

## **MODULE- 2**

### **REQUIREMENTS ENGINEERING**

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

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

**Elicitation:** 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.

Christel and Kang identify a number of problems that are encountered as elicitation occurs.

- Problems of scope: The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system 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 untestable.
- Problems of volatility: The requirements change over time.

To help overcome these problems, you must approach requirements gathering in an organized manner.

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

**Negotiation:**

- ✓ It is common for customers and users to request more than what can be accomplished with the available business resources.
- ✓ It is also quite common for different customers or users to propose conflicting requirements, each insisting that their version is "essential for our specific needs."
- ✓ We have to reconcile these conflicts through a process of **negotiation**.
- ✓ Customers, users, and other stakeholders are asked to **rank requirements** and then discuss conflicts in priority.
- ✓ Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

**Specification:**

- ✓ 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.
- ✓ A standard template should be developed for technical specifications. Using a consistent template will help ensure requirements are presented in a clear and understandable manner.

**Validation:**

- ✓ The work products produced as a consequence of requirements engineering are assessed for quality during a validation step.



### 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 Wiegiers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at [www.processimpact.com/process\\_assets/srs\\_template.doc](http://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.

- ✓ 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 (unachievable) requirements.



### Requirements Validation Checklist

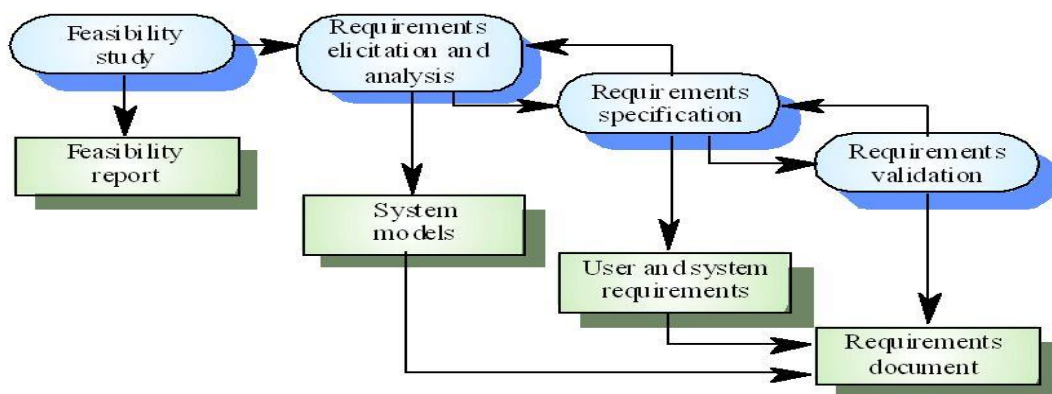
It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- 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?

### Requirements management:

- ✓ Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system.
- ✓ Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- ✓ Many of these activities are identical to the software configuration management (SCM) techniques.

## Requirement engineering



## **INITIATING THE REQUIREMENTS ENGINEERING PROCESS (GROUNDWORK)**

Steps to be followed:

1. Identifying Stakeholders
2. Recognizing Multiple Viewpoints
3. Working toward Collaboration
4. Asking the First Questions

### **Identifying Stakeholders:**

- A stakeholder is “anyone who benefits in a direct or indirect way from the system which is being developed”
- They are 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.
- At inception, you should create a list of people who will contribute input as requirements are elicited.

### **Recognizing Multiple Viewpoints**

- In the process of developing a new system, various stakeholder groups, including marketing, business managers, end users, software engineers, and support engineers, each have unique requirements based on their specific interests and needs.
- These requirements can range from marketable features and budget compliance to user-friendly interfaces, technical infrastructure, and maintainability.
- As these diverse and sometimes conflicting requirements are collected during the requirements engineering process, they need to be effectively categorized and managed.
- This categorization helps in identifying inconsistencies and conflicts, allowing decision-makers to select a consistent and unified set of requirements for the system's development.

## Working toward Collaboration

- In a software project involving multiple stakeholders, differing opinions on requirements are common. Effective collaboration among stakeholders and software engineers is crucial for a successful system.
- The role of a requirements engineer is to pinpoint areas where stakeholders agree (commonality) and areas of disagreement or inconsistency.
- Handling conflicting requirements, where one stakeholder's needs clash with another's, is particularly challenging.
- Collaboration does not necessarily involve making decisions by committee; rather, stakeholders may provide input on requirements, but a strong project champion (like a business manager or senior technologist) often makes the final decisions on which requirements are included in the project.

## Asking the First Questions

- 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:

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?

## ELICITING REQUIREMENTS

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

### Collaborative Requirements Gathering

Different approaches to collaborative requirements gathering have been proposed. And the following are 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.
- **Quality Function Deployment**
- Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD “concentrates on maximizing customer satisfaction from the software engineering process”
- To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.
- QFD identifies three types of requirements:

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

**Expected requirements:** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.

Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

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

### Usage Scenarios

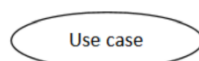
- Gathering system requirements helps create a vision of its functions, but diving into technical work requires understanding how users will use these functions.
- To do this, developers and users make Usage scenarios ( Use Case) showing how the system will be used.

### Elicitation Work Products

- The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built.
- For most systems, the work products include
  - A statement of need and feasibility.
  - A bounded statement of scope for the system or product.
  - A list of customers, users, and other stakeholders who participated in requirements elicitation.
  - A description of the system's technical environment.
  - A list of requirements (preferably organized by function) and the domain constraints that apply to each.
  - A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
  - Any prototypes developed to better define requirements

## DEVELOPING USE CASES

- **Use case** is the description of set of sequences of actions.
- It is graphically represented as an ellipse and labelled with the name of the use case.
- Use case represents an action performed by a system





- **Actor:** An actor represents a coherent set of roles that users of use case can play while interacting with use cases.
- An actor represents a role that a human, hardware device or another system plays when it communicate with the system.
- It is represented with the stick man notation.
- **Primary actors** interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- **Secondary actors** support the system so that primary actors can do their work.



### System Boundary:

- System boundary specifies the scope of an application in order to specify functionality.
- It indicates what the system includes and what it omits.
- System boundary groups together logically related things.
- It separates use cases and actors involved in the system.
- System boundary is shown with a box in a use case diagram.



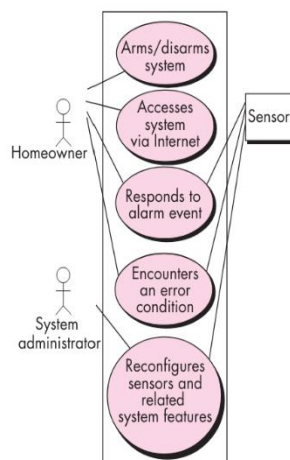
### Include Relationship:

- The Include Relationship indicates that a use case includes the functionality of another use case.
- It is denoted by a dashed arrow pointing from the including use case to the included use case.

This relationship promotes modular and reusable design.

**FIGURE 5.2**

UML use case diagram for SafeHome home security function



## BUILDING THE REQUIREMENTS MODEL

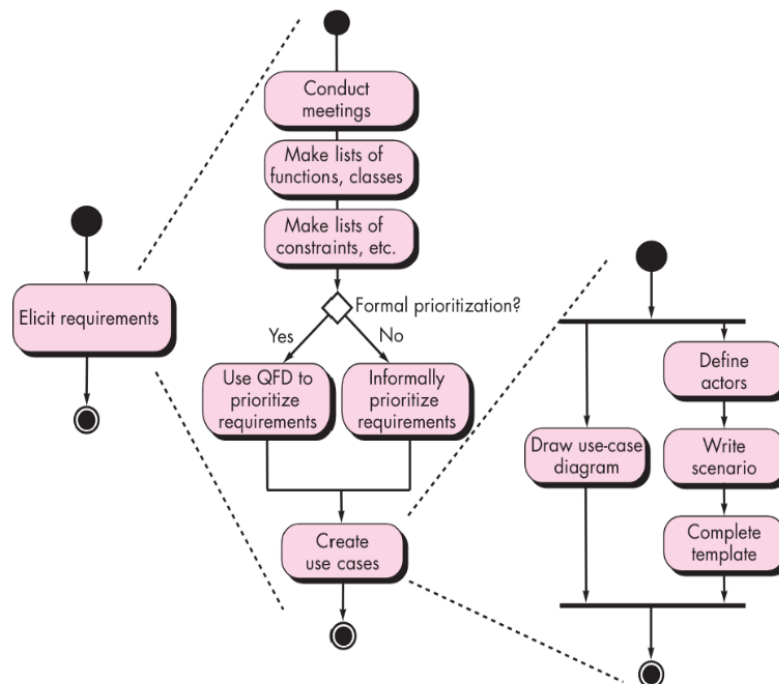
- The intent of the **analysis model** is to provide a description of the required informational, functional, and behavioural domains for a computer-based system.
- The model changes dynamically as we learn more about the system to be built, and other stakeholders understand more about what they really require.

### Elements of the Requirements Model

**Scenario-based elements:** The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use-case diagrams evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other

**FIGURE 5.3**

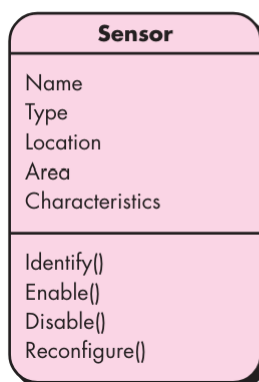
UML activity diagrams for eliciting requirements



modeling elements. Figure 5.3 depicts a UML activity diagram for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

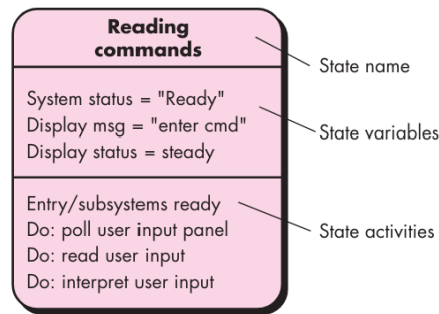
**Class-based elements:** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviours. For example, a UML class diagram can be used to depict a Sensor class for the SafeHome security function (Figure 5.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., identify, enable) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes.

**FIGURE 5.4**  
Class diagram  
for sensor



**Behavioral elements.:** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior. 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. A state is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event. To illustrate the use of a state diagram, consider software embedded within the SafeHome control panel that is responsible for reading user input. A simplified UML state diagram is shown in Figure 5.5

**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. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light

**FIGURE 5.5**UML state  
diagram  
notation

a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity.

## Analysis Patterns

- Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain.
- These analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.
- **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.
- Information about an analysis pattern (and other types of patterns) is presented in a standard template.

## Negotiating Requirements:

- In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market.
- The intent of this 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 (as a member of the software team) win by working to realistic and achievable budgets and deadlines.
- Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:
  1. Identification of the system or subsystem’s key stakeholders.
  2. Determination of the stakeholders’ “win conditions.”
  3. Negotiation of the stakeholders’ win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

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

## INFO



### 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.

### 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?

## **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 behaviours does the system exhibit, what interfaces are defined, and what constraints apply?

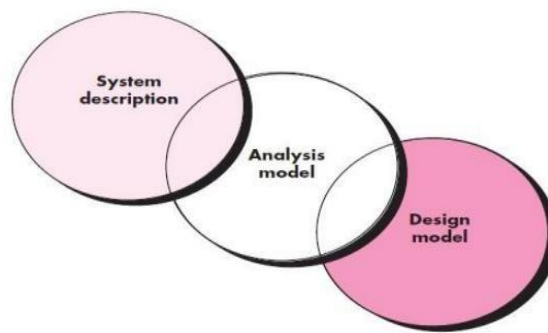


Fig : The requirements model as a bridge between the system description and the design model

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**

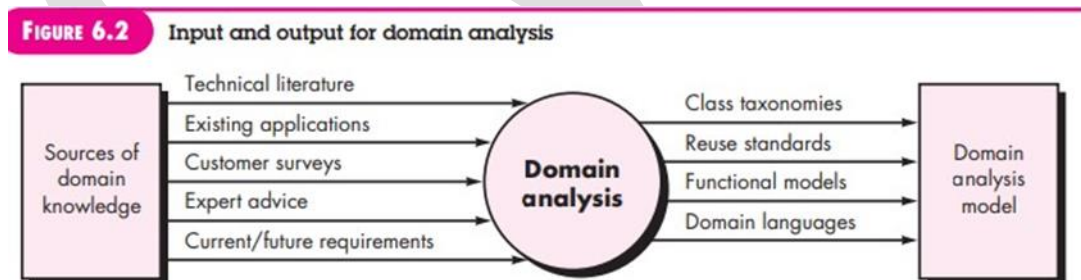
Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

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 behavior of the system.

3. 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.
4. 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.
5. Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model
6. 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.



### 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.

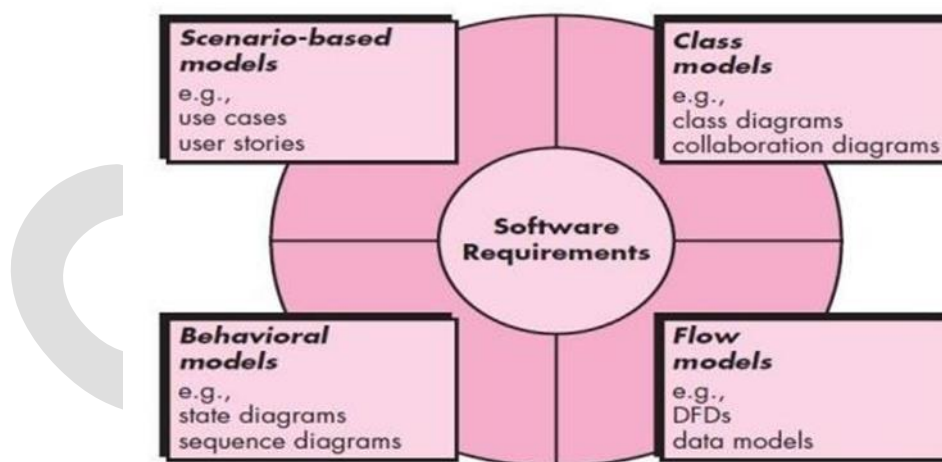


Fig : Elements of the analysis model

## 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:

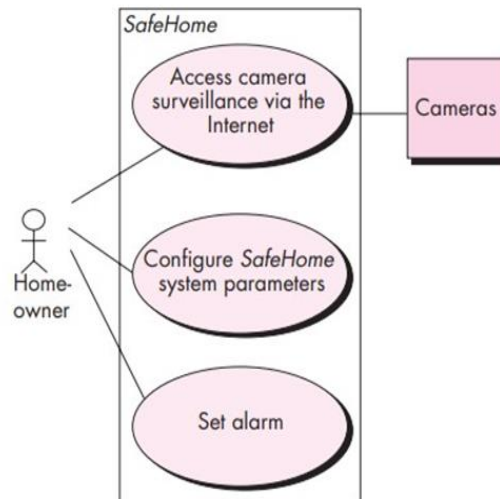
- Are there cases in which some “validation function” occurs during this use case? This implies that validation function is invoked and a potential error condition might occur.
- Are there cases in which a supporting function (or actor) will fail to respond appropriately? For example, a user action awaits 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,

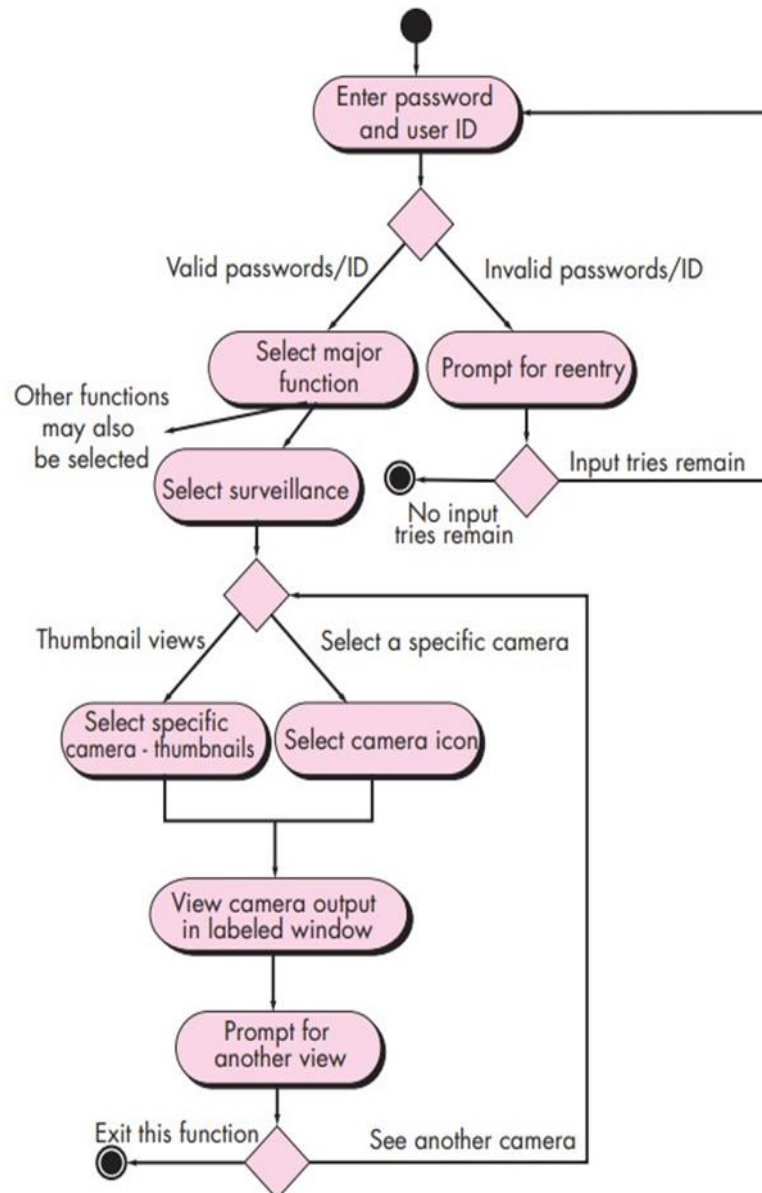
An activity diagram uses:

- **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**

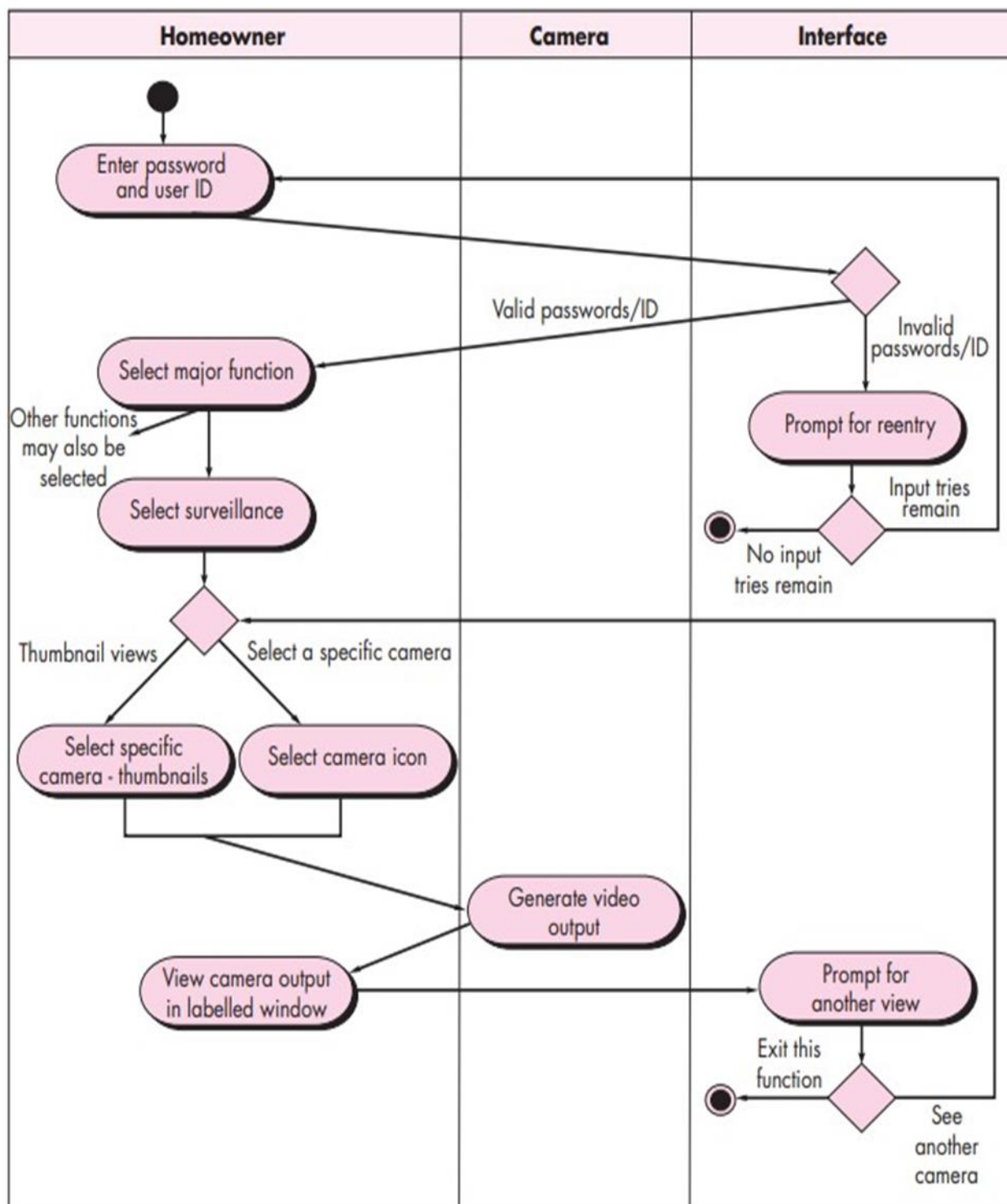
Activity diagram for Access camera surveillance via the Internet—display camera views function.



### 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



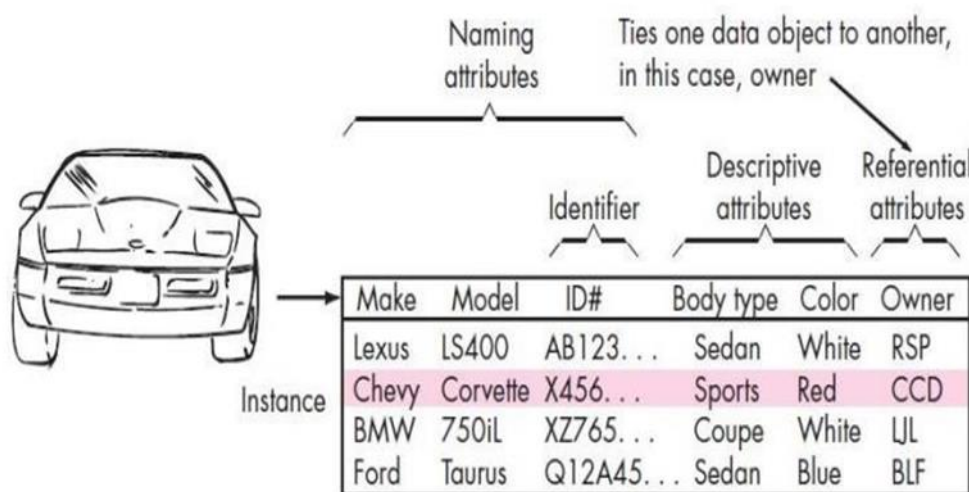
## 2.4 DATA MODELING CONCEPTS

- 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

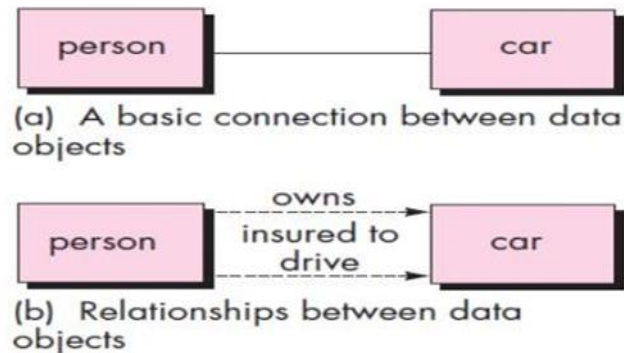
- Data attributes define the properties of a data object and take on one of three different characteristics.

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.

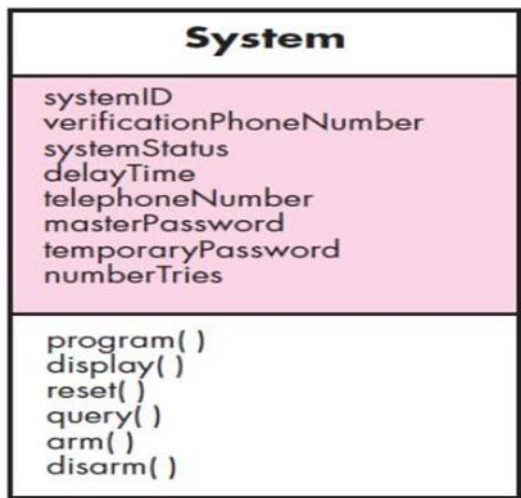
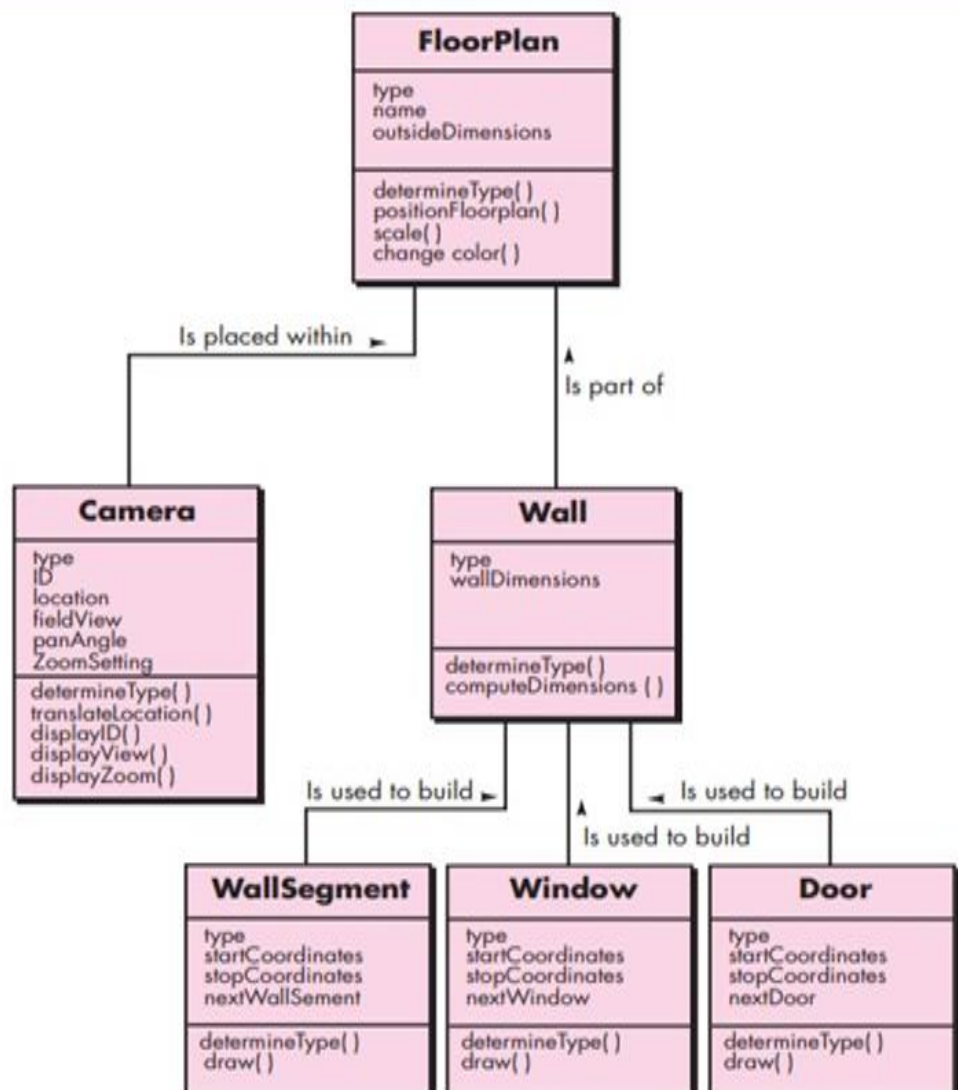


Fig : Class diagram for the system class

**FIGURE 6.10**  
Class diagram  
for FloorPlan  
(see sidebar  
discussion)



### 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.

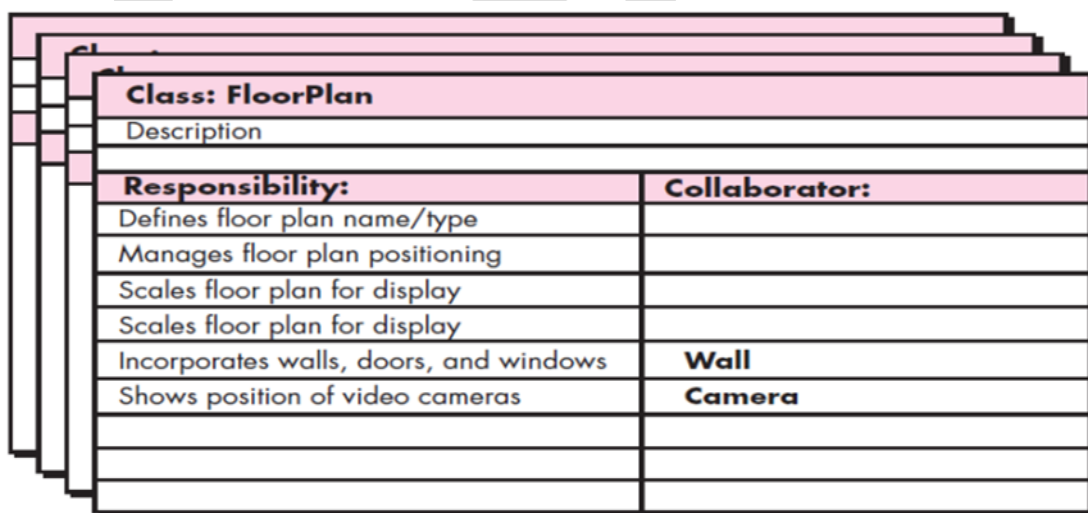
Ambler describes CRC modeling in the following way:

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.



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

**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:

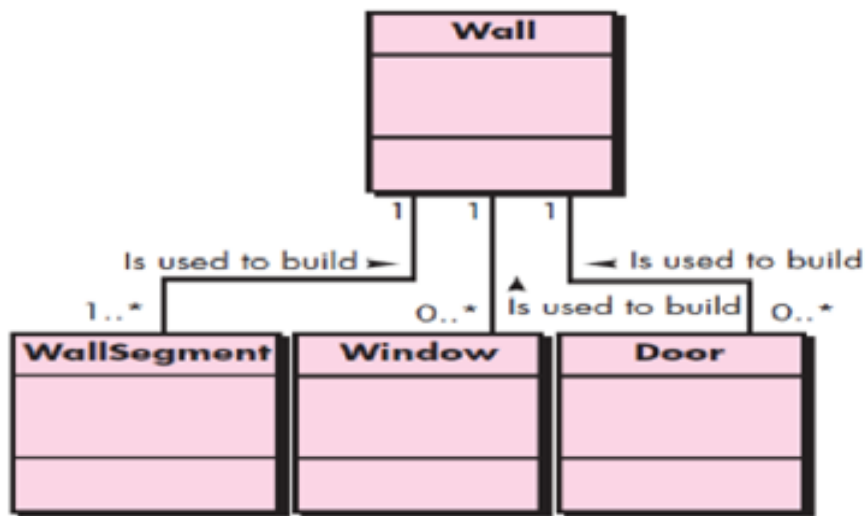
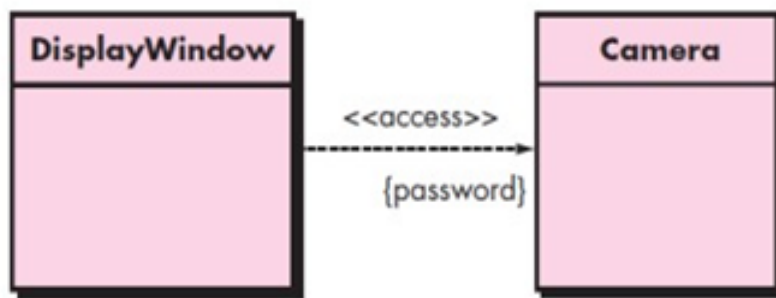
- A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
- A class can collaborate with other classes.

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.

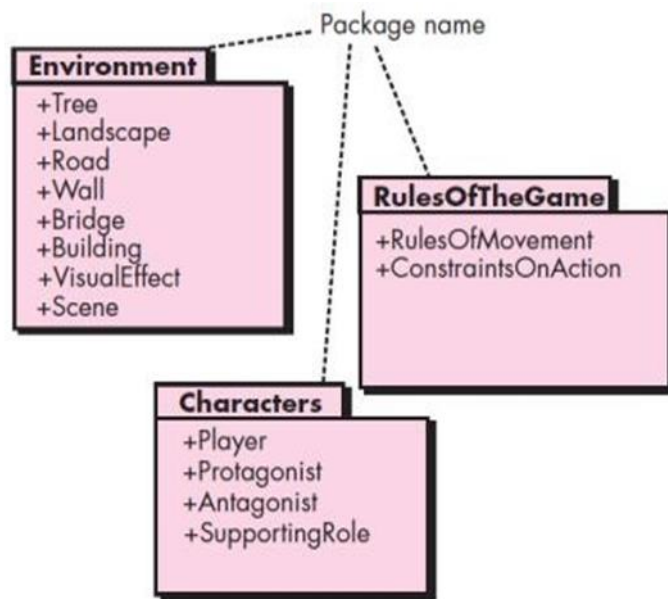
#### 2.5.5 Associations and Dependencies:

- 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**