# SUBJECT: Software Engineering & Project Management (BCS501)

## MODULE-2 Understanding Requirements, Requirements Modeling Scenarios, and Requirement Modeling Strategies

***Syllabus:*** ***Understanding Requirements****: Requirements Engineering, Establishing the ground work, Eliciting Requirements, Developing use cases, Building the requirements model, Negotiating Requirements, Validating.*

***Requirements. Requirements Modeling Scenarios, Information and Analysis classes:*** *Requirement Analysis, Scenario based modeling, UML models that supplement the Use Case, Data modeling Concepts, Class-Based Modeling.*

***Requirement Modeling Strategies****: Flow oriented Modeling, Behavioral Modeling*

## Software Requirements

Software requirement is a condition needed by a user to solve a problem or to achieve an objective.

**Types of Requirements**

- **Functional requirements:** Statements of services, how the system should be particular inputs, and what functionalities is to provided. Functional requirements are not concerned with how these functions are to be achieved, just what is to achieved.
- **Non-Functional requirements:** Deals with attributes or properties of the software rather than functions. We include here aspects of the software such as its performance, its usability, any safety aspects and a range of other attributes.

## Requirement Engineering

**Definition:** The process of collecting requirements from the customer, understanding , analysis and documenting them is called Requirement engineering.

Requirement engineering constructs a bridge for design and construction. Requirement engineering is the discipline concerned with understanding the externally imposed conditions on a proposed computer system, determining what capabilities will meet these imposed conditions and documenting those capabilities as the software requirements for the computer system. The process of establishing the services that the customer requires from system and the constraints under which it operates and is developed

**Importance of Requirements**

- Making design decisions without understanding all the constraints of the system to be developed results in a system which fails to meet customers' expectations.
- Costs of correcting errors increases as the design process advances.
- An error detected in the maintenance phase is 20 times as costly to fix an error detected in the coding stage.

**Requirements Perspective**

- **User requirements:** Statements in natural language plus diagrams of the services the system provides and its operational constraints written for customers
- **System requirements:** A structured document setting out detailed descriptions of the systems' functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

## Requirements Engineering Tasks: Seven Distinct Tasks

- **Inception**
- **Elicitation**
- **Elaboration**
- **Negotiation**
- **Specification**
- **Validation**
- **Requirements management**

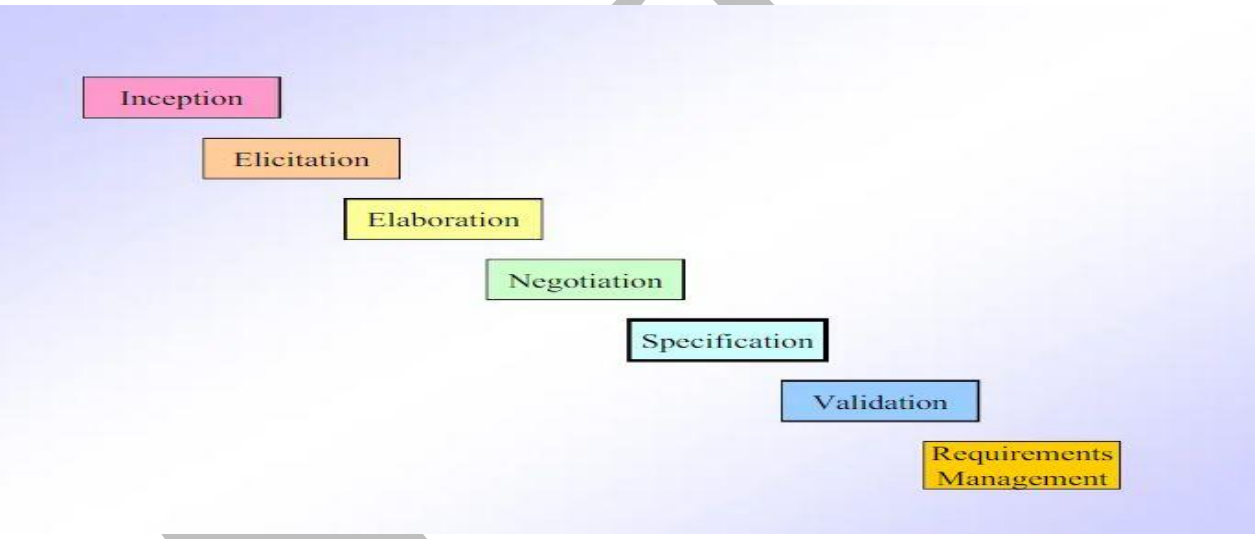



**Fig: 1.1 Requirement engineering tasks**

1) **Inception:** Inception means beginning, It is always problematic to the developer that from where to start. The customer and developer meet and they decide overall scope and nature of the problem.
   In this phase they understand
   a. They understand basic details, aim, and goal of the project and find out the solution.
   b. They identify stakeholders and who want the solution.
   c. They understand the nature of the solution.
   d. Enhance collaboration between customer and developer.

2) **Elicitation Task:** Elicitation means to draw out the truth or reply from anybody. In relation with requirement engineering, elicitation is a task that helps the customer to define what is required.

   **Following are some problem occur while deciding fixed set of requirements**
   - Problem of scope: Unnecessary details by the customer may confuse developer instead of giving of overall system objectives.

- Problems of understanding: Sometimes both customer as well as developer has poor understanding of what is needed, capabilities and limitations of the computing environments, etc.
- Problem of volatility: Volatility means change from one state to another state. The customer's requirements may change time to time.

**3) Elaboration Task:** Elaboration means 'to work out in detail'. During elaboration, the software engineering takes the information obtained during inception and elicitation and begins to expand and refine it.

Elaboration focuses on developing a refined technical model of software that will include functions, features, and constraints

**4) Negotiation Task:** Negotiation means 'discussion on financial and other commercial issues'.

This phase will involve the negotiation between what user actual expects from the system and what is actual feasible for the developer to build. But based on the other aspects and feasibility of a system the customer and developer can negotiate on the few key aspect of the system.

**5) Specification Task:** The specification is the final work product produced by the requirement engineers. It serves as the foundation for subsequent software engineering activities.
It describes the function and performance of a computer based system and the constraints that will govern its development. It formalizes the informational, functional, and behavioral requirements of the proposed software in both a graphical and textual format.

6) **Validation Task:** During validation the work products produced as a result of requirements engineering are assessed for quality.
- It check the errors and debugging in the final work product or SRS document.
- It checks all the requirements have been stated and met correctly as per stakeholders.
- It checks any missing information or to add information.

7) **Requirements Management Task:** During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds.

Each requirement is then placed into one or more traceability tables.
These tables may be stored in a database that relates features, sources, dependencies, subsystems, and interfaces to the requirements

## Establishing groundwork for understanding of software requirements

Establishing groundwork create a strong foundation for understanding the project's needs and objectives. It involves setting up the necessary processes, communication, and environment to ensure that everyone involved in the project including stakeholders, customers, end users, and software engineers has a clear understanding of what the software should do.

The steps required to establish the ground work for an understanding of software requirements, are

1. **Identifying stakeholders**
2. **Recognizing multiple view points**
3. **Working toward collaboration**
4. **Asking first questions**

**Identifying stakeholders: A** stakeholder is "anyone who benefits in a direct or indirect way from the system which is being developed".

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

Identify all the stakeholders involved in the project and ensure they are engaged from the start. Each stakeholder has a different view of the system, achieves different benefits and is open to different risks if the development effort should fail.

**Recognizing multiple viewpoints:** Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. Each of these voters will contribute information to the requirements engineering process. As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

It is important categorize all stakeholder information including inconsistent and conflicting requirements in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

**Working towards collaboration:** There are different opinions are involved by different stakeholders on the set of requirements. Customer must work together with software development team to create successful system.

Requirements engineer has to identify areas of commonality requirement as well as conflict requirements. Finally project head & requirement engineer has to decide which requirement are accepted.

**Asking the first questions:** 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.

## Eliciting requirements

Requirements elicitation is also called requirements gathering.It 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. It will perform various task

- Collaborative requirements gathering
- Quality function deployment
- Usage scenarios
- Elicitation work products

**1) Collaborative requirement gathering**

Collaborative requirements gathering is performed by following

- Meetings are conducted and attended by both software engineers and others 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" 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.
- 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.
- Others would contribute to this narrative during the requirements gathering meeting and considerably more information would be available.
- 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 list 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 came up during the discussion, but not deleting anything.
- In many cases, an objective or service described on a list will require further explanation. To accomplish this, stakeholders develop mini-specifications for entries on the lists.

**2) 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".

QFD identifies three types of requirements

1) Normal requirements
2) Expected requirements
3) Exciting requirements.

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

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

c) **Exciting requirements**: These features go beyond the customers' 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., multi touch screen, visual voice mail) that delight every users of the product.

**3) Usage Scenarios:** As requirements are gathered, an overall vision of system functions and features begins to materialize. It is difficult to move into 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 will vary depending on the system, but should include one or more of the following items.

- 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** (organized by function) and the domain constraints that applies to each.
- A set of preliminary usage scenarios (in the form of use cases) that provide insight into the use of the system or product under different operating conditions.

## Developing use cases

A use case describes how a system responds to requests from its stakeholders. Captures a contract of the system's behavior under various conditions. Focuses on interactions between the user (actor) and the system.

Use case tells a stylized story about how an end user interacts with the system under a specific set of circumstances. The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation.

Regardless of its form, a use case depicts the software or system from the end user's point of view.

Use cases are descriptions of the functionality of a system from user perspective.

- Depicts the behavior of the system, as it appears to an outside user.
- **Describe the functionality and users (actors) of the system.**
- Shows the relationships between the actors that use the system, the use cases (functionality) they use, and the relationship between different use cases.
- **Document the scope of the system.**
- Illustrate the developers understanding of the users requirements.

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. Every actor has one or more goals when using the system. Since 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 .

Primary actors work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.

**Once actors have been identified, use cases can be developed.**

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 change

Example: Safe Home security system

## Following template for detailed descriptions of use cases:

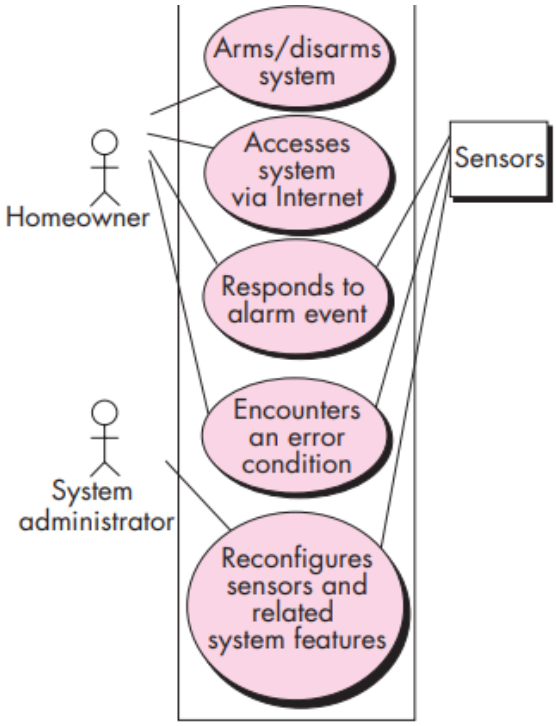
**Use case**: Managing the home security system of a smart home, where the user can arm, disarm, and control security features remotely.

- **Primary Actor**
- **Homeowner:** The main user who controls the home security features

- **Secondary Actor**
- **Smart Home Device:** The smart home hub or device that processes commands from the homeowner and controls the security system.
- **Security Service Provider:** A monitoring service that receives alerts if there is a security breach.
- **Actor Goals in the Smart Home Security Use Case**
- **Homeowner Goals**:
  - Arm or disarm the security system.
  - Monitor home security status remotely.
  - Configure security settings (e.g., set security zones).
- **Security Service Provider Goals**:
  - Receive alerts for security breaches.
  - Dispatch
- **Main Tasks**
  - Homeowner opens the smart home app.
  - Selects the security function.
  - Enters a password to verify identity.
  - Arms or disarms the security system
- **Exception**
  - If the homeowner forgets the password, they cannot arm/disarm the system immediately.
  - The system might prompt the homeowner to reset their password or verify their identity through additional steps
- **Variation**
  - The homeowner can also use voice commands to arm/disarm the system if the smart home device supports it.
- **Evolution of Requirements Gathering for Smart Home Security**
  - Initial iterations might identify only the **Homeowner** as the primary actor.
  - Later iterations could add secondary actors like a **Family Member** with restricted access or a **Guest User** with temporary permissions.
  - As requirements evolve, new scenarios like integrating with third-party security cameras or alarm systems might be added.

## Developing a High-Level Use-Case Diagram

UML use case diagram for *SafeHome* home security function




## Negotiating Requirements

Negotiating requirements in software engineering is a crucial process that involves discussion between stakeholders to agree on the software's needs, constraints, and priorities. The goal is to reach a mutual understanding and agreement on what the software should do and what features it should include. Effective negotiation helps ensure that all parties involved are aligned with the project's objectives and that the requirements are realistic and achievable within the given constraints.

The following activities has to be followed

1. **Identify Stakeholders**:
   - The first step is to identify all stakeholders involved in the project. Stakeholders may include customers, users, developers, project managers, business analysts, and any other individuals or groups affected by the software's outcome.
2. **Gather Initial Requirements:**
   - Collect initial requirements from the stakeholders to understand their needs and expectations. This step usually involves interviews, surveys, and brainstorming sessions to gather as much information as possible.
3. **Prioritize Requirements:**
   - Prioritize the requirements based on factors like business value, urgency, cost, technical feasibility, and risk. Stakeholders need to agree on which requirements are most critical to the project's success.
4. **Resolve Conflicts**: Discuss any disagreements and find compromises.
5. **Evaluate Feasibility**: Check if the requirements are realistic in terms of budget, time, and resources.
6. **Document Agreement:** Write down the final, agreed-upon requirements.
7. **Get Approval:** Ensure all stakeholders formally agree with the requirements.

These steps help align everyone's expectations and ensure the project moves

Importance of Negotiating Requirements

- **Improves Communication:** It fosters clear communication between stakeholders and development teams.
- **Manages Expectations:** Helps in aligning stakeholders' expectations with the project's goals.
- **Reduces Risks:** Minimizes the risk of project failure due to misunderstood or unrealistic requirements.
- **Enhances Flexibility:** Allows for changes in requirements as the project progresses, improving adaptability.

## Validating Requirements

Validating requirements is a critical process in software engineering that ensures the requirements gathered for the software project are complete, accurate, consistent, and aligned with the stakeholders' needs. The goal is to confirm that the requirements define what the software should do in a way that can be implemented and tested effectively.

Steps in Validating Requirements

1. **Review:** Check the requirements with stakeholders to ensure they're clear and complete.
2. **Consistency:** Make sure the requirements do not conflict with each other.
3. **Feasibility:** Verify that the requirements are realistic and achievable.
4. **Prototyping:** Create mock-ups to visualize how the requirements will look in the system.
5. **Stakeholder Approval:** Get confirmation from stakeholders that the requirements meet their needs.
6. **Sign-Off:** Get formal approval to confirm everyone agrees with the requirements.

Importance of Validating Requirements

- **Reduces Errors:** Helps identify and correct errors in the requirements early in the development process, reducing costly changes later.
- **Ensures Alignment:** Ensures that the software solution aligns with business objectives and stakeholders' needs.
- **Improves Quality:** Contributes to higher-quality software by providing a solid foundation for design, development, and testing.
- **Minimizes Rework:** Reduces the likelihood of rework due to incorrect or incomplete requirements, saving time and resources.

# REQUIREMENTS MODELING

## BUILDING REQUIREMENT MODEL

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow. However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system.

**Requirements Modeling:** Requirement analysis is significant and essential activity after elicitation. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, the under stability of the project may improve significantly.

1. **Elements of the Requirements Model**
2. **Analysis Patterns.**

### Elements of the Requirements Model:

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes.

1) **Scenario-based elements**: 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 modeling 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.

2) **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.
   - For example, a UML class diagram can be used to depict a Sensor class for the SafeHome security function 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.

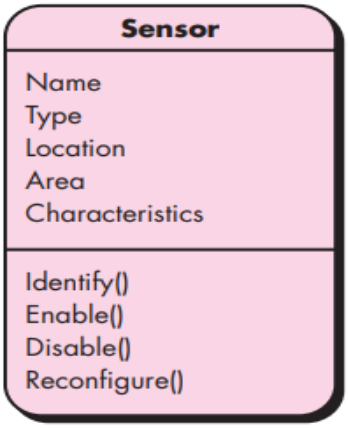



**Figure: Class diagram for sensor**

3) **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.

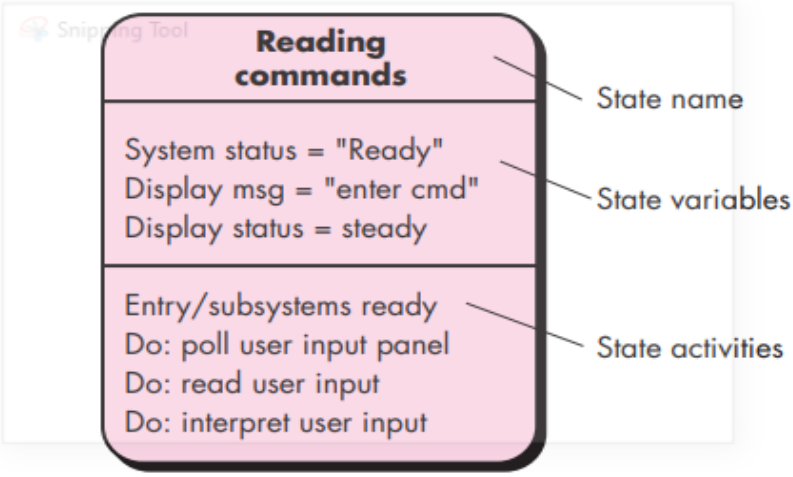



**Figure: UML State diagram**

4) **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.

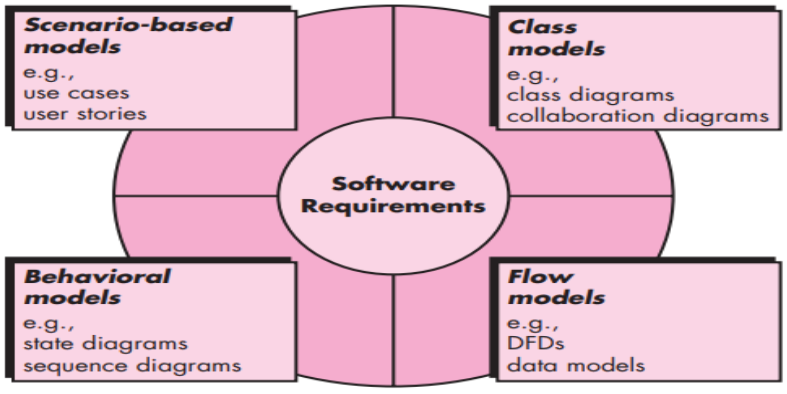



**Figure : Elements of analysis model**

Software requirements act as effective bridge to software design, each element of the requirements model presents the problem from a different point of view

1) 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) Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to affect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined.
3) Behavioral elements depict how external events change the state of the system or the classes that reside within it.
4) Finally, flow-oriented elements represent the system as information transform, depicting how data objects are transformed as they flow through various system functions.

## Analysis Patterns:

Few software projects begin 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. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them

### Analysis Rules of Thumb

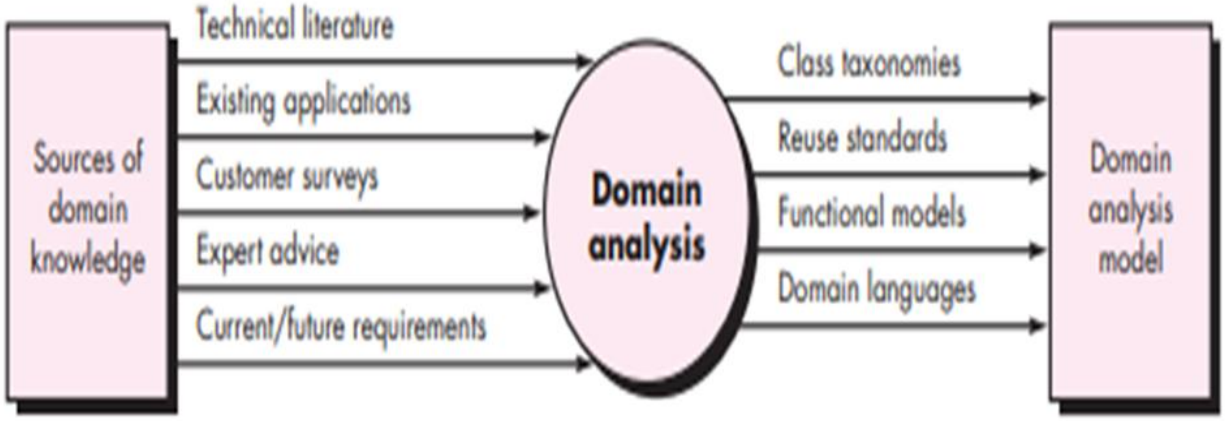
Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

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.
4) Minimize coupling throughout the system
5) Be certain that the requirements model provides value to all stakeholders
6) Keep the model as simple as it can be

## Domain Analysis

- Domain analysis is the process of identifying, analyzing, and specifying the common requirements, objects, classes, and patterns that are relevant within a particular domain a specific area of interest or field.
- The aim is to find reusable elements that can be applied to multiple projects within that domain.
- This approach helps create software that is more efficient, consistent, and adaptable.
- This improves time-to-market and reduces development costs.
- The role of the domain analyst is similar to that of a master toolsmith in a manufacturing environment:
- The tool smith designs and builds tools that are used by many people for similar tasks.

Similarly, the domain analyst identifies and defines analysis patterns, classes, and frameworks that can be used by software engineers working on various applications within the same domain.



## Requirements Modeling Approaches

There are two types of requirement modeling approaches

1) **Structured analysis**: considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
2) **Object-oriented analysis**: It focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

## Scenario Based Modeling

Scenario-based modeling is a technique used in software engineering to describe how users interact with the system in different situations. It involves creating scenarios or use cases that represent real-world interactions between users and the software. This approach helps to understand the system's behavior, identify user requirements, and ensure that the software meets user needs. Analysis modeling with UML begins with the creation of scenarios in the form:

- Use-cases diagrams
- Activity diagrams
- Swim lane diagrams

**Steps :**

1) Creating a Preliminary Use Case
2) Refining a Preliminary Use Case
3) Writing a Formal Use Case

1. **Creating a Preliminary Use Case:** Creating a preliminary use case involves identifying the functions or activities that a specific actor performs with the system. This process is one of the first steps in scenario-based modeling and helps in understanding how different users will interact with the software. 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 requirements modeling tool. (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description?

   Example: Safe home surveillance function

**The SafeHome home surveillance function (subsystem) identifies the following) that are performed by the homeowner actor:**

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

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

## 2. Refining a Preliminary Use Case:

A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case.

**Therefore, each step in the primary scenario is evaluated by asking the following questions:**

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
- Is it possible that the actor will encounter some other behavior at this point (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 that are part of the original use case but represent alternative behavior. For example, consider steps 6 and 7 in the primary scenario presented earlier:

6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house

## 3. Writing a Formal Use Case:

The informal use cases are sometimes sufficient for requirements modeling. However, 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. Diagrammatic representation can facilitate understanding, particularly when the scenario is complex

The scenario lists the specific actions that are required by the actor and the appropriate system responses. Exceptions identify the situations uncovered as the preliminary use case is refined. Additional headings may or may not be included and are reasonably self-explanatory

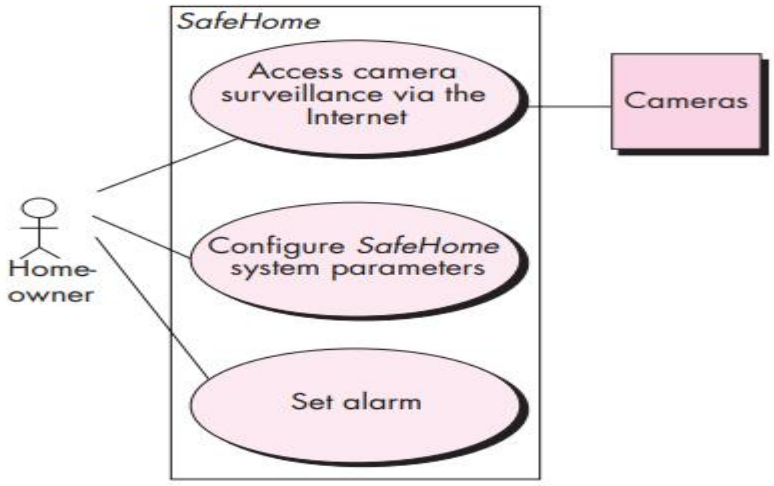



**Figure: Use case diagram for Safe Home system**

## UML Models that supplement the use case

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 and solid horizontal lines to indicate that parallel activities are occurring.
   It should be noted that the activity diagram adds additional detail not directly mentioned by the use case.

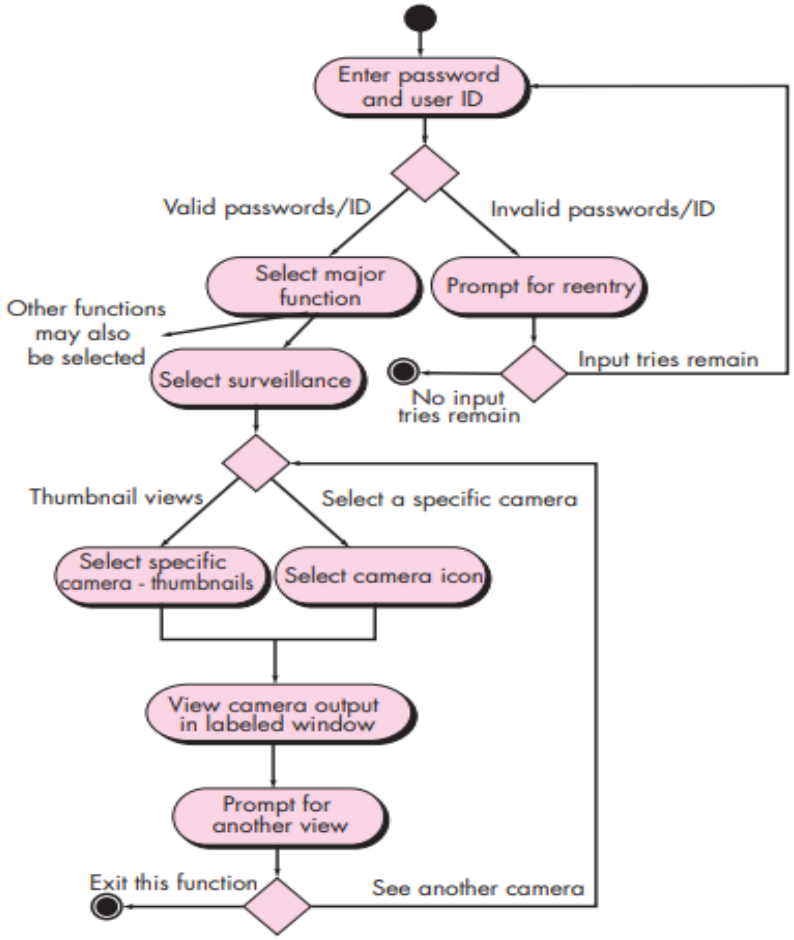



**Figure:** Activity diagram for Access camera surveillance via the Internet— display camera views function

## Swim lane Diagrams

The UML swim lane diagram is a useful variation of the activity diagram and allows 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.

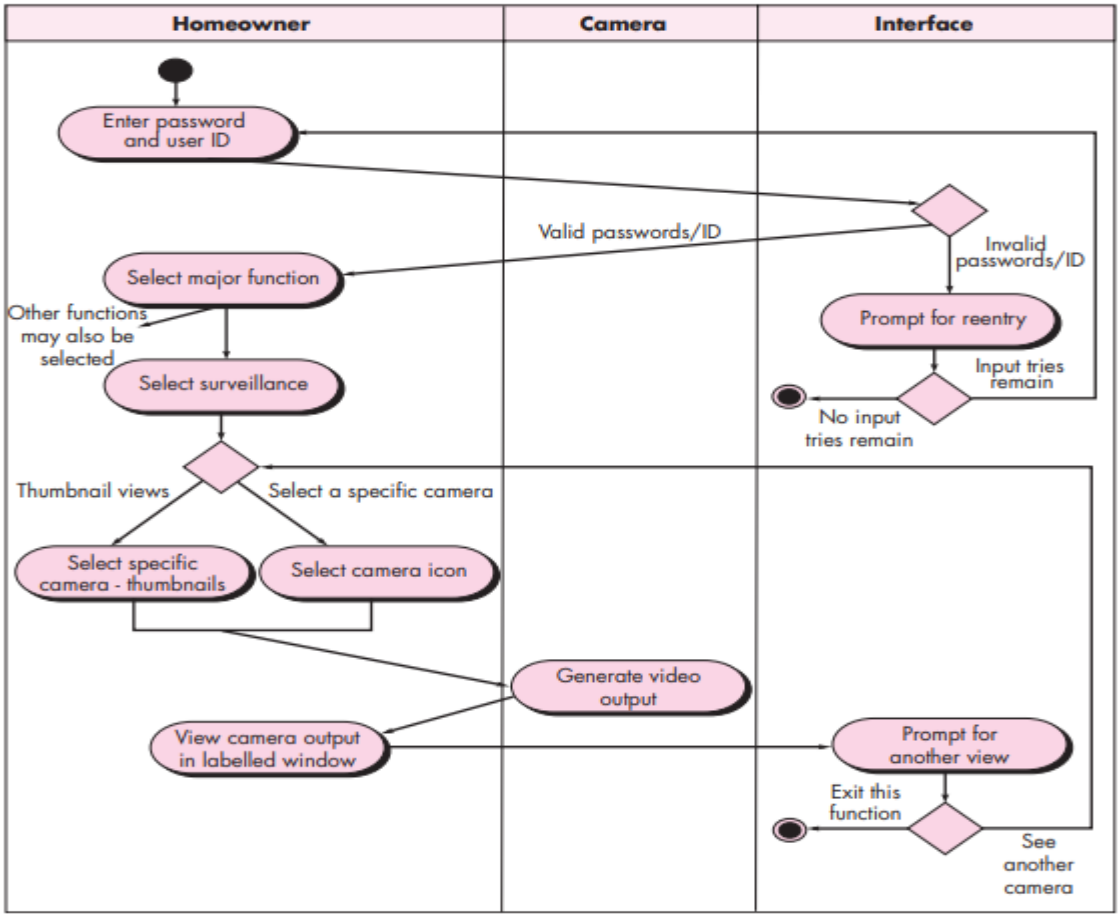



Figure: Swimlane diagram for Access camera surveillance via the Internet—display camera views function

## Data Modeling Concepts

Data modeling is the process of creating a visual representation of how data is organized, stored, and accessed within a system. It helps in defining the structure of the data, the relationships between data elements, and the rules governing data usage. Data models serve as a blueprint for designing databases and ensuring that data is managed consistently and efficiently.

The software team choose to create a data model as part of overall requirements modeling.A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

1) **Data Objects:** 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).

2) 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

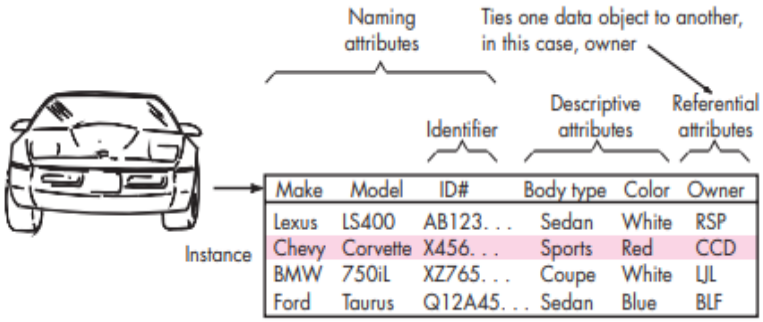



**Figure:** Tabular representation of data objects.

3) **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.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software.

In the latter case, the attributes for car might also include ID number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

4) **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 simple notation.
A connection is established between person and car because the two objects are related

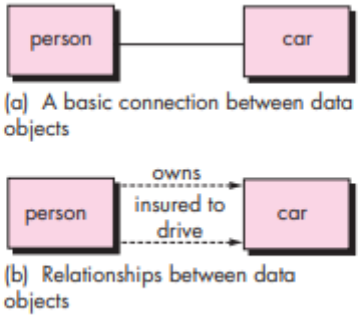



Figure: Relationships between data object

## Class based modeling

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) 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

Guidelines for Identifying and Representing Classes

1. **Identifying Analysis Classes**

When trying to identify potential classes in the system, the following guidelines can be helpful

- **Examine use cases or scenarios**: Review the detailed descriptions of how the system will be used.
- **Identify nouns or noun phrases**: Highlight these words in the use cases as they often represent potential classes.
- **Look for synonyms**: Different terms might refer to the same concept, so it's essential to keep them in mind.

**Categories of Analysis Classes**:

- **External Entities**: Elements like systems or devices that interact with the software.
- **Things**: Items like reports or signals that are relevant to the application domain.
- **Occurrences/Events**: Specific actions or incidents like transactions that happen during system operations.
- **Roles**: Specific positions or job titles (e.g., manager, engineer) that interact with the system.
- **Organizational Units**: Departments or teams that are significant in the application.
- **Places**: Locations like warehouses or office spaces that relate to the system.
- **Structures**: Objects or entities that can be grouped into a class.

2. **Selection Characteristics for Classes**

When deciding whether to include a potential class in the analysis model, consider the following criteria:

1. **Retained Information**: The class must store data that is critical for system functionality.
2. **Needed Services**: The class should have operations that manipulate its data.
3. **Multiple Attributes**: The class should have several attributes rather than just one.
4. **Common Attributes**: Attributes should apply to all instances of the class.
5. **Common Operations**: Operations should be applicable to all instances of the class.
6. **Essential Requirements**: The class should represent something essential to solving the problem.

3. **Specifying Attributes**

Attributes describe the properties of a class and help to differentiate one object from another. To specify attributes:

- Analyze use cases to identify the key data elements associated with the class.
- Select attributes that meaningfully belong to the class in the context of the system requirements.

4. **Defining Operations**

Operations specify the actions that can be performed on objects of a class. These operations generally fall into four categories:

1. **Data Manipulation**: Operations that add, delete, or modify the data in the attributes.
2. **Computation**: Operations that perform calculations using the object's data.

3. **State Inquiry**: Operations that retrieve or check the state of the object.
4. **Event Monitoring**: Operations that monitor the object for specific events or changes.

Example Class Diagram

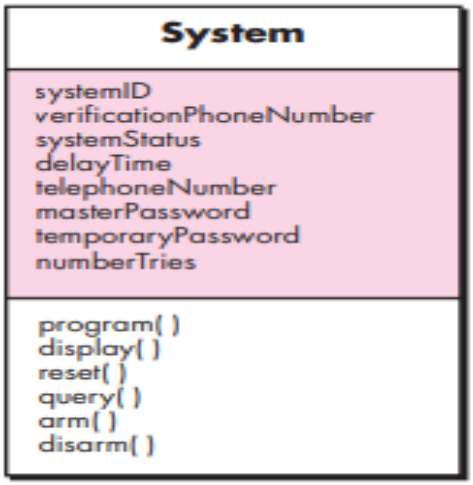



Figure : Class diagram for the system class

## 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

- 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.


Figure : CRC Card

## Classes

**The taxonomy of class types**

1) **Entity classes**: Also called model or business classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor). These classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).
2) **Boundary classes** : Are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used. Entity objects contain information that is important to users, but they do not display themselves.
   **Boundary classes** designed with the responsibility of managing the way entity objects are represented to users. For example, a boundary class called Camera Window would have the responsibility of displaying surveillance camera output for the SafeHome system.

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

## Responsibilities.

Basic guidelines for identifying responsibilities (attributes and operations) have been presented

**suggest five guidelines for allocating responsibilities to classes**:

1. **System intelligence should be distributed across classes to best address the needs of the problem** : This intelligence can be distributed across classes in a number of different ways. "Dumb" classes (those that have few responsibilities) can be modeled to act as servants to a few "smart" classes (those having many responsibilities).

2. Each responsibility should be stated as generally as possible. This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).

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.

5. Responsibilities should be shared among related classes, when appropriate.

## 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.

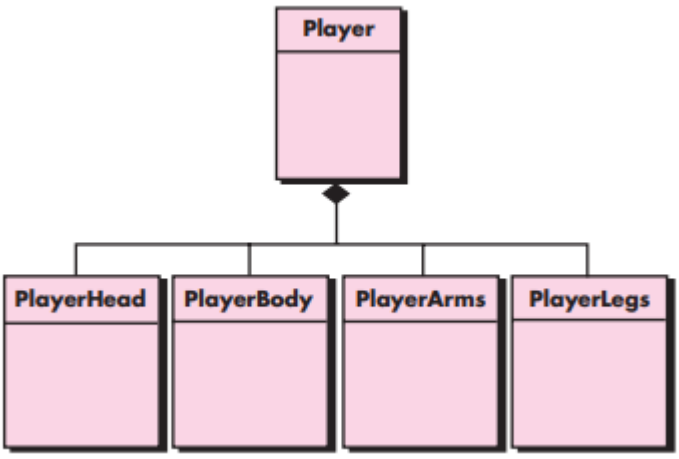



**Figure: A composite aggregate class**

- When one class must acquire information from another class, the **has-knowledge of relationship** is established. The determine-sensor-status() responsibility noted earlier is an example of a has-knowledge-of relationship.
- The **depends-upon relationship** implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of. For example, Player Head must always be connected to Player Body.

**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 Sensor card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes.

## Associations and Dependencies

- In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another .In UML these relationships are called **Associations.**
- Example: the FloorPlan class is defined by identifying a set of associations between FloorPlan and two other classes, Camera and Wall refer below diagram.

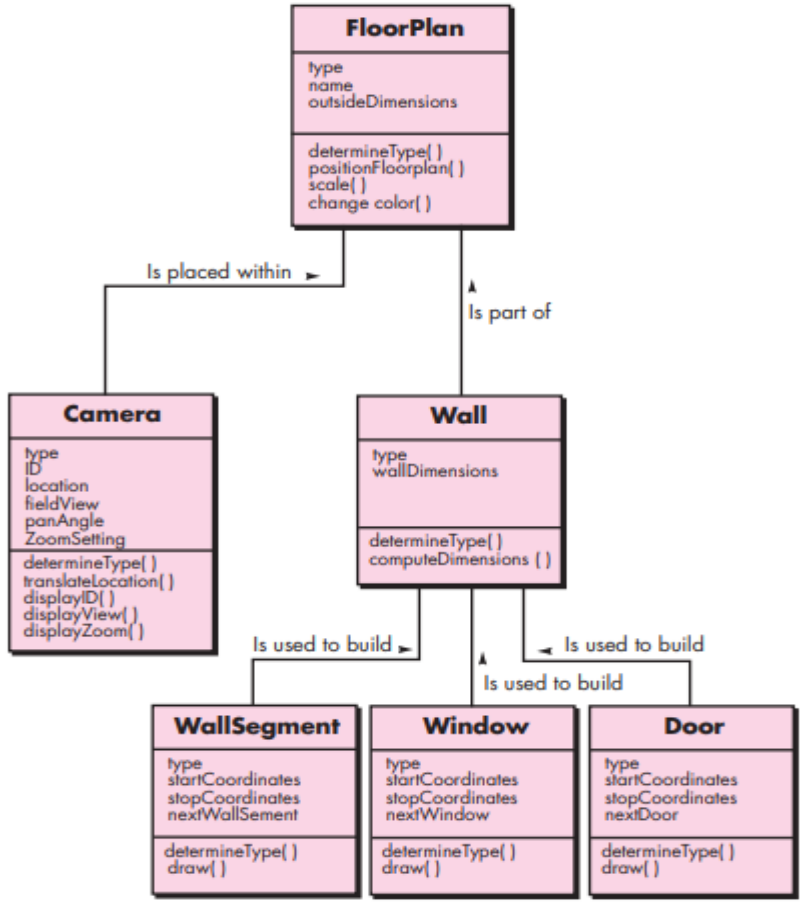



Figure: Class diagram for FloorPlan

- In some cases, an association may be further defined by indicating multiplicity.
- Referring to above Figure , a Wall object is constructed from one or more Wall Segment objects.
- In addition, the Wall object may contain **0** or **more** Window objects and **0 or more Door** objects.
- These multiplicity constraints are illustrated in above Figure .
- where "one or more" is represented using **1. .*,** and "0 or more" by **0 . .***

## Dependencies

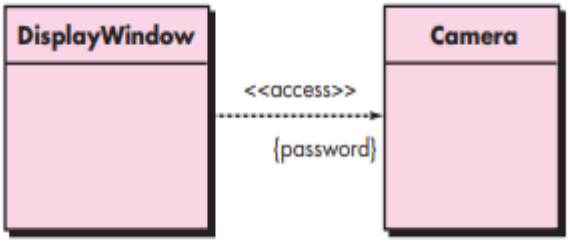



**Figure: Dependencies**

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

## Analysis Packages

**Definition:** 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 representative name

**Example:** As the analysis model for the video game is developed, a large number of classes are derived. Some focus on the game environment—the visual scenes that the user sees as the game is played.

Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and Visual Effect might fall within this category.

Others focus on the characters within the game, describing their physical features, actions, and constraints.

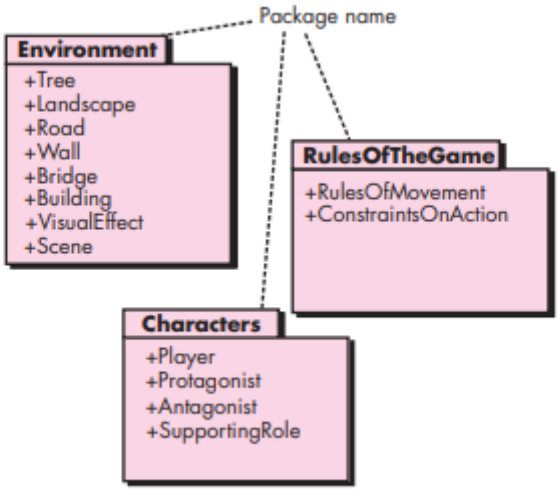



**Figure: Packages**