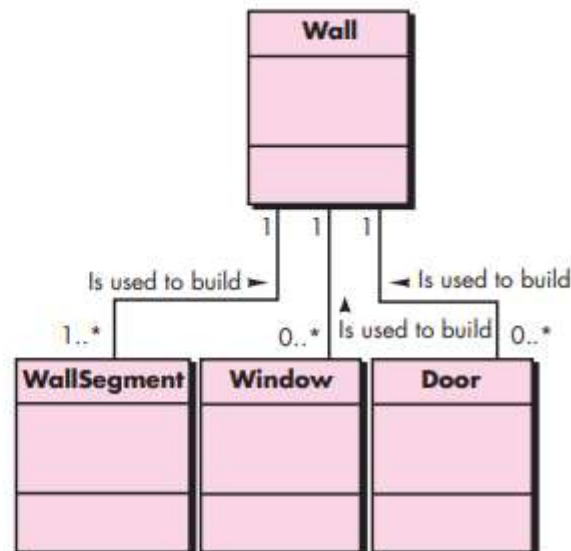
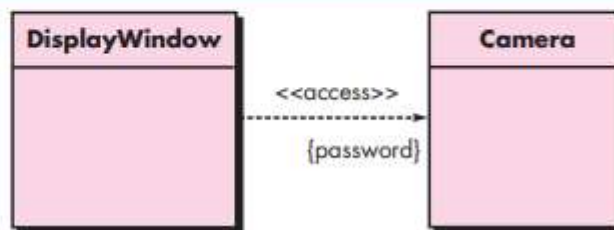


FIGURE 6.14

Dependencies



In some cases, an association can be specified further by indicating its multiplicity. Referring to Figure 6.10, a Wall object consists of one or more WallSegment objects. Additionally, the Wall object can include zero or more Window objects and zero or more Door objects. These multiplicity constraints are depicted in Figure 6.13, where "one or more" is denoted by 1..* and "zero or more" by 0..*. In UML, the asterisk (*) signifies an unlimited upper bound.

In many scenarios, there exists a client-server relationship between two analysis classes. Here, the client class relies on the server class in some capacity, establishing a dependency relationship. Such dependencies are defined using a stereotype, which serves as an extensibility mechanism within UML. Stereotypes are indicated using double angle brackets (e.g., <<stereotype>>).

6.5.6 Analysis Packages

A package is used to assemble a collection of related classes.

- To illustrate the concept of analysis packages, let's consider the video game example introduced earlier. As we develop the analysis model for the video game, numerous classes emerge. Some classes focus on the game environment, representing the visual scenes that players interact with during gameplay. Examples include Tree, Landscape, Road, Wall, Bridge, Building, and VisualEffect.
- Other classes concentrate on the characters within the game, detailing their physical attributes, behaviors, and limitations. These classes may include Player (as discussed previously), Protagonist, Antagonist, and SupportingRoles.
- Additionally, there are classes that define the game's rules and mechanics, governing how players navigate through the virtual environment. Classes like RulesOfMovement and ConstraintsOnAction fall into this category.
- These various categories of classes can be organized into analysis packages, as depicted in Figure 6.15. The presence of a plus sign preceding the analysis class name within each package denotes that these classes possess public visibility. This means they are accessible and can be utilized by other packages within the system.
- A minus sign (-) indicates that an element is private or hidden from all other packages. A hash symbol (#) indicates that an element is accessible only to packages contained within a specific package.

FIGURE 6.15

Packages

