ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: SOFTWARE ENGINEERING & PROJECT MANAGEMENT

COURSE CODE: BCS501

SEMESTER: V

COURSE COORDINATOR: Mr. Chetan G

(ACADEMIC YEAR 2026-27)

MODULE 2

CHAPTER 1—UNDERSTANDING REQUIREMENTS

1.1 REQUIREMENTS ENGINEERING

- *Requirements analysis*, also called *Requirements engineering*, is the process of determining user expectations for a new or modified product.

- 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.*

1.1.1 Requirements Engineering Task

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.**

1. **Inception:** It establishes 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.

2. **Elicitation**: In this stage, proper information is extracted to prepare and document the requirements. 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.

    - **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 objectives.

    - **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 un testable.

- **Problems of volatility**. In this problem, the requirements change from time to time and it is difficult while developing the project.

3. **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.

4. **Negotiation**: To negotiate the requirements of a system to be developed, it is necessary to identify **conflicts** and to resolve those conflicts. 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**.

5. **Specification**: 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.

6. **Validation**: Requirements validation **examines** the specification 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 requirements.

7. **Requirements management**. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system.
   It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project. Many of these activities are identical to the software configuration management (SCM) techniques. These tasks start with the identification and assign a **unique identifier** to each of the requirement. After finalizing the requirement **traceability table** is developed.

1.2 ESTABLISHING THE GROUNDWORK

There could be many issues to start Requirement Engineering Process:

- Customers or end users may be located in a different city/country.
- Customers do not have a clear idea of the requirements.
- Lack of technical knowledge or customers have limited time to interact with requirement engineer.

Hence by following the below steps we can start a requirement engineering process.

3.2.1 Identifying Stakeholders

- Stakeholder is the one who benefits in a direct or indirect way from the system which is being developed.

- Business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others.

- Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

3.2.2 Recognizing Multiple ViewPoints

- As many stakeholders exist, they all have different views regarding the system to be developed, hence it is the duty of software engineers to consider all the viewpoints of stakeholders in a way that allows decision makers to choose an internally consistent set of requirements for the system.

- For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell and End users may want features that are easy to learn and use.

3.2.3 Working toward Collaboration

Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

3.2.4 Asking the First Questions

Questions asked at the inception of the project should be "**context fre**e". The first set of context - free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- 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?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

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 activity itself. Gause and Weinberg call these "meta-questions" and propose the following list:

- 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?

These questions will help to "break the ice" and initiate the communication that is essential to successful elicitation.

1.3 ELICITING REQUIREMENTS

Requirements Elicitation (also called requirements gathering) combines elements of problem solving, elaboration, negotiation, and specification.

1) 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 worksheets, 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.

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request."

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 that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) 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.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything.

Collaborative requirements gathering ----Scenario: Campus Event Management App

Step 1: Identify Stakeholders

- **Students:** Potential users who will attend events.
- **Event Organizers:** People who will create and manage events.
- **University Administration:** They might have specific requirements or policies to comply with.
- **Developers:** The team responsible for building the app.

Step 2: Conduct Initial Meetings

- Organize a kickoff meeting with all stakeholders to introduce the project and its goals.
- Use this meeting to explain the importance of gathering accurate requirements and how it will impact the final product.

Step 3: Use Collaborative Techniques

- Brainstorming Session
- Surveys and Questionnaires
- Workshops and User Stories.

Step 4: Document and Validate Requirements

Step 5: Create a Prototype or Wireframes

Step 6: Iterate and Improve

2) Quality function deployment (QFD)

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, the mobile phone with standard features, but the developer adds few additional functionalities like voice searching, multi-touch screen etc. then the customer is more excited about that feature.

Although QFD concepts can be applied across the entire software process, QFD uses customer interviews and observation, surveys, and examination of historical data 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.

3) 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.

4) 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 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.

1.4 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 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. 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, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. Different people may play the role of each actor.

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. Secondary actors support the system so that primary actors can do their work. Once actors have been identified, use cases can be developed.

Jacobson suggests a number of questions 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?

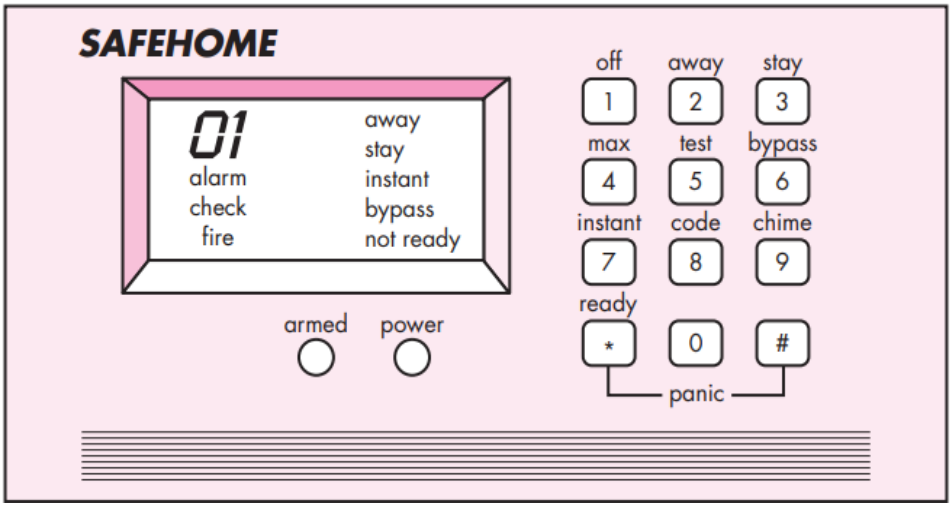
Basic SafeHome requirements, we define four ***actors***: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the SafeHome home security 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:

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

FIGURE 5.1

SafeHome control panel



1. The homeowner observes the SafeHome control panel as shown in Fig 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.

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 and reset itself for additional input. If the password is correct, the control panel awaits further action.

3. The homeowner select the keys in stay or away to activate the system.

✓ Stay activates only perimeter sensors (inside motion detecting sensors are deactivated).

✓ Away activates all sensors.

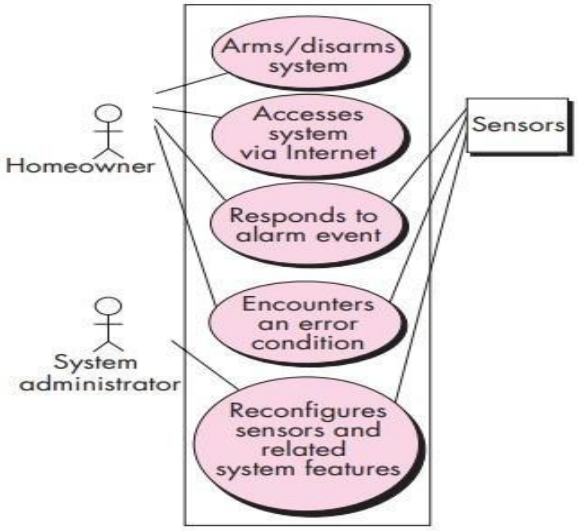
4. When activation occurs, a red alarm light can be observed by the homeowner. The basic use case presents a high-level story that describes the interaction between the actor and the system.

Cockburn provides Detailed Description of Use Case:

USE CASE	**InitiateMonitoring**
Primary actor	Home Owner
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.
When available:	First increment
Priority:	Essential, must be implemented
Frequency of use:	Many times per day
Channel to actor	Via control panel interface
Secondary actors:	Support technician, sensors
Channels to secondary actors:	1. Support technician: phone line 2. 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?

FIGURE 5.2
UML use case diagram for *SafeHome* home security function



Note:

The basic use case presents a high-level story that describes the interaction between the actor and the system.

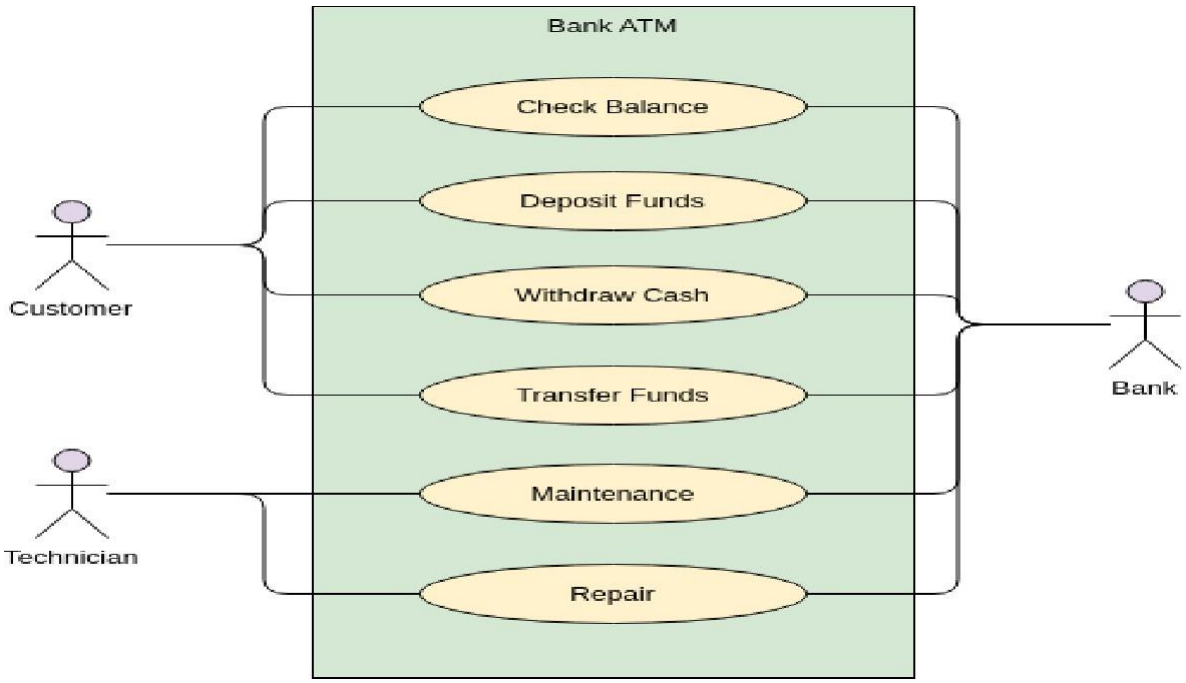
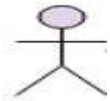


Fig 3.1-Use Case Diagram


----- Primary Actors-Who initiate/interact directly Ex: Customer and Technician

- ---------- Secondary Actor—Who supports the system Ex: Bank/Serve

------ Use Case (System) ----Functionality /service provided by the system.

- --------- Relation—It shows relation between actor and the system.

1.5 BUILDING THE REQUIREMENT MODEL

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The analysis model is a snapshot of requirements at any given time.

3.5.1 Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most requirements models.

Scenario Based Elements: The system is described from the user's point of view using a scenario-based approach **Ex: Use Case diagrams and Activity diagrams.**

Class-based Elements: Each usage scenario implies a set of **objects** that are manipulated as an actor interacts with the system. These objects are categorized into **classes**—a collection of things that have similar attributes and common behaviors(operations). **Ex: Class diagram, Collaboration diagram.**

Behavioral Elements: In Software Engineering, the Behavioral Elements Model is a concept used to describe the dynamic behavior of a software system. It focuses on how the system's components interact with each other and with external entities to achieve specific tasks or behaviors. The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. This model indicates how the software will respond on occurrence of external event. **Ex: State diagram and Sequential Diagram.**

Flow-Oriented elements: Information is transformed as it flows through a computer- based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. **Ex: Data-Flow diagram, Control Flow diagram.**

1.5.1 Analysis Patterns

Analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler suggest two benefits that can be associated with the use of analysis patterns:

First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations.

Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems. Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use

1.6 NEGOTIATING REQUIREMENTS

The intent of negotiation is to develop a **project plan** that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team. The best negotiations strive for a **"win-win"** result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to realistic and achievable budgets and deadlines.

Boehm 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.

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

1.7 VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. 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?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

MODULE 2

CHAPTER 2—REQUIREMENT MODELLING SCENARIOS, INFORMATION AND ANALYSIS CLASSES

2.1 REQUIREMENT 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 you 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 (attributes and operations) 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.

- The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. **The analysis model is a snapshot of requirements at any given time.**

- 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.

- Throughout requirements modeling, primary focus is on **what, not how**. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what

interfaces are defined, and what constraints apply?

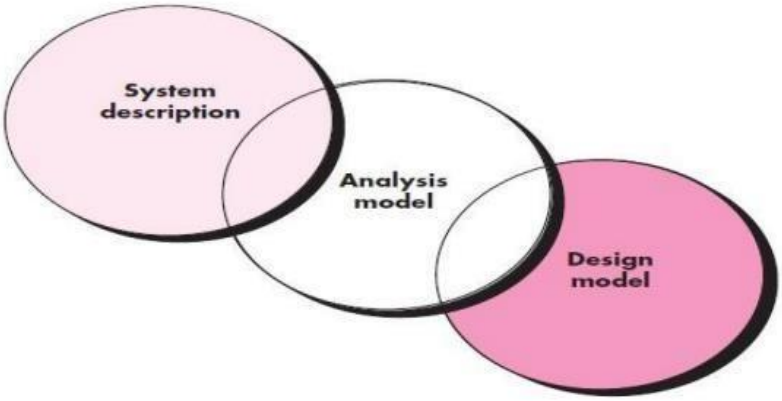


Fig : The requirements model as a bridge between the system description and the design model

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.

2.1.1 Analysis Rules of Thumb

Arlow and Neustadt 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.

- ***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

- **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.

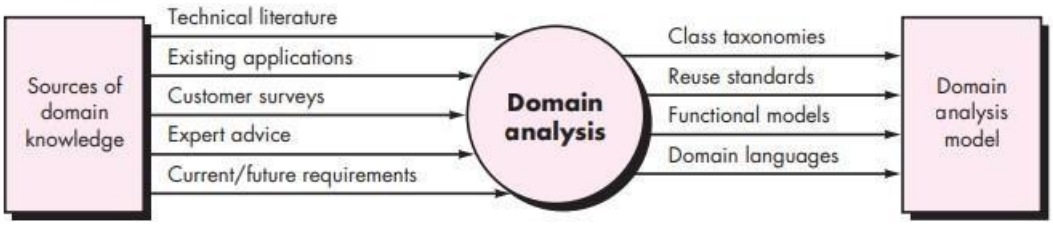
2.1.2 Domain Analysis

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices.

The goal of domain analysis is straightforward: to identify common problem-solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

FIGURE 6.2 Input and output for domain analysis



2.1.3 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.*

- 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.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

- ***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.

- **Flow-oriented elements** represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

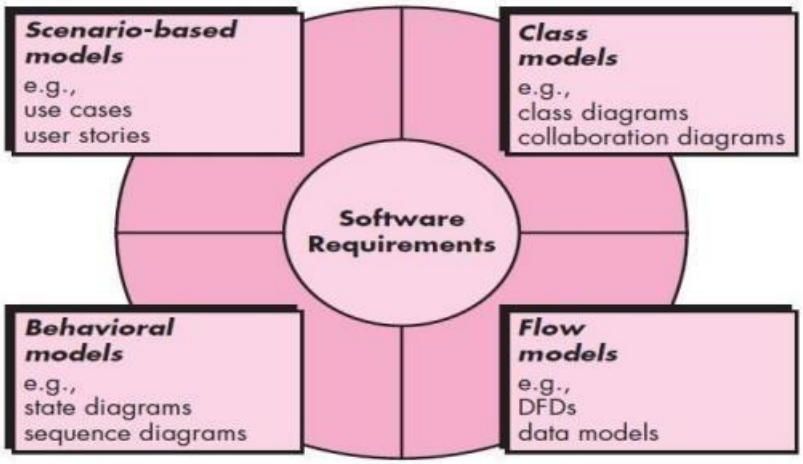


Fig : Elements of the analysis model

2.2 SCENARIO-BASED MODELLING

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

2.2.1 Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a “contract for behavior”, the “contract” defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself.

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

These are the questions that must be answered if use cases are to provide value as a requirement modeling tool.

- what to write about,
- how much to write about it,
- how detailed to make your description, and
- how to organize the description? To begin developing a set of use cases, list the functions or activities performed by a specific actor.

1) Creating a Preliminary Use Case:

Home Surveillance Functions:

Actor: Home Owner

- Select Camera View
- Request Thumbnails from all cameras
- Display camera view in PC window
- Selectively record camera output
- Replay Camera output
- Access camera *surveillance* via Internet

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

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

*****Use cases of this type are sometimes referred to as primary scenarios***.**

2.2.2 Refining a Preliminary Use Case

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 (e.g., behavior that is invoked by some event outside the actor's control)? If so, what might it be?

Answers to these questions result in the creation of a set of **secondary scenarios.**

1. View thumbnail snapshots for all cameras.
2. No floor plan configured for this house.
3. Alarm condition encountered.

Cockburn recommends using a "**brainstorming**" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- 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.

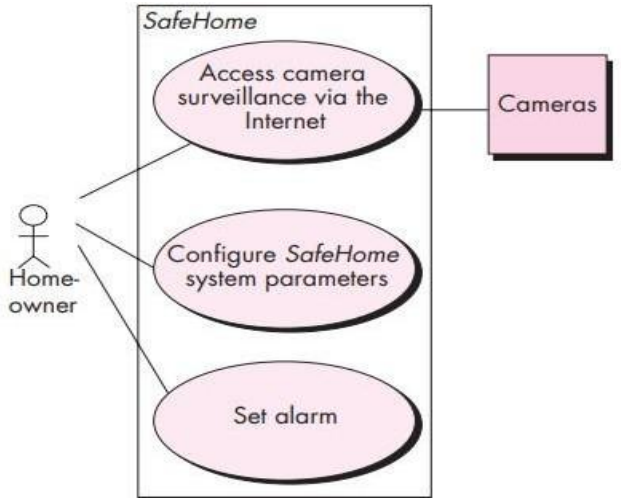
- Can poor system performance result in unexpected or improper user actions?

2.2.3 Writing a Formal Use Case

When a use case involves a critical activity or describes a complex set of steps with a significant number of exceptions, a more formal approach may be desirable.

The typical outline for formal use cases can be in following manner:

- 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



Preliminary Use-Case diagram for the Safe-Home Systems

However, **scenario-based modeling** is appropriate for a significant majority of all situations that you will encounter as a software engineer.

2.3 UML MODELS THAT SUPPLEMENT THE USE CASE

2.3.1 Developing an Activity Diagram

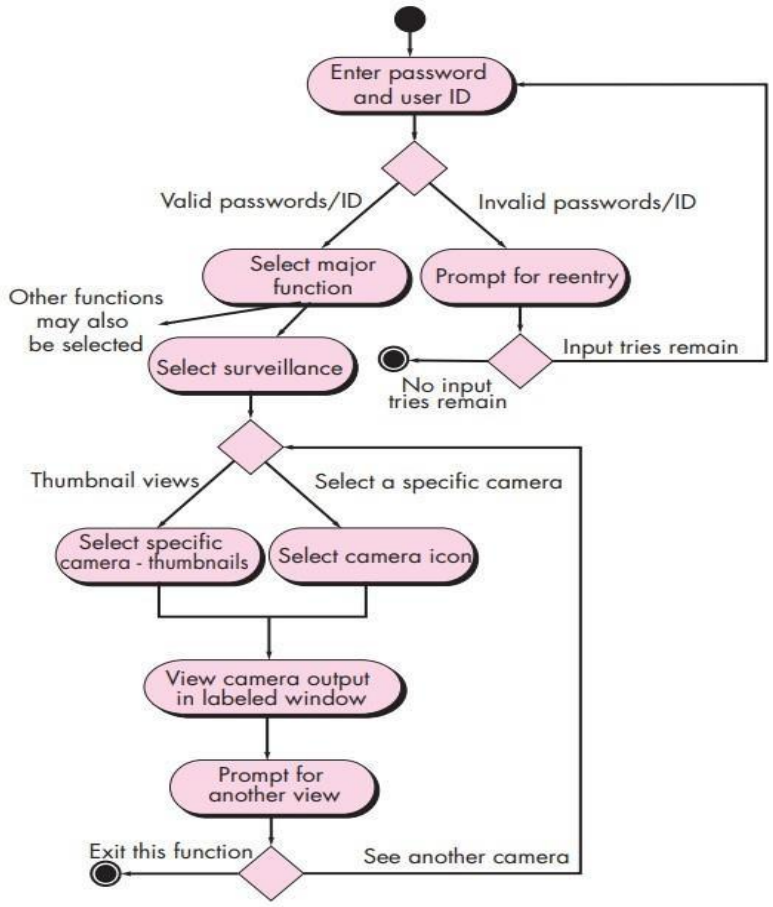
The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart,
An activity diagram uses:

- **Rounded rectangles** to imply a specific system function
- **Arrows** to represent flow through the system
- **Decision diamonds** to depict a branching decision.
- **Solid horizontal lines** to indicate that parallel activities are occurring.

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

FIGURE 6.5

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



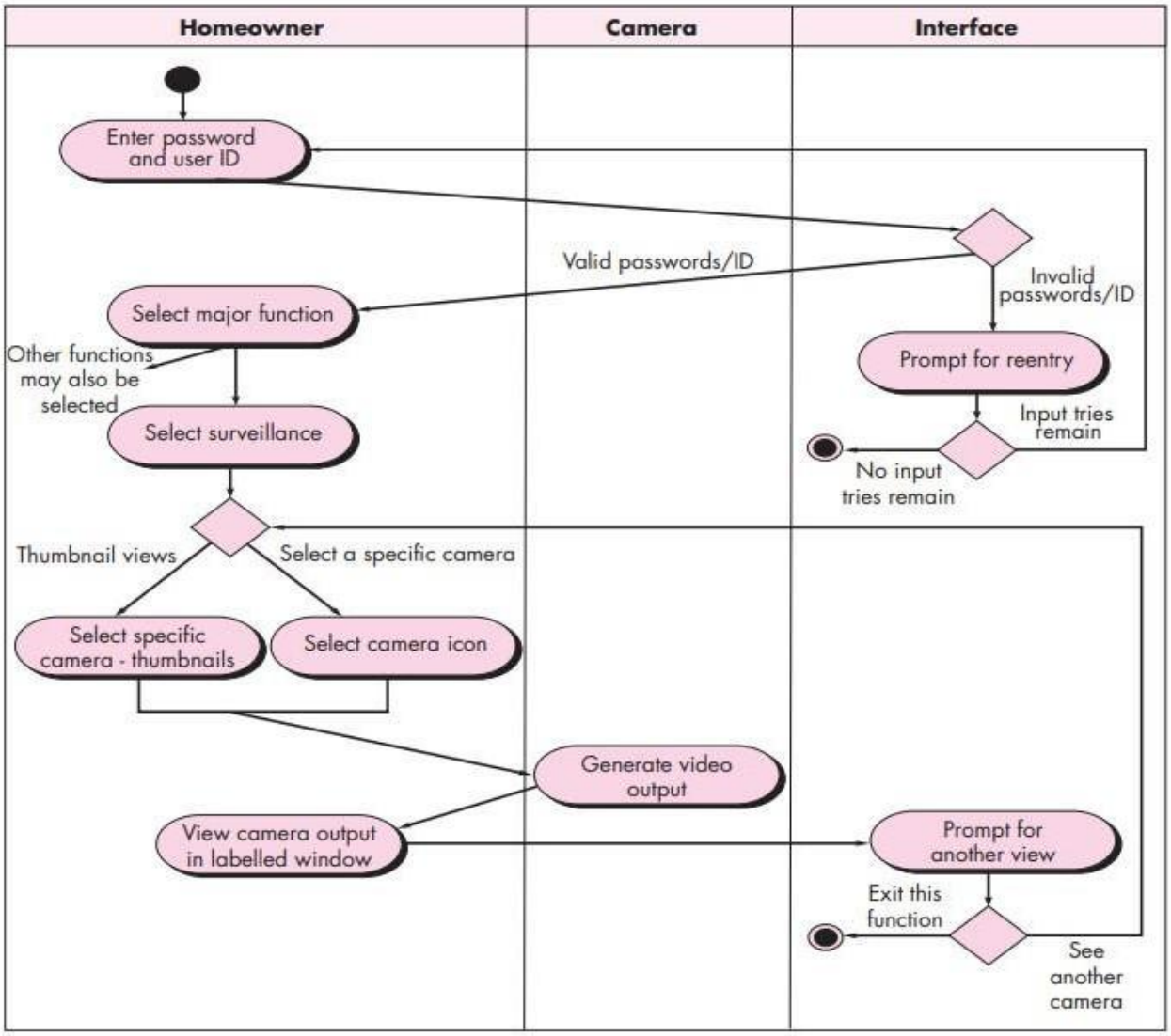
2.3.2 Swimlane Diagrams

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 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 6.5.

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



2.4 DATA MODELING CONCEPTS

Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow.

The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application.

The most widely used data Model by the Software engineers is **Entity-Relationship Diagram (ERD),** it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

2.4.1 Data Objects

A data object is a representation of composite information that must be understood by software. 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).

For example**, a person or a car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.

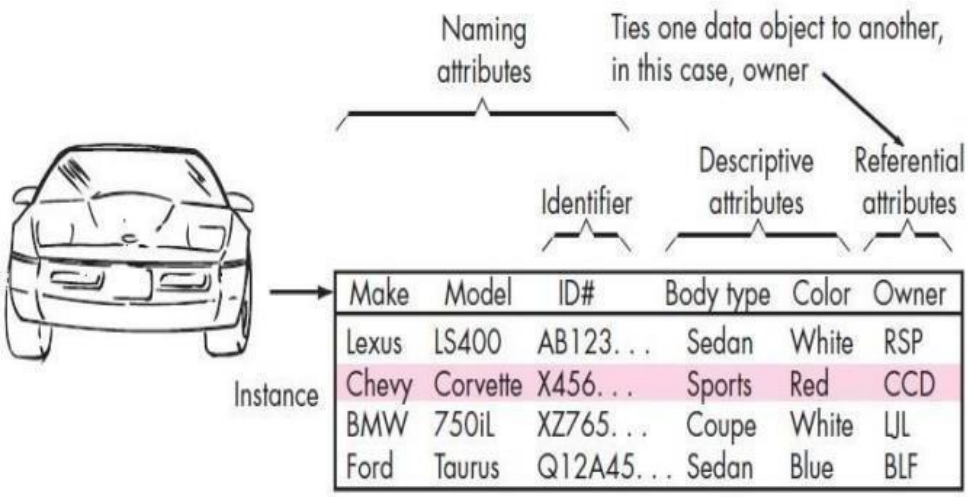


Fig : Tabular representation of data objects

2.4.2 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.

2.4.3 Relationships

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 following simple notation and relationships are 1) A person owns a car, 2) A person is insured to drive a car.

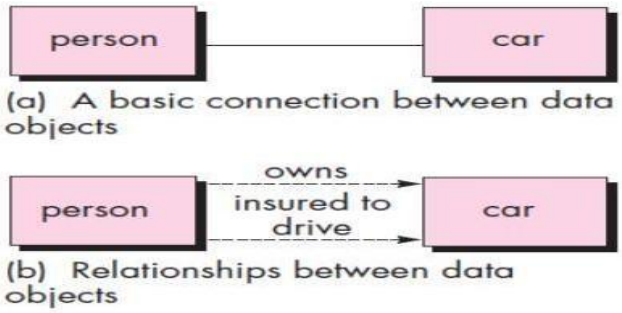


Fig : Relationships between data objects

2.5 CLASS-BASED MODELING

Class-based modeling represents 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.

The elements of a class-based model include classes and objects, attributes, operations, class responsibility collaborator (CRC) models, collaboration diagrams, and packages.

2.5.1 Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "**grammatical parse**" on the use cases developed for the system to be built.

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.

Coad and Yourdon suggest six selection characteristics that should be used as you consider 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.

2.5.2 Specifying Attributes

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class.

2.5.3 Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event.

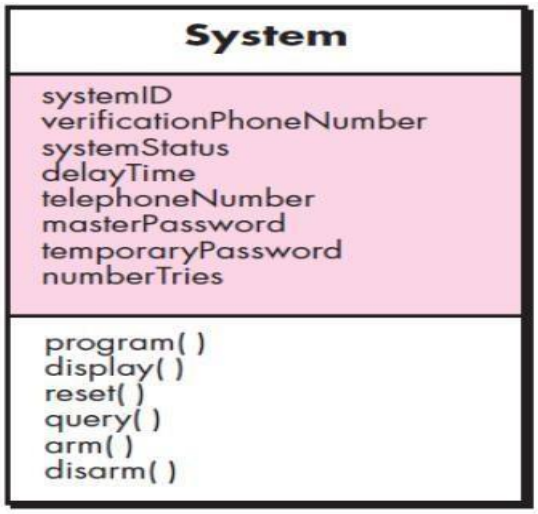
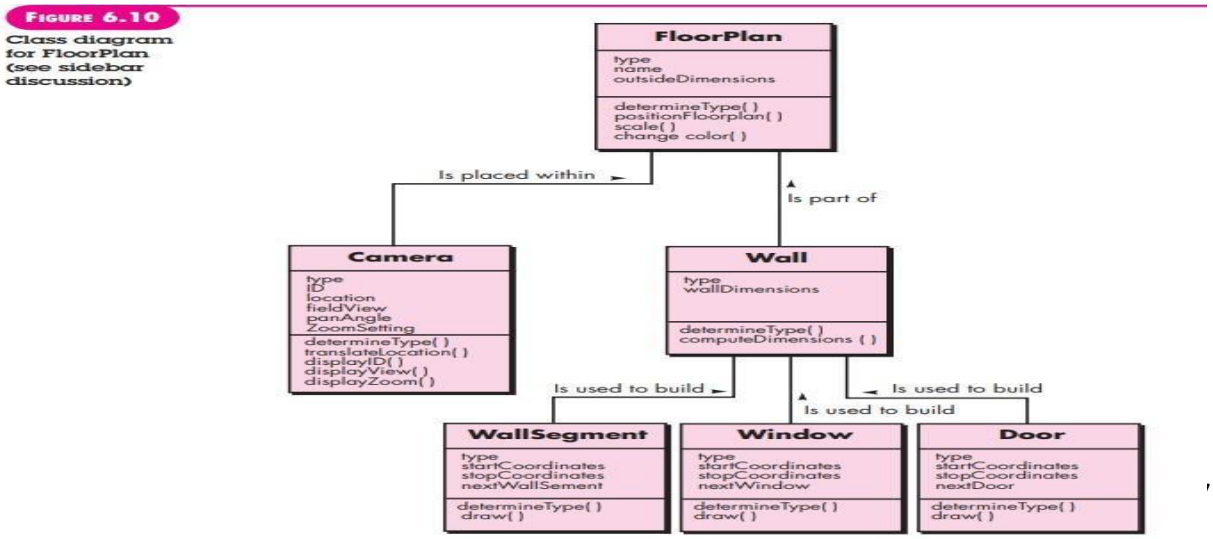


Fig : Class diagram for the system class



2.5.4 Class-Responsibility-Collaborator (CRC) Modeling

Class-responsibility-collaborator (CRC) modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

Ambler 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. 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. i.e., 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.

A simple CRC index card is illustrated in following figure.

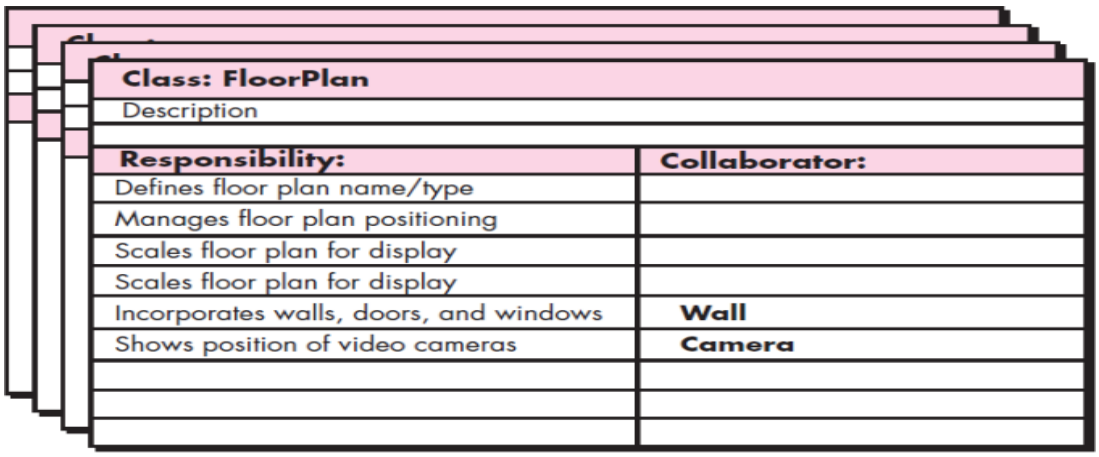


Fig : A CRC model index card

Classes: The taxonomy of class types can be extended by considering the following categories:

- **Entity classes**, also called model or business classes, are extracted directly from the statement of the problem. 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 that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

- **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: Wirfs-Brock and her colleagues suggest 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.

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.

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

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).

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 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 (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

2.5.5 Associations and Dependencies:

An association defines a relationship between classes. An association may be further defined by indicating ***multiplicity***.

Multiplicity defines how many of one class are related to how many of another class. 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 you to define a special modeling element whose semantics are custom defined. In UML. Stereotypes are represented in double angle brackets (e.g., **<<stereotype>>**).

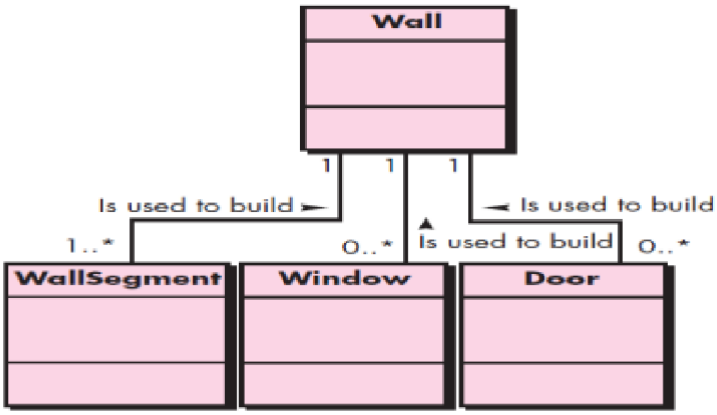


Fig : Multiplicity

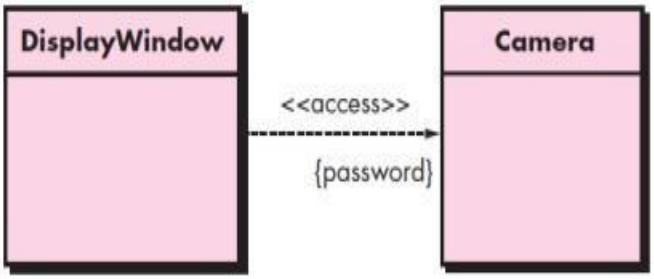


Fig : Dependencies

2.5.6 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.

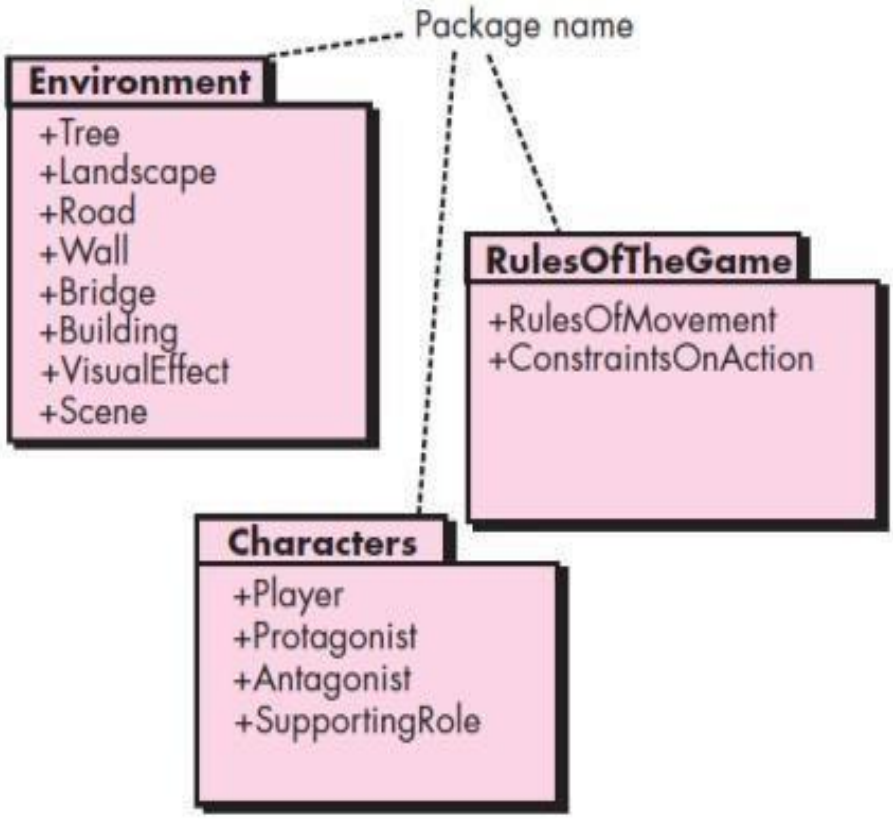


Fig : Packages