Module-02

Chapter:-1 - Understanding Requirements

Requirements Engineering

The Challenge of Building Software:

- Software development is engaging and creative but can lead to premature action without proper understanding of requirements.
- Some developers argue that understanding evolves during the build, stakeholders understand better with early iterations, and rapid changes make detailed requirements unnecessary.
- These arguments have elements of truth but can lead to project failure if not managed properly.

Requirements Engineering:

- A crucial software engineering action starting during the communication activity and continuing into the modelling activity.
- It builds a bridge to design and construction, originating from stakeholders' needs or a broader system definition.

Tasks of Requirements Engineering:

- **Inception:** Establishes a basic understanding of the problem, stakeholders, desired solution, and preliminary communication and collaboration.
- **Elicitation:** Involves gathering requirements from customers/users, often encountering problems of scope, understanding, and volatility.
- **Elaboration:** Refines and expands the information obtained during inception and elicitation, developing a detailed requirements model.

- **Negotiation:** Reconciles conflicts between different stakeholders' requirements through prioritization, cost and risk assessment, and iterative discussions.
- **Specification:** Documents requirements in various forms, such as written documents, graphical models, usage scenarios, prototypes, or combinations thereof.
- **Validation:** Assesses the quality of requirements to ensure they are unambiguous, consistent, complete, and conform to standards.
- **Requirements Management:** Tracks and controls changes to requirements throughout the project's life, similar to software configuration management.

Inception Stage:

- Projects usually begin from a business need or market opportunity.
- Stakeholders define the business case, market scope, feasibility, and project scope, initiating discussions with the software engineering team.

Elicitation Stage:

- Challenges include ill-defined system boundaries, misunderstandings about needs, and evolving requirements.
- An organized approach to requirements gathering is essential.

Elaboration Stage:

- Focuses on developing a refined model that identifies software functions, behaviours, and information.
- User scenarios are used to extract and define analysis classes and their attributes, services, relationships, and collaborations.

Negotiation Stage:

- Resolves conflicts between different stakeholders' requirements through iterative prioritization and compromise to achieve mutual satisfaction.

Specification Stage:

- Requirements can be documented in various forms, with flexibility depending on the project size and complexity.
- Consistent presentation of requirements is beneficial for clarity.

Validation Stage:

- Technical reviews are conducted to ensure requirements are clear, complete, and correct, involving a review team of engineers, customers, users, and stakeholders.

Requirements Management Stage:

- Ongoing activity to manage changes to requirements throughout the project's lifecycle, involving identification, control, and tracking of changes.

Establishing the ground work

Ideal vs. Real-world Settings:

- In an ideal scenario, stakeholders and software engineers work closely as a single team.
- Reality often involves geographical separation, vague requirements, conflicting opinions, limited technical knowledge, and time constraints.

Establishing Groundwork for Requirements:

- It's crucial to lay a strong foundation for understanding software requirements to ensure project success.

Identifying Stakeholders:

- Defined as anyone who benefits directly or indirectly from the system being developed.
- Common stakeholders include business managers, product managers, marketing, internal/external customers, end users, consultants, product engineers, software engineers, and support/maintenance engineers.
- Begin with an initial list of stakeholders and expand it by asking each stakeholder for additional contacts.

Recognizing Multiple Viewpoints:

- Different stakeholders have different views and priorities for the system.
- Marketing focuses on marketable features, business managers on budget and timelines, end users on usability, software engineers on technical infrastructure, and support engineers on maintainability.
- Collect and categorize stakeholder information to address inconsistencies and conflicts in requirements.

Working toward Collaboration:

- Collaboration involves stakeholders providing their viewpoints and working together to resolve conflicts.
- A requirements engineer identifies commonalities and conflicts, facilitating decision-making.
- A strong project champion often makes final decisions on requirements.

Asking the First Questions:

- Initial questions should be context-free, focusing on identifying stakeholders, project goals, and benefits:

- Who is behind the request?
- Who will use the solution?
- What is the economic benefit of the solution?
- Are there alternative sources for the solution?

- Follow-up questions aim to understand the problem and solution context:
    - What constitutes good output?
    - What problems will the solution address?
    - Describe the business environment for the solution.
    - Any special performance issues or constraints?
- Meta-questions assess the effectiveness of communication:
    - Are you the right person to answer these questions?
    - Are your answers official?
    - Are my questions relevant?
    - Am I asking too many questions?
    - Can anyone else provide more information?
    - Should I ask anything else?

Moving Beyond Initial Q&A:

- Initial Q&A sessions are useful for the first encounter but should evolve into a more dynamic format involving problem-solving, negotiation, and specification for effective requirements elicitation.

Eliciting Requirements

1. **Initial Meeting Preparation:**
    - **Attendees:** Include both software engineers and stakeholders.
    - **Rules and Agenda:** Establish guidelines for preparation and participation, ensuring the agenda is formal enough to cover key points but informal enough to encourage idea exchange.

- **Facilitator:** Assign a facilitator to control the meeting.
- **Definition Mechanism:** Use tools such as worksheets, flip charts, wall stickers, or electronic platforms (bulletin boards, chat rooms) to organize and display ideas.

2. **Inception Phase:**
    - Conduct initial Q&A sessions to establish the scope of the problem and overall perception of a solution.
    - Develop a preliminary "product request" document summarizing the project's aims and initial requirements.
    - Distribute the product request to all attendees before the meeting.
3. **Requirements Gathering Meeting:**
    - **Before the Meeting:**
        - Attendees review the product request and prepare lists of objects, services, constraints, and performance criteria.
    - **During the Meeting:**
        - Present individual lists for each topic area.
        - Combine lists to eliminate redundancies and add new ideas.
        - Engage in discussion to refine and achieve a consensus on the lists of objects, services, constraints, and performance criteria.
    - **Post-Meeting:**
        - Develop mini-specifications for objects or services that need further elaboration.
        - Review and refine mini-specifications with all stakeholders, adding new requirements as necessary.
        - Maintain an issues list for unresolved topics.

Quality Function Deployment (QFD):

QFD translates customer needs into technical requirements, focusing on maximizing customer satisfaction. It distinguishes three types of requirements:

1. **Normal Requirements:**
   - Objectives and goals explicitly stated by the customer (e.g., specific system functions, performance levels).
2. **Expected Requirements:**
   - Fundamental requirements implicit to the product (e.g., ease of use, reliability, installation ease). Absence leads to dissatisfaction.
3. **Exciting Requirements:**
   - Unexpected features that delight customers (e.g., innovative capabilities like multitouch screens in smartphones).

Usage Scenarios:

Creating scenarios (or use cases) helps to understand how different user classes will use the system. These scenarios describe the system's functions and features in practical use, aiding in the transition to technical software engineering activities.

Elicitation Work Products:

The outputs of requirements elicitation vary by project size but typically include:

1. **Statement of Need and Feasibility:** Clarifies the necessity and practicality of the project.
2. **Bounded Statement of Scope:** Defines the system or product's boundaries.
3. **List of Participants:** Documents the customers, users, and stakeholders involved.
4. **Technical Environment Description:** Details the system’s operational context.
5. **Requirements List:** Organized by function, with domain constraints for each requirement.
6. **Usage Scenarios:** Provide insights into system use under different conditions.

7. **Prototypes:** Developed as needed to better define requirements.

Each work product is reviewed by all participants to ensure accuracy and completeness.

Developing use cases

Writing Effective Use Cases

Alistair Cockburn defines a use case as a description of the system's behavior as it responds to requests from its stakeholders, capturing a contract between the stakeholders and the system. A use case typically tells a story about how an end user interacts with the system under specific circumstances.

1. **Actors:**
   - **Definition:** Actors are entities that interact with the system, playing specific roles. These can be people or devices.
   - **Primary Actors:** Directly interact with the system to achieve a goal.
   - **Secondary Actors:** Support the system so that primary actors can perform their roles.
   - **Example:** For a home security system, actors might include the homeowner, setup manager, sensors, and monitoring subsystem.
2. **Identifying Actors:**
   - Not all actors are identified in the first iteration; primary actors are identified early, and secondary actors are added as understanding of the system evolves.
3. **Developing Use Cases:**
   - **Questions to Answer:**
     - Who are the primary and secondary actors?
     - What are the actors' goals?
     - What preconditions should exist?
     - What tasks or functions do the actors perform?

- What exceptions might occur?
- What variations in interactions are possible?
- What information does the actor acquire, produce, or change?
- How does the actor inform the system about changes?
- What information does the actor desire from the system?
- Does the actor need to be informed about unexpected changes?

4. **Example Use Case: SafeHome System**

   Actors:

   - Homeowner
   - Setup Manager
   - Sensors
   - Monitoring Subsystem

   Homeowner Interactions:

   - Entering a password
   - Inquiring about the status of security zones and sensors
   - Pressing the panic button
   - Activating/deactivating the system

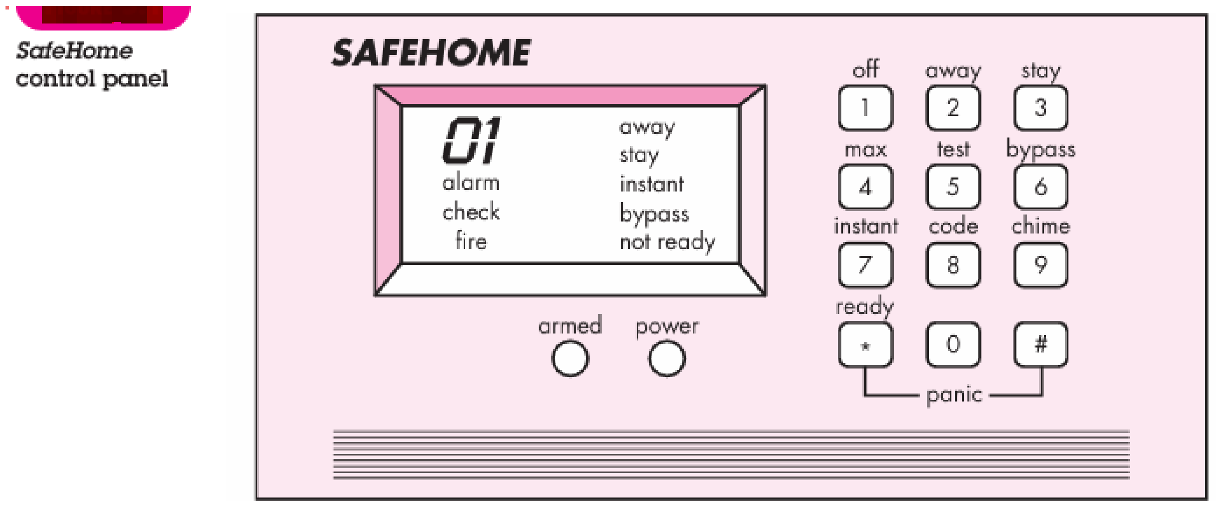
Basic Use Case: System Activation

Homeowner checks if the system is ready via the control panel.

Homeowner enters a four-digit password.

Homeowner selects "stay" or "away" to activate the system.

The system confirms activation with a visual indicator.



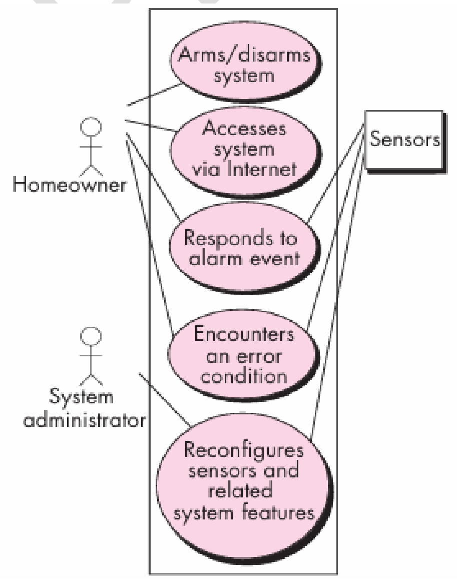
Detailed Use Case: Initiate Monitoring

- **Primary Actor:** Homeowner
- **Goal in Context:** To set the system to monitor sensors.
- **Preconditions:** System programmed with a password and sensor recognition.
- **Trigger:** Homeowner decides to activate the system.
- **Scenario:**
  1. Homeowner observes the control panel.
  2. Homeowner enters the password.
  3. Homeowner selects "stay" or "away".
  4. Homeowner sees the alarm light indicating the system is armed.
- **Exceptions:**
  0. System not ready: Homeowner checks and closes all sensors.
  1. Incorrect password: Homeowner re-enters the correct password.
  2. Password not recognized: Contact support to reprogram.
  3. "Stay" selected: Perimeter sensors activated.
  4. "Away" selected: All sensors activated.
- **Priority:** Essential

- **When Available:** First increment
- **Frequency of Use:** Regular
- **Channels to Actor:** Via control panel
- **Secondary Actors:** Support technician, sensors
- **Channels to Secondary Actors:** Phone line, radio frequency interfaces
- **Open Issues:**
  - Alternate activation methods (e.g., abbreviated password)
  - Additional text messages on the control panel
  - Time limit for entering the password
  - Deactivation options before activation

Use cases should be reviewed meticulously to ensure clarity and completeness. Ambiguities in the use case can indicate potential problems that need addressing.

UML use case diagram for *SafeHome* home security function


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

Building the requirements model

The Intent of the Analysis Model

The analysis model is designed to provide a comprehensive description of the informational, functional, and behavioral domains required for a computer-based system. It evolves as more is learned about the system and as stakeholders refine their understanding of their needs, serving as a dynamic snapshot of requirements at any given time. Elements of the model may stabilize over time, forming a solid foundation for subsequent design tasks, while others may remain volatile, reflecting areas where stakeholder understanding is still evolving.

Elements of the Requirements Model

There are various ways to represent requirements for a computer-based system, and different modes of representation can provide different perspectives, helping to uncover omissions, inconsistencies, and ambiguities. Here are some common elements in most requirements models:

Scenario-Based Elements

Scenario-based elements describe the system from the user's point of view, often starting with basic use cases and evolving into more elaborate template-based use cases. These scenarios serve as input for creating other modeling elements.

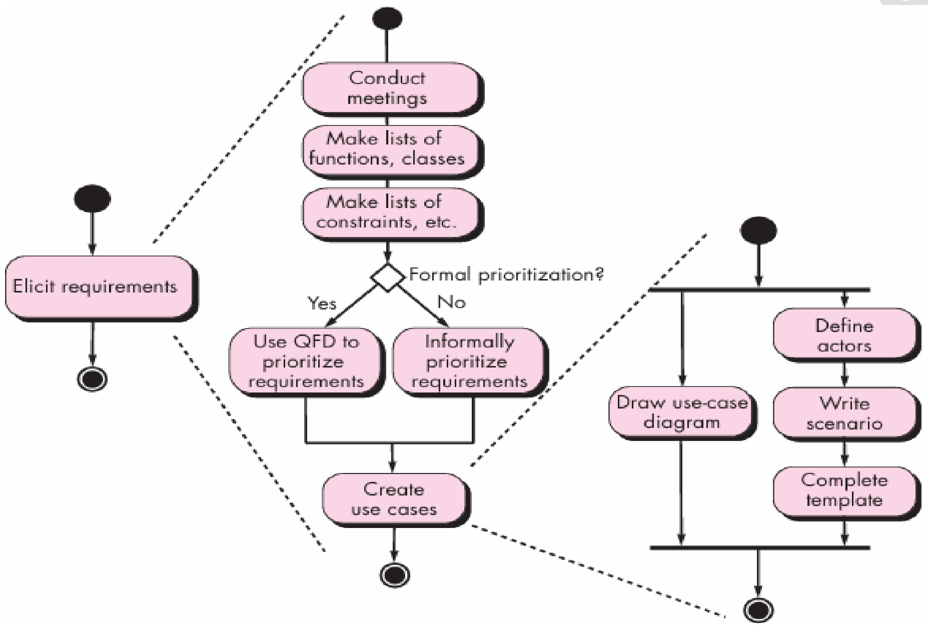
Class-Based Elements

Each usage scenario implies a set of objects manipulated by the actor interacting with the system. These objects are categorized into classes, collections of things with similar attributes and behaviors. Class diagrams, such as a UML class diagram, depict these classes and their relationships, showing attributes and operations that can modify these attributes.

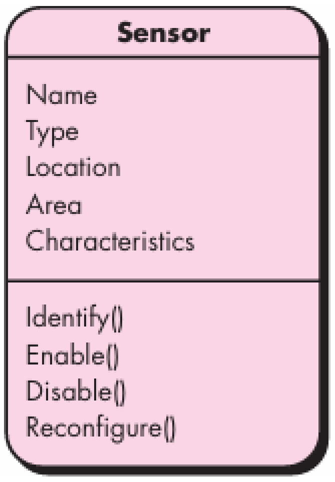
Behavioral Elements

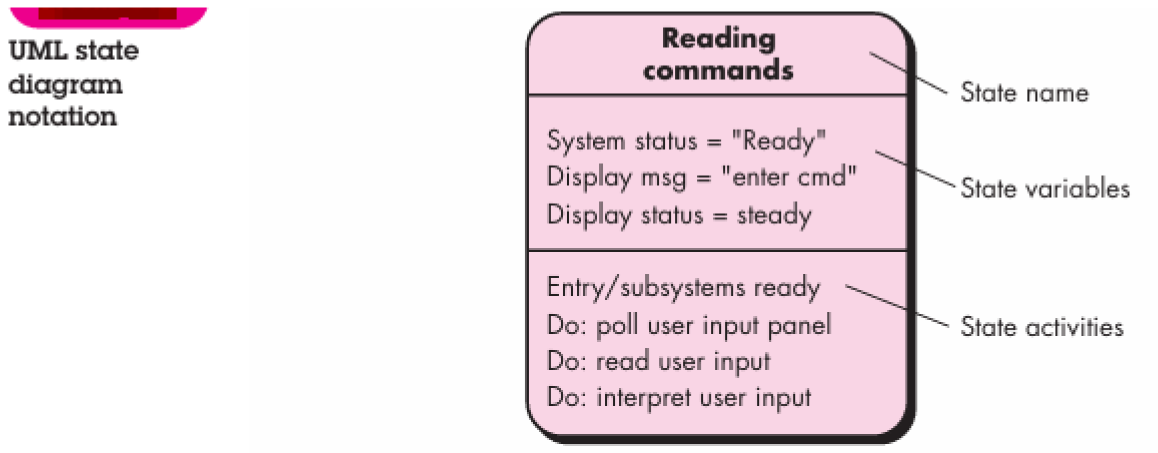
The behavior of a system is critical to its design and implementation. Behavioral modeling elements, such as state diagrams, depict the system's states and the events that cause state changes. These diagrams also indicate actions taken as a consequence of specific events. Behavioural modelling extends to individual classes, detailing their specific behaviours.

UML activity diagrams for eliciting requirements



Class diagram for sensor





Flow-Oriented Elements

Flow-oriented elements model the transformation of information as it moves through the system. This involves depicting the input, the transformations applied to it, and the output produced. Flow models can be created for any computer-based system, regardless of size or complexity.

Analysis Patterns

Analysis patterns are reusable solutions for common problems within a specific application domain. These patterns provide two main benefits:

1. **Speeding up Development:** They help develop abstract analysis models that capture the main requirements of a problem by providing reusable models with examples and descriptions of their advantages and limitations.
2. **Facilitating Transformation:** They aid in transforming the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are referenced by name within the analysis model and stored in a repository, making them accessible for reuse by requirements engineers. Information about an analysis pattern is presented in a standard template, further discussed in detail in subsequent chapters.

Example: SafeHome System

Actors and Interactions

For the SafeHome system, consider the homeowner actor. The interactions include:

- Entering a password
- Inquiring about the status of security zones and sensors
- Pressing the panic button
- Activating/deactivating the system

Basic Use Case: System Activation

1. **Homeowner** observes the control panel to ensure the system is ready.
2. **Homeowner** enters a four-digit password.
3. **Homeowner** selects "stay" or "away" to activate the system.
4. **Homeowner** observes a red alarm light indicating system activation.

Detailed Use Case: Initiate Monitoring

- **Primary Actor:** Homeowner
- **Goal in Context:** Set the system to monitor sensors.
- **Preconditions:** System programmed with a password and sensor recognition.
- **Trigger:** Homeowner decides to activate the system.
- **Scenario:**
    1. Homeowner observes the control panel.
    2. Homeowner enters the password.
    3. Homeowner selects "stay" or "away".
    4. Homeowner sees the alarm light indicating the system is armed.

- **Exceptions:**
    1. System not ready: Homeowner checks and closes all sensors.
    2. Incorrect password: Homeowner re-enters the correct password.
    3. Password not recognized: Contact support to reprogram.
    4. "Stay" selected: Perimeter sensors activated.
    5. "Away" selected: All sensors activated.
- **Priority:** Essential
- **When Available:** First increment
- **Frequency of Use:** Regular
- **Channels to Actor:** Via control panel
- **Secondary Actors:** Support technician, sensors
- **Channels to Secondary Actors:** Phone line, radio frequency interfaces
- **Open Issues:**
    - Alternate activation methods (e.g., abbreviated password)
    - Additional text messages on the control panel
    - Time limit for entering the password
    - Deactivation options before activation

Use cases and other elements should be reviewed thoroughly to ensure clarity and completeness, as ambiguities can indicate potential problems that need addressing.

Negotiating Requirements

Negotiation in Requirements Engineering

In an ideal scenario, the tasks of inception, elicitation, and elaboration determine customer requirements in sufficient detail to proceed seamlessly to subsequent software engineering activities. However, this ideal is rarely achieved. In reality, negotiation with stakeholders often becomes necessary to balance functionality, performance, and other product or system characteristics against constraints like cost and time-to-market.

Negotiation Goals and Approach

The primary goal of negotiation is to develop a project plan that meets stakeholder needs while accommodating real-world constraints (e.g., time, budget, and personnel). The best negotiations aim for a "win-win" outcome, where:

- **Stakeholders win** by getting a system or product that satisfies the majority of their needs.
- **The software team wins** by working within realistic and achievable budgets and deadlines.

Negotiation Activities

Barry Boehm defines a set of negotiation activities to be conducted at the beginning of each software process iteration. These activities are designed to ensure effective stakeholder communication and to reconcile differing priorities and constraints. The activities include:

1. **Identification of Key Stakeholders:**
    - Identify individuals or groups who have a vested interest in the system or subsystem being developed.
    - Stakeholders typically include end-users, customers, project sponsors, developers, and possibly regulatory bodies.
2. **Determination of Stakeholders' "Win Conditions":**
    - Understand the specific needs, desires, and success criteria for each stakeholder.
    - Win conditions may include specific functional requirements, performance targets, budget limits, or deadlines.
3. **Negotiation of Win Conditions:**
    - Reconcile the various win conditions to develop a set of mutually acceptable goals.

- Aim to create a scenario where all stakeholders feel their primary needs are addressed.
- Ensure the negotiated plan is feasible within the given constraints.

Achieving a Win-Win Result

Successful completion of these initial negotiation steps results in a win-win outcome, which serves as the key criterion for moving forward with subsequent software engineering activities. Achieving a win-win result involves:

- **Clear Communication:** Ensure all stakeholders understand each other's priorities and constraints.
- **Compromise and Flexibility:** Be prepared to adjust expectations and find middle ground.
- **Focus on Shared Goals:** Emphasize the common objectives and benefits to all parties.

Practical Example: SafeHome System

For instance, in the development of the SafeHome system:

1. **Identification of Stakeholders:**
   - Homeowners (end-users)
   - Setup managers (users with administrative roles)
   - Monitoring and response subsystem operators
   - Development team
   - Budget managers
2. **Determination of Win Conditions:**
   - Homeowners: Reliable security system with easy-to-use interface.
   - Setup managers: Flexible configuration options.
   - Monitoring subsystem: Seamless integration with existing systems.

- Development team: Clear requirements and achievable deadlines.
- Budget managers: Cost-effective solution within budget.

3. **Negotiation of Win Conditions:**
    - Homeowners may need to compromise on advanced features for a simpler interface.
    - Setup managers may agree to phased implementation of configuration options.
    - The development team may negotiate for extended deadlines for critical features.
    - Budget managers may adjust the budget for essential features, with cost-saving measures for others.

By addressing these negotiation activities, the project can move forward with a shared understanding and commitment to the project's success.

Validating Requirements

Reviewing requirements is a critical step to ensure the requirements model accurately reflects stakeholder needs and provides a solid foundation for design. Here are the key questions to consider during a requirements review:

1. **Consistency with Objectives:**
    - Is each requirement consistent with the overall objectives for the system or product?
2. **Proper Level of Abstraction:**
    - Have all requirements been specified at the proper level of abstraction?
    - Do some requirements provide a level of technical detail that is inappropriate at this stage?
3. **Necessity:**

- Is the requirement really necessary, or does it represent an add-on feature that may not be essential to the system's objective?

4. **Bounded and Unambiguous:**
   - Is each requirement bounded and unambiguous?
5. **Attribution:**
   - Does each requirement have attribution?
   - Is a source (generally a specific individual) noted for each requirement?
6. **Conflict Checking:**
   - Do any requirements conflict with other requirements?
7. **Achievability:**
   - Is each requirement achievable in the technical environment that will house the system or product?
8. **Testability:**
   - Is each requirement testable once implemented?
9. **Reflection of Information, Function, and Behavior:**
   - Does the requirements model properly reflect the information, function, and behavior of the system to be built?
10. **Partitioning:**
    - Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
11. **Use of Requirements Patterns:**
    - Have requirements patterns been used to simplify the requirements model?
    - Have all patterns been properly validated?
    - Are all patterns consistent with customer requirements?

Additional Considerations

- **Prioritization and Grouping:**
  - Have the requirements been prioritized by stakeholders?

  - Are the requirements grouped within requirements packages that will be implemented as software increments?

By asking these questions, stakeholders and development teams can ensure that the requirements model is comprehensive, clear, and aligned with the project's goals. This process helps in identifying potential issues early, thereby reducing the risk of costly changes later in the development process.

Chapter:-2

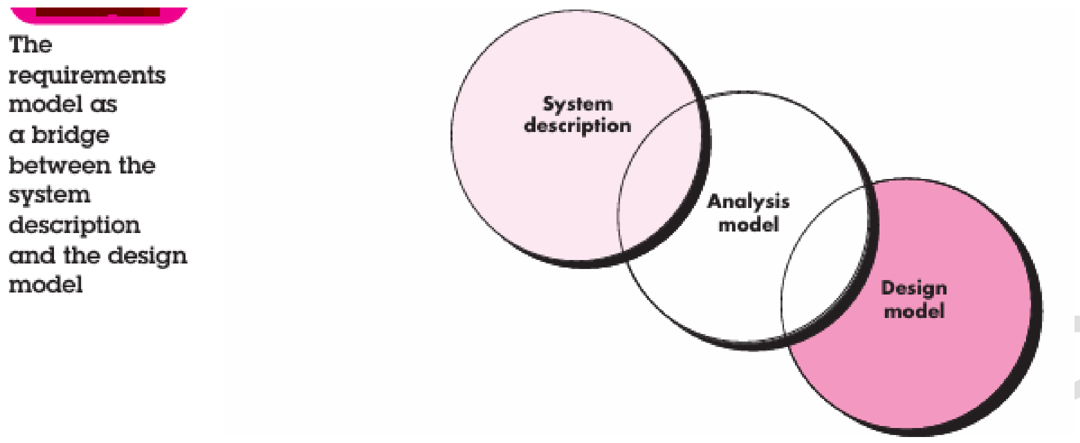
Requirements Modeling Scenarios, Information and Analysis classes

Requirements Analysis

1. **Purpose and Outcome**:
   - Specifies software's operational characteristics.
   - Indicates software's interface with other system elements.
   - Establishes constraints that software must meet.
2. **Tasks Involved**:
   - Elaborates on basic requirements from inception, elicitation, and negotiation phases of requirements engineering.

Requirements Modeling Types

1. **Scenario-Based Models**:
   - Focus on requirements from the perspective of various system actors.
2. **Data Models**:
   - Depict the information domain for the problem.
3. **Class-Oriented Models**:
   - Represent object-oriented classes (attributes and operations) and their collaboration to achieve system requirements.
4. **Flow-Oriented Models**:
   - Represent the functional elements of the system and data transformation.
5. **Behavioral Models**:
   - Depict software behavior in response to external events.


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

Overall Objectives and Philosophy of Requirements Modeling

1. **Focus on 'What', Not 'How'**:
    - Identify what user interactions occur.
    - Determine what objects the system manipulates.
    - Define what functions the system must perform.
    - Describe what behaviors the system exhibits.
    - Specify what interfaces are defined.
    - Establish what constraints apply.
2. **Iterative Approach**:
    - Complete specification of requirements may not be possible initially.
    - Customers may be unsure of precise requirements for certain system aspects.
    - Developers may be uncertain if specific approaches will meet function and performance needs.
    - Model what is known and use it as the basis for designing the software incrementally.
3. **Primary Objectives of the Requirements Model**:
    - **Describe Customer Requirements**:
        - Clearly articulate what the customer needs from the system.
    - **Establish Basis for Software Design**:

        - Provide essential information that guides the creation of the software design.
    - **Define Validatable Requirements**:
        - Outline requirements that can be validated once the software is built.
4. **Bridging Analysis and Design**:
    - The analysis model connects the overall system or business functionality (achieved through software, hardware, data, human, and other elements) to a detailed software design.
    - The design includes the software's application architecture, user interface, and component-level structure.

By adhering to these principles, requirements modeling ensures a thorough understanding of the system requirements, facilitating effective and efficient software design and development.

Analysis Rules of Thumb (Arlow and Neustadt)

1. **Focus on Visible Requirements**:
    - Keep the level of abstraction high, avoid unnecessary details.
2. **Add to Understanding**:
    - Each model element should enhance understanding of the requirements.
3. **Delay Nonfunctional Considerations**:
    - Focus on problem domain analysis before addressing infrastructure needs.
4. **Minimize Coupling**:
    - Represent relationships but strive to reduce high levels of interconnectedness.
5. **Provide Stakeholder Value**:
    - Ensure the model is useful to all stakeholders involved.
6. **Simplicity**:
    - Keep models simple and avoid unnecessary complexity.

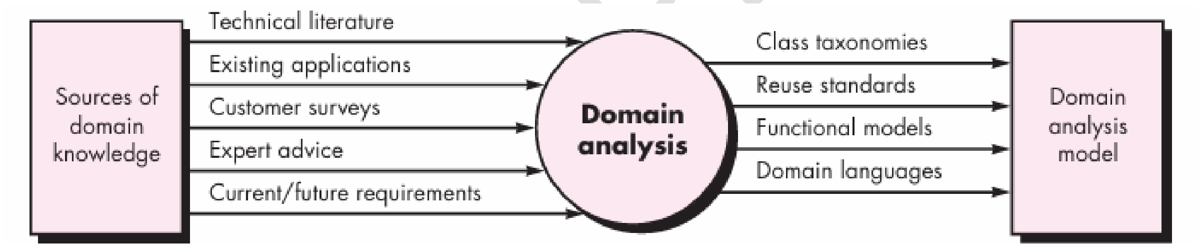
Domain Analysis

1. **Purpose and Importance**:
    - **Recognition of Recurrent Patterns**:
        - Identify and categorize analysis patterns that often reoccur across applications within a specific business domain.
    - **Expedited Model Creation**:
        - Recognizing and applying these patterns expedites the creation of the analysis model.
    - **Improved Time-to-Market and Reduced Costs**:
        - Increases the likelihood of using design patterns and executable software components, leading to faster development and lower costs.
2. **Domain Analysis Definition (Firesmith)**:
    - **Software Domain Analysis**:
        - Identification, analysis, and specification of common requirements within a specific application domain for reuse across multiple projects.
    - **Object-Oriented Domain Analysis**:
        - Focuses on identifying, analyzing, and specifying common, reusable capabilities in terms of common objects, classes, subassemblies, and frameworks.
3. **Application Domains**:
    - Examples include avionics, banking, multimedia video games, and software for medical devices.
    - The goal is to find or create broadly applicable analysis classes and patterns for reuse.
4. **Ongoing Activity**:
    - Domain analysis is a continuous software engineering activity, not tied to a specific project.

- **Role of Domain Analyst**:
    - Similar to a master toolsmith who designs and builds reusable tools.
    - Discover and define reusable analysis patterns and classes for multiple applications.

5. **Inputs and Outputs of Domain Analysis**:
    - **Sources of Domain Knowledge**:
        - Surveyed to identify objects that can be reused across the domain.
    - **Outputs**:
        - Analysis patterns, analysis classes, and related reusable information.

By conducting domain analysis, software engineers can leverage common patterns and classes across multiple projects, enhancing efficiency and fostering consistency in software development.



Requirements Modeling Approaches

1. **Structured Analysis**:
    - **Data and Process Separation**:
        - Treats data and the processes that transform it as separate entities.
    - **Data Object Modeling**:
        - Defines attributes and relationships of data objects.
    - **Process Modeling**:

    - Shows how processes manipulate data as it flows through the system.

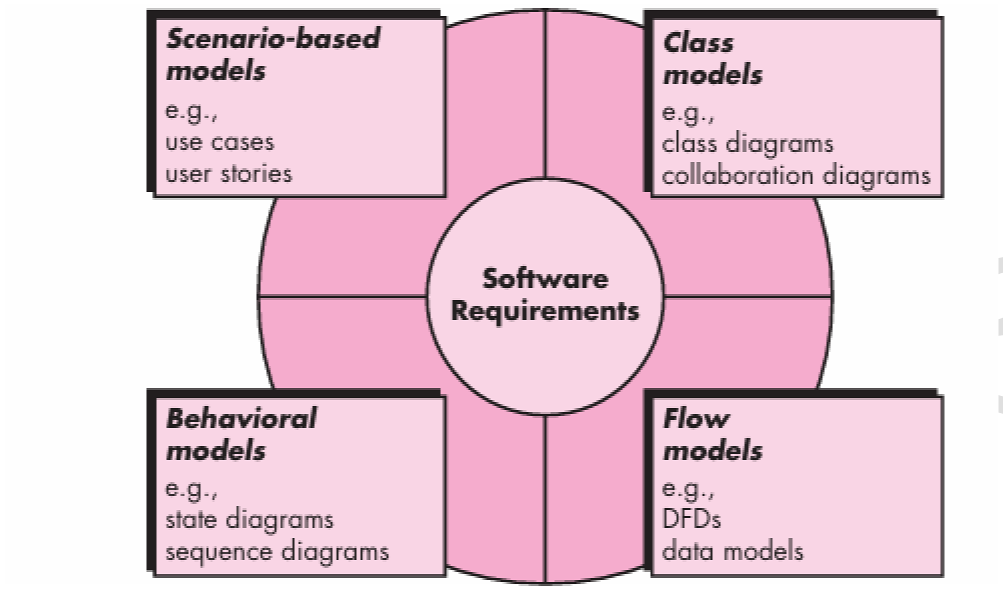
2. **Object-Oriented Analysis (OOA)**:
    - **Class Definition and Collaboration**:
        - Focuses on defining classes and how they collaborate to meet customer requirements.
    - **Use of UML and Unified Process**:
        - Predominantly uses object-oriented approaches such as Unified Modeling Language (UML) and the Unified Process.
3. **Combining Approaches**:
    - The book proposes a model that combines features of both structured and object-oriented analysis.
    - **Choosing an Approach**:
        - Software teams may choose one approach and exclude representations from the other.
        - The choice depends on which combination provides stakeholders with the best model and the most effective bridge to software design.
4. **Elements of the Requirements Model**:
    - **Scenario-Based Elements**:
        - Depict user interactions with the system and the sequence of activities.
    - **Class-Based Elements**:

    - Model objects the system will manipulate, the operations on these objects, relationships (including hierarchical), and collaborations between classes.

  - **Behavioral Elements**:
    - Show how external events change the system or class states.
  - **Flow-Oriented Elements**:
    - Represent the system as an information transform, showing data transformation as it flows through system functions.

5. **Derivation and Specific Content**:
  - Analysis modeling leads to the derivation of these modeling elements.
  - **Project-Specific Content**:
    - The specific content and diagrams used to construct these elements may vary from project to project.
  - **Simplicity Principle**:
    - Only those modeling elements that add value to the model should be used to keep the modeling process simple.

By understanding and effectively combining these approaches, software teams can create comprehensive and useful models that accurately represent system requirements and provide a solid foundation for design.



Scenario based modeling,

Importance of User Satisfaction in Requirements Modeling

- **User Satisfaction**:
  - User satisfaction is the primary measure of a computer-based system or product's success.
  - Understanding end user interactions helps in characterizing requirements and building effective models.

Requirements Modeling with UML

- **Scenarios in Requirements Modeling**:
  - Begins with creating use cases, activity diagrams, and swimlane diagrams.
  - Use cases capture interactions between users (actors) and the system.

Creating a Preliminary Use Case

- **Definition by Alistair Cockburn**:

- A use case is a "contract for behavior," detailing how an actor uses a system to achieve a goal.
- Captures interactions between information producers/consumers and the system.

Writing Use Cases

1. **What to Write About**:
   - Derived from inception and elicitation tasks.
   - Identify stakeholders, define problem scope, set operational goals, and outline functional requirements.
2. **Developing Use Cases**:
   - List functions or activities performed by an actor.
   - Use lists of required system functions, stakeholder conversations, and activity diagrams.
3. **Example: SafeHome Surveillance System**:
   - Functions like selecting cameras, viewing thumbnails, controlling cameras, recording, and accessing via the Internet.
4. **Informal Narrative Use Case**:
   - Written in straightforward language from the actor's point of view.
   - Example narrative for accessing camera surveillance via the Internet, detailing steps and interactions.
5. **Sequential Use Case**:
   - Represents user actions as a sequence of declarative sentences.
   - Example sequence for the ACS-DCV function with step-by-step actions.

Refining a Preliminary Use Case

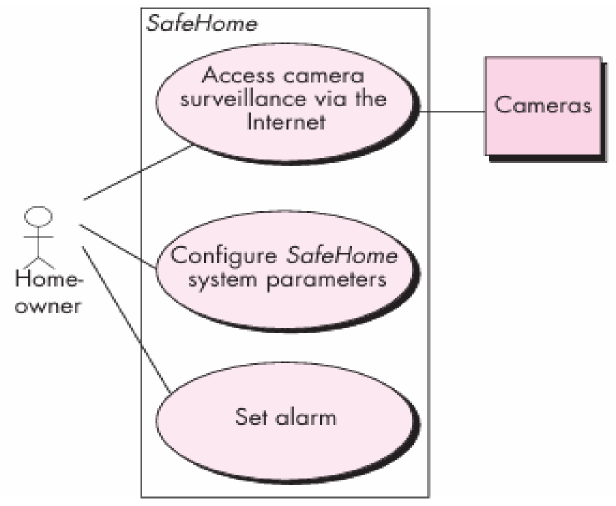
- **Secondary Scenarios**:

- Evaluate each primary scenario step for alternative actions, error conditions, and other behaviors.
- Create secondary scenarios for alternative interactions and error conditions.

- **Example of Refinement**:
    - Identify secondary scenarios like viewing all camera thumbnails or handling error conditions.

Writing a Formal Use Case

- **Formal Use Case Structure**:
    - **Goal in Context**: Defines the overall scope.
    - **Precondition**: What is true before the use case starts.
    - **Trigger**: Event or condition that starts the use case.
    - **Scenario**: Specific actions by the actor and system responses.
    - **Exceptions**: Situations uncovered during refinement.
- **Graphical Representation**:
    - Use-case diagrams in UML can help visualize complex scenarios.
    - Example: Preliminary use-case diagram for SafeHome, showing various use cases.


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

Preliminary use-case diagram for the *SafeHome* system

Limitations and Benefits of Use Cases

- **Limitations**:
    - Quality depends on the author's clarity.
    - Focuses on functional and behavioral requirements, not suitable for nonfunctional requirements.
    - May not be sufficient for detailed and precise requirements, especially in safety-critical systems.
- **Benefits**:
    - Scenario-based modeling is appropriate for most situations encountered in software engineering.
    - Properly developed use cases provide substantial benefits as a modeling tool.

UML models that supplement the Use Case

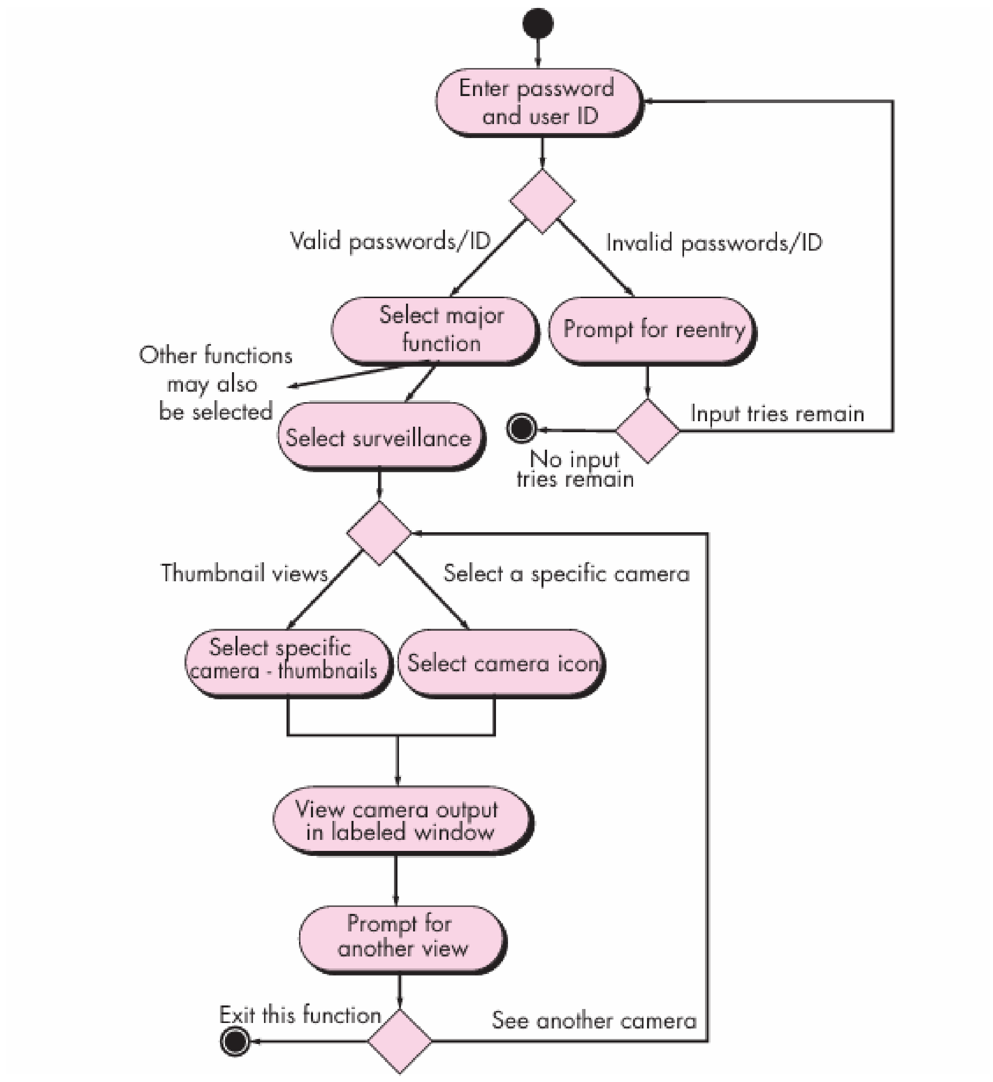
Enhancing Requirements Modeling with UML Graphical Models

Developing an Activity Diagram

- **Purpose**: Provides a graphical representation of interaction flow within a scenario, supplementing text-based models like use cases.
- **Components**:
    - **Rounded Rectangles**: Represent specific system functions.
    - **Arrows**: Indicate the flow through the system.
    - **Decision Diamonds**: Depict branching decisions with labeled arrows.
    - **Solid Horizontal Lines**: Indicate parallel activities.

- **Example**: An activity diagram for the ACS-DCV use case (Figure 6.5) shows details like the limited number of attempts a user has to enter their user ID and password.

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



Swimlane Diagrams

Purpose: The UML swimlane diagram is a variation of the activity diagram designed to represent the flow of activities within a use case while highlighting the responsibilities of different actors or analysis classes. This is achieved by dividing the diagram into parallel segments, resembling the lanes in a swimming pool.

Structure:

- **Parallel Segments (Lanes)**: Each lane represents a different actor or analysis class responsible for certain actions within the use case.
- **Activity Rectangles**: Placed within the lanes, these rectangles depict specific actions or system functions.
- **Arrows**: Indicate the flow of activities and the sequence of operations.
- **Decision Diamonds**: Used to show branching decisions, with arrows leading to different paths based on conditions.

Example:

- **Analysis Classes**: Consider three analysis classes—Homeowner, Camera, and Interface. Each has distinct responsibilities depicted within their respective swimlanes.
- **Interface Class**: Responsible for user prompts like "prompt for reentry" and "prompt for another view." These activities fall within the Interface swimlane.
- **Homeowner Actions**: Arrows from the Interface swimlane lead back to the Homeowner swimlane, indicating the homeowner's actions following the prompts.

Context: Swimlane diagrams, like activity diagrams and use cases, are procedurally oriented. They focus on how various actors invoke specific functions or follow procedural steps to meet system requirements.

Procedural View vs. Information Space

Procedural View:

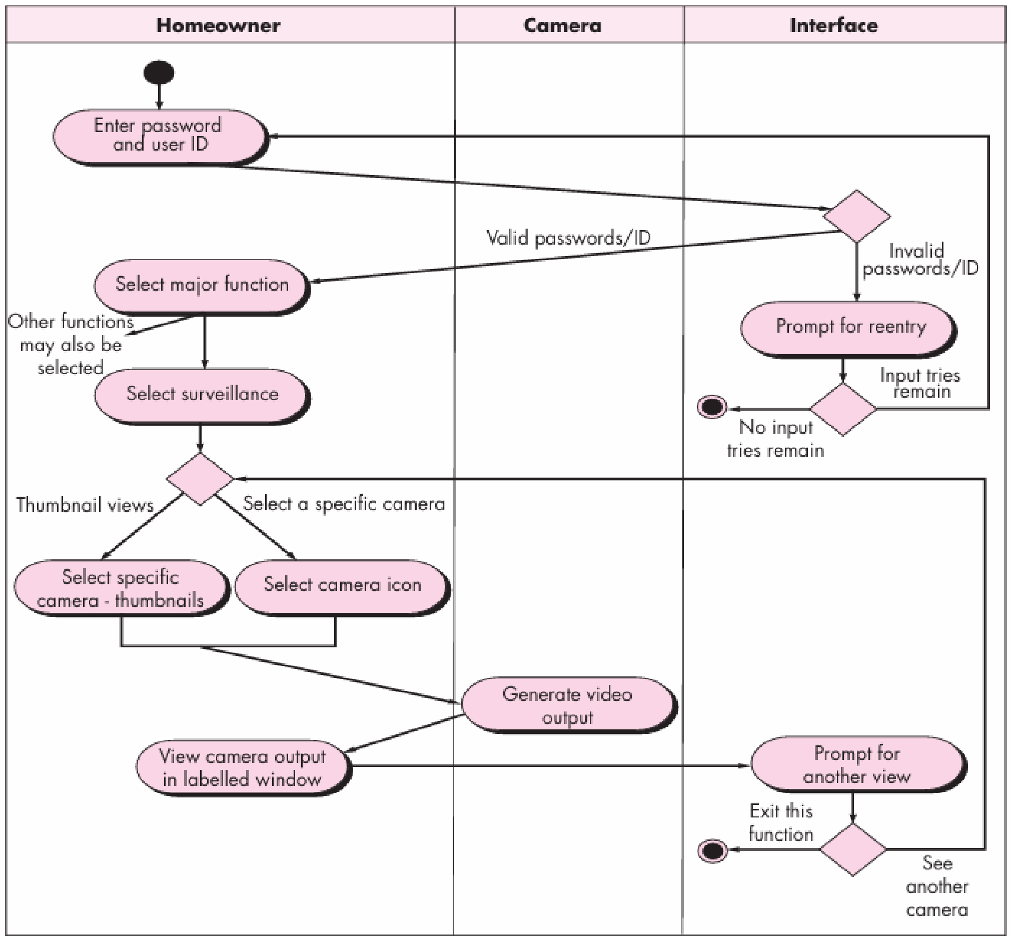
- **Use Cases**: Describe interactions between actors and the system.
- **Activity Diagrams**: Show the flow of activities within a use case.

- **Swimlane Diagrams**: Illustrate responsibilities of different actors or classes along with the flow of activities.

Information Space:

- **Data Requirements**: Beyond the procedural view, it's essential to represent and understand the data requirements of the system.
- **Upcoming Sections**: Will explore how to model the information space to ensure comprehensive coverage of all system requirements.

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



Data Modeling for Requirements

Purpose: In software requirements modeling, creating a data model is crucial, especially when there is a need to interact with a database or handle complex data structures. Data modeling helps in defining and understanding data objects, their attributes, and their relationships.

Entity-Relationship Diagram (ERD)

Entity-Relationship Diagram (ERD):

- **Purpose**: Represents all data objects that are processed within an application and illustrates the relationships between these objects.
- **Components**: Data objects, attributes, and relationships.

Data Objects

Definition: A data object is a composite piece of information that is processed within the system. It consists of multiple attributes.

Types of Data Objects:

1. **External Entity**: Produces or consumes information.
2. **Thing**: Example, a report or a display.
3. **Occurrence/Event**: Example, a telephone call or an alarm.
4. **Role**: Example, salesperson.
5. **Organizational Unit**: Example, accounting department.
6. **Place**: Example, a warehouse.
7. **Structure**: Example, a file.

Example: A car can be a data object defined by attributes such as make, model, ID number, body type, color, and owner.

Table Representation: Data objects can be represented in a tabular form where:

- **Headings**: Represent attributes of the data object.
- **Body**: Represents specific instances of the data object.

Data Attributes

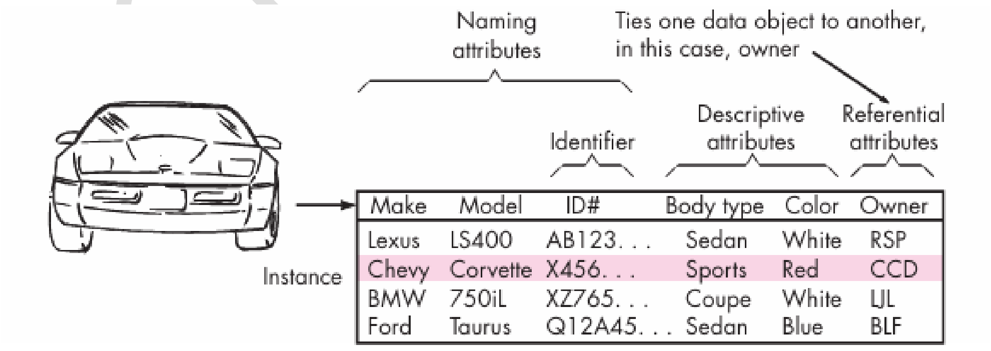
Definition: Properties that define a data object. Attributes can:

1. **Name** an instance of the data object.
2. **Describe** the instance.
3. **Reference** another instance in another table.

Identifier: One or more attributes that serve as keys to uniquely identify an instance of the data object. For example, the ID number for a car.

Example: For a department of motor vehicles, attributes for the data object car might include make, model, ID number, body type, color, and owner. For a manufacturing control software, additional attributes like interior code, drive train type, trim package designator, and transmission type might be needed.

Tabular representation of data objects



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

Relationships

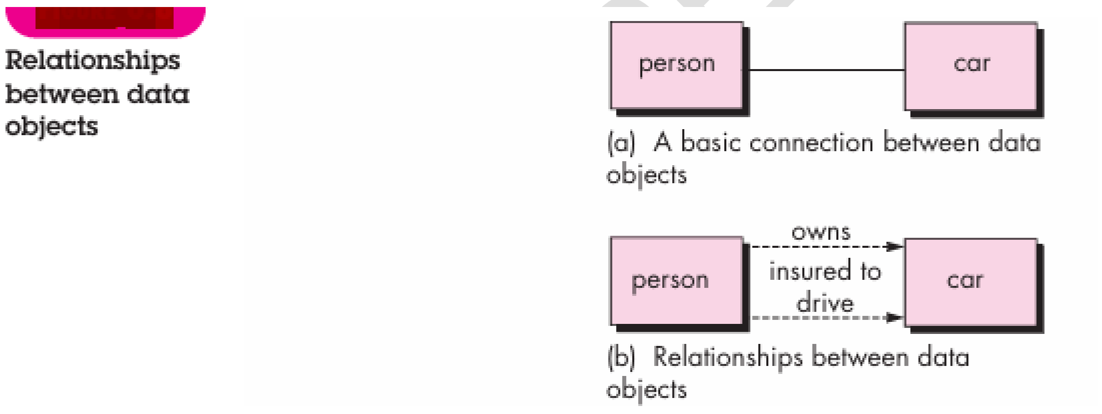
Definition: Connections between data objects that define how they interact or relate to one another.

Example Relationships Between Person and Car:

- **Owns**: A person owns a car.
- **Insured to Drive**: A person is insured to drive a car.

Graphical Representation:

- **Simple Connection**: Shows basic relationship (e.g., person and car are connected).
- **Object/Relationship Pairs**: Defines specific relationships (e.g., owns, insured to drive).
- **Directionality**: Arrows indicate the direction of the relationship, which helps in reducing ambiguity.


Relationships between data objects
person
car
(a) A basic connection between data objects
person
owns
insured to drive
car
(b) Relationships between data objects

Class-Based Modeling

1. **Definition and Purpose**:
   - Class-based modeling represents the objects the system manipulates, operations on those objects, relationships between objects, and collaborations among classes.

- It includes elements such as classes, objects, attributes, operations, CRC (Class-Responsibility-Collaborator) models, collaboration diagrams, and packages.

2. **Identifying Analysis Classes**:
    - Start with usage scenarios from the requirements model.
    - Perform a grammatical parse on use cases to identify nouns or noun phrases as potential classes.
    - Synonyms should be noted, and classes needed for the solution are part of the solution space; those needed only for description are in the problem space.
3. **Categories of Analysis Classes**:
    - **External entities**: Other systems, devices, people.
    - **Things**: Reports, displays, letters, signals.
    - **Occurrences/Events**: Property transfers, completion of robot movements.
    - **Roles**: Manager, engineer, salesperson.
    - **Organizational units**: Division, group, team.
    - **Places**: Manufacturing floor, loading dock.
    - **Structures**: Sensors, vehicles, computers.
4. **Examples and Non-Examples of Classes**:
    - Classes should not have imperative procedural names.
    - Example: An image is a class; image inversion is an operation on the image class, not a separate class.

5. **Applying Selection Characteristics** (Coad and Yourdon's guidelines):
   - **Retained information**: Must retain information for system functionality.
   - **Needed services**: Must have identifiable operations that can change its attributes.
   - **Multiple attributes**: Should have multiple significant attributes.
   - **Common attributes**: Attributes apply to all instances.
   - **Common operations**: Operations apply to all instances.
   - **Essential requirements**: Produce or consume essential information for the system.
6. **Example of Analysis for SafeHome Security Function**:
   - Grammatical parsing of a narrative identifies potential classes.
   - Example classes: Homeowner, sensor, control panel, installation, system, number, type, master password, telephone number, sensor event, audible alarm, monitoring service.
7. **Selection and Evaluation of Classes**:
   - Apply selection characteristics to potential classes.
   - Example evaluation:
     - **Accepted**: Sensor, control panel, system, sensor event, audible alarm.
     - **Rejected**: Homeowner, installation, number, type, master password, telephone number, monitoring service.
8. **Notes**:

  - The list of classes is not exhaustive; additional classes may be needed.
  - Rejected classes may become attributes of accepted classes.
  - Different problem statements may lead to different decisions regarding class acceptance or rejection.

Specifying Attributes in Class-Based Modeling

1. **Purpose of Attributes**:
   - Attributes define a class and clarify its meaning in the problem space.
   - The relevance of attributes depends on the context in which the class is used.
2. **Context-Specific Attributes**:
   - Attributes for the same class can vary significantly based on the context of the system.
   - Example:
     - For a baseball statistics system: name, position, batting average, fielding percentage, years played, games played.
     - For a baseball pension system: average salary, credit toward full vesting, pension plan options chosen, mailing address.
3. **Developing Attributes**:
   - Study each use case to identify relevant attributes that belong to the class.
   - Answer the question: "What data items (composite and/or elementary) fully define this class in the context of the problem at hand?"
4. **Example: SafeHome System Class Attributes**:
   - **Identification Information**:
     - System ID
     - Verification phone number
     - System status
   - **Alarm Response Information**:

    - Delay time
    - Telephone number
  - **Activation/Deactivation Information**:
    - Master password
    - Number of allowable tries
    - Temporary password

5. **Composite Data Items**:
   - Each composite data item can be further broken down into elementary data items if needed.
   - For practical purposes, composite data items provide a reasonable list of attributes.
6. **Class vs. Attribute**:
   - Avoid defining an item as an attribute if multiple instances of the item are associated with the class.
   - Example: Sensors are not attributes of the System class but are defined as a separate Sensor class associated with the System class.

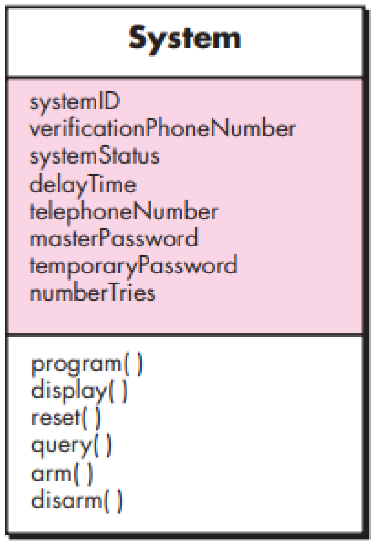
Defining Operations in Class-Based Modeling

1. **Purpose of Operations**:
   - Operations define the behavior of an object.
   - They manipulate data, perform computations, inquire about object states, or monitor for events.
2. **Categories of Operations**:
   - **Data manipulation**: Adding, deleting, reformatting, selecting data.
   - **Computation**: Performing calculations or processing data.
   - **State inquiry**: Checking or retrieving the state of an object.
   - **Event monitoring**: Watching for specific events to occur.
3. **Relationship with Attributes and Associations**:

  - Operations work with the class's attributes and associations.
  - They require knowledge of these elements to function correctly.

4. **Deriving Operations from Use Cases**:
   - Study processing narratives or use cases to identify relevant operations.
   - Use grammatical parsing to isolate verbs, which often indicate operations.
5. **Example: SafeHome System Operations**:
   - From the narrative:
     - "Sensor is assigned a number and type" suggests an **assign()** operation for the **Sensor** class.
     - "A master password is programmed for arming and disarming the system" suggests a **program ()** operation for the **System** class.
     - **arm()** and **disarm()** operations for the **System** class.
6. **Refining Operations**:
   - Initial operations might be high-level and can be broken down into more specific suboperations.
   - Example: **program()** might include suboperations like specifying phone numbers, configuring system characteristics, and entering passwords.
7. **Communication Between Objects**:
   - Objects interact by passing messages to each other.
   - Understanding this communication helps identify additional operations.

Class diagram for the system class



By following these guidelines, operations for each class can be effectively identified and defined, ensuring that the behavior of each object in the system is thoroughly understood and documented.

Class-Responsibility-Collaborator (CRC) Modeling

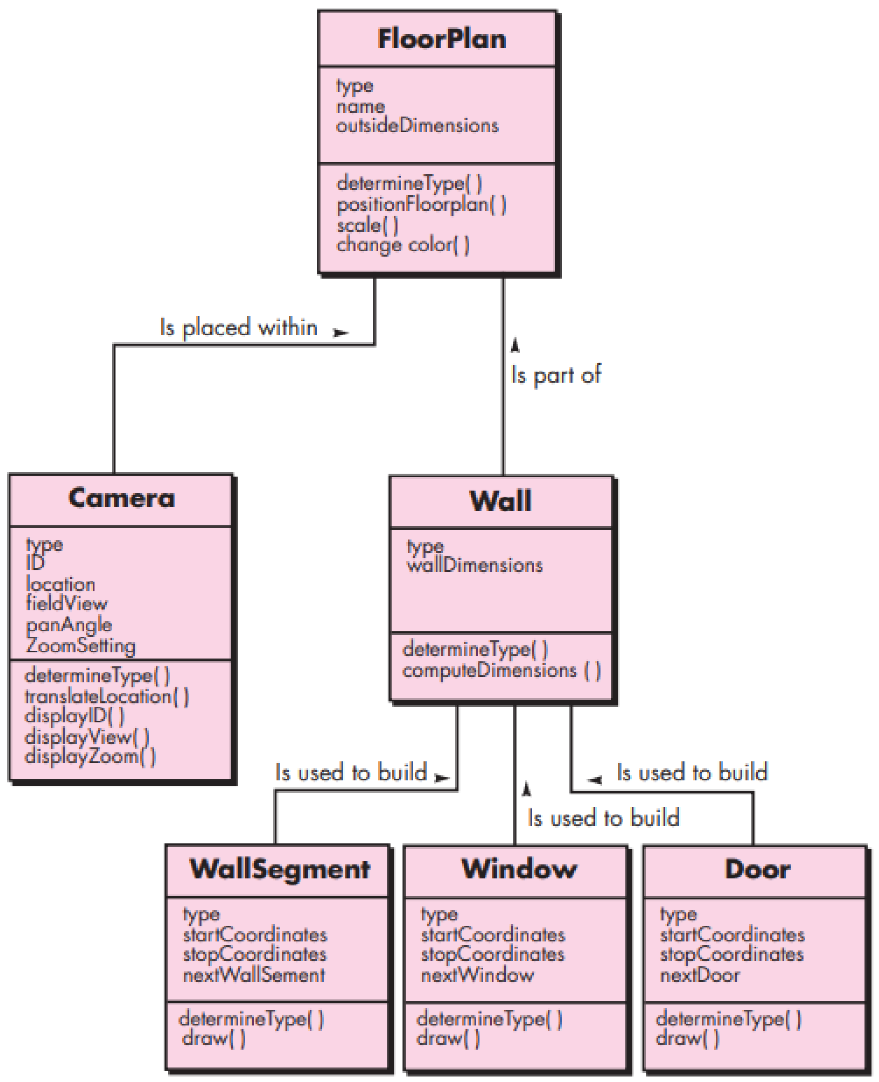
CRC modeling is a technique used to identify and organize classes relevant to system or product requirements. This model is typically represented using index cards, each card detailing a class, its responsibilities, and its collaborators.

CRC Card Structure

Each CRC card is divided into three sections:

1. **Class Name**: The name of the class.
2. **Responsibilities**: Attributes and operations that define what the class knows or does.
3. **Collaborators**: Other classes that provide necessary information or actions for the class to fulfill its responsibilities.

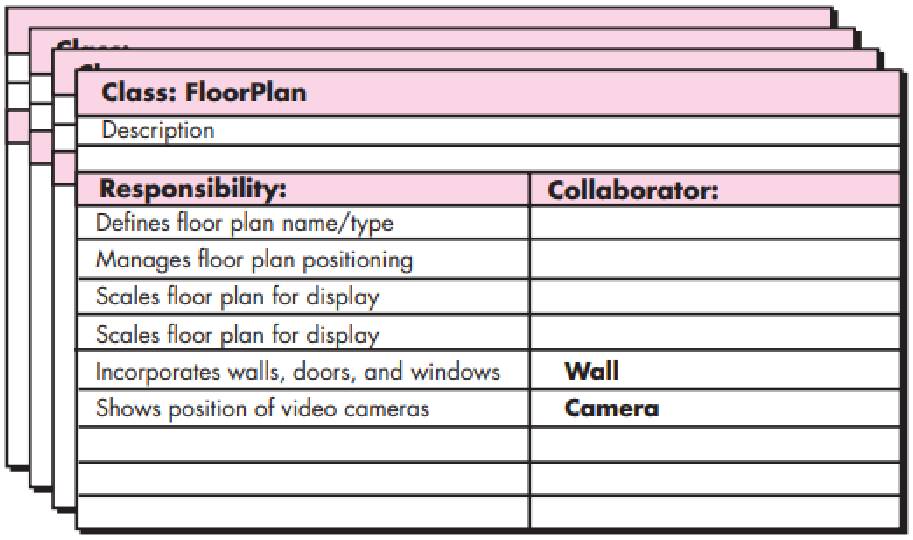
Class diagram for FloorPlan (see sidebar discussion)



Identifying Classes

Classes can be categorized into:

1. **Entity Classes**: Represent entities derived from the problem statement, e.g., FloorPlan, Sensor.
2. **Boundary Classes**: Interface elements the user interacts with, e.g., CameraWindow.
3. **Controller Classes**: Manage operations from start to finish, e.g., creation or update of entity objects, complex communication, validation of data.



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**

A CRC model index card

Identifying Responsibilities

Responsibilities are anything a class knows or does, including both attributes and operations. The following guidelines help in allocating responsibilities:

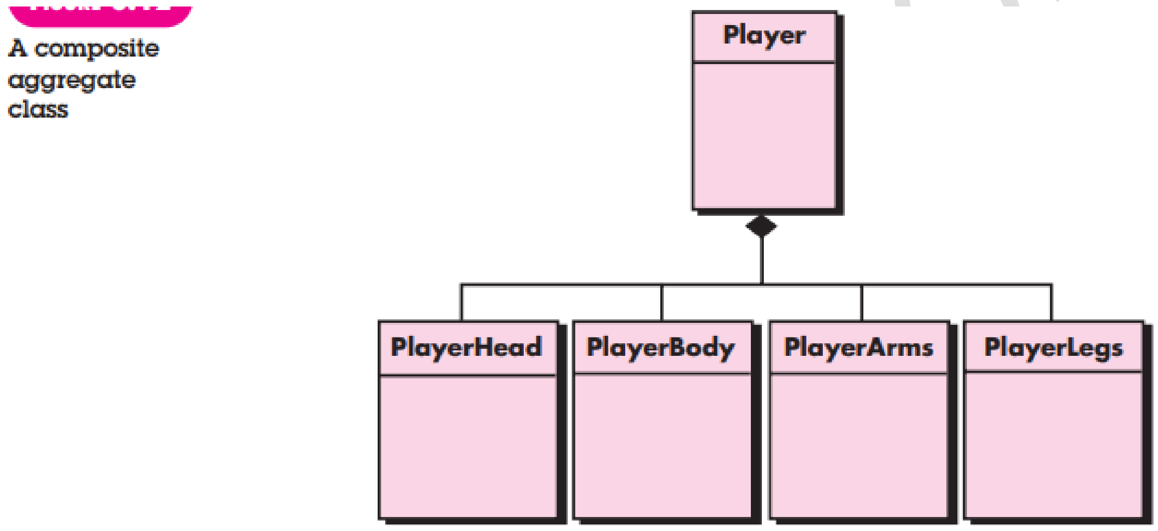
1. **Distribute System Intelligence**: Ensure intelligence is evenly distributed across classes to improve cohesiveness and maintainability.
2. **General Responsibilities**: State responsibilities generally to allow for high-level abstraction and reuse.
3. **Encapsulation**: Keep information and behavior related to it within the same class.
4. **Localize Information**: Information about one thing should reside within a single class.
5. **Share Responsibilities When Appropriate**: Related classes should share responsibilities when they need to exhibit the same behavior.

Identifying Collaborations

Collaborations represent requests from one class (client) to another (server) to fulfill a responsibility. These collaborations are necessary when a class cannot fulfill a responsibility on its own.

Three types of relationships help identify collaborators:

1. **Is-Part-Of**: Aggregate relationships where one class is part of another, e.g., PlayerBody is part of Player.
2. **Has-Knowledge-Of**: One class acquires information from another, e.g., ControlPanel determining sensor status from Sensor.
3. **Depends-Upon**: Classes dependent on each other without direct knowledge, facilitated by a third class.



Review Process

The CRC model is reviewed through a collaborative process:

1. **Distribute CRC Cards**: Participants receive a subset of CRC cards, ensuring no one has collaborating cards.
2. **Organize Use-Case Scenarios**: Use-case scenarios are categorized.
3. **Read Use Case**: The review leader reads the use case, passing a token to the holder of the corresponding class card.
4. **Describe Responsibilities**: The cardholder describes the responsibilities, verifying if they satisfy the use-case requirement.

5. **Modify Cards**: If necessary, define new classes or modify responsibilities and collaborations on existing cards.

This process ensures the model accurately represents the requirements and guides further development.

Associations and Dependencies

In UML (Unified Modeling Language), associations and dependencies represent relationships between classes.

Associations

Definition: Associations are relationships between two or more classes. These relationships often signify how objects of one class interact or are connected to objects of another class.

Example: In the context of a floor plan system, a FloorPlan class may be associated with Camera and Wall classes. Similarly, a Wall class might be associated with WallSegment, Window, and Door classes.

Multiplicity: Associations can have multiplicity, which defines how many instances of one class are associated with instances of another class.

Multiplicity Notation:

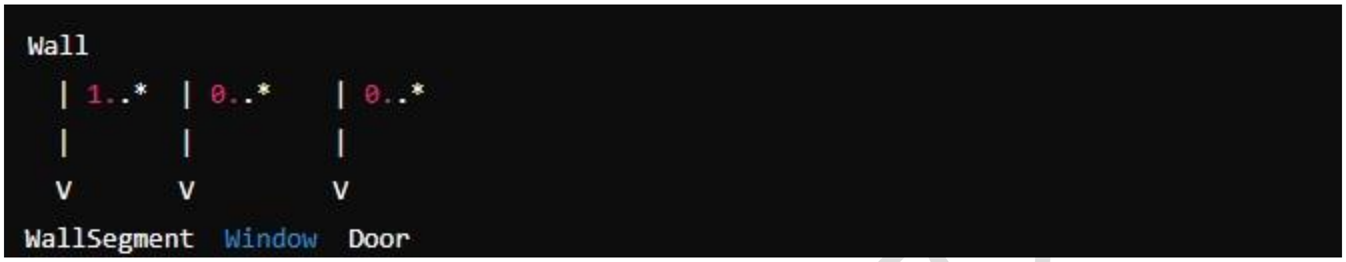
- 1..* indicates one or more instances.
- 0..* indicates zero or more instances.

Example of Multiplicity:

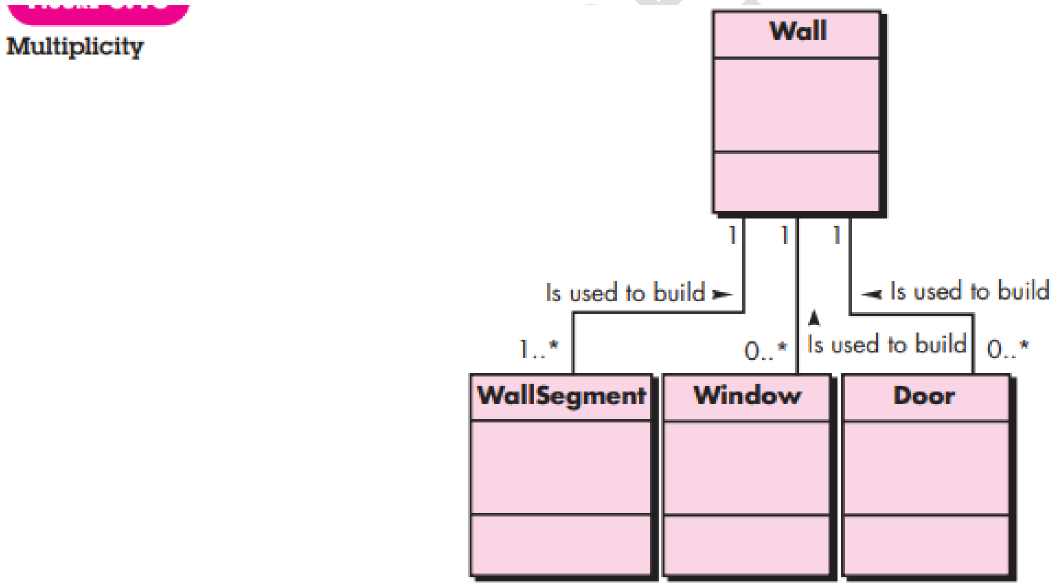
- A Wall object can be constructed from one or more WallSegment objects (1..*).

- A Wall object may contain zero or more Window objects (0..*) and zero or more Door objects (0..*).

The relationships and multiplicities can be illustrated in UML diagrams, as shown below:



```
Wall
  | 1..*  | 0..*    | 0..*
  |       |         |
  v       v         v
WallSegment  Window  Door
```



Dependencies

Definition: Dependencies represent a client-server relationship where one class (the client) depends on another class (the server) for some functionality or service. A dependency indicates that a change in the server class may affect the client class.

Stereotypes: Dependencies are often defined using stereotypes, which are special modeling elements with custom-defined semantics. In UML, stereotypes are enclosed in double angle brackets (e.g., <<stereotype>>).

Example: In the SafeHome surveillance system, a Camera object (server class) provides video images to a Display Window object (client class). This relationship is more complex than a simple association and can be represented as a dependency.

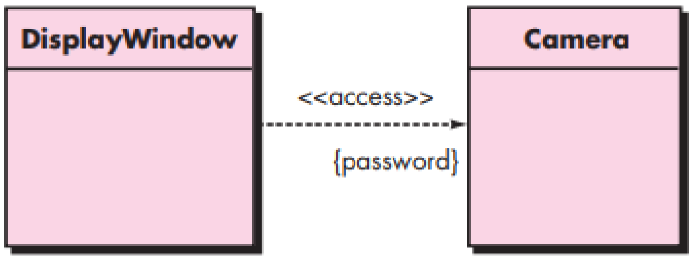
Example of Dependency:

- The Display Window depends on the Camera for video images. Additionally, access to specific camera views may be controlled by a special password.

This can be represented as shown in **Figure 6.14**: DisplayWindow <<access>> Camera

In this diagram, `<<access>>` implies that the use of the camera output by the `DisplayWindow` is controlled by a special password.

Dependencies



Associations: Represent general relationships between classes and may include multiplicities to indicate how many instances of one class relate to instances of another.

Dependencies: Indicate that one class relies on another, often represented with stereotypes to provide additional semantics. These relationships are crucial for understanding the interactions and dependencies within a system.

Analysis Packages

Analysis packages are used in analysis modeling to group related elements, such as use cases and analysis classes, into categorized units. This organization helps manage complexity by structuring the model into coherent sections.

Example: Video Game Analysis Packages

Consider a video game analysis model, which consists of numerous classes. These classes can be categorized into different analysis packages based on their roles and functionalities within the game.

Categories of Classes:

1. **Game Environment**:
    - Classes that describe the visual scenes and environment elements the user sees during gameplay.
    - Examples: Tree, Landscape, Road, Wall, Bridge, Building, VisualEffect.
2. **Game Characters**:
    - Classes that define the physical features, actions, and constraints of characters within the game.
    - Examples: Player, Protagonist, Antagonist, SupportingRoles.
3. **Game Rules**:
    - Classes that define the rules of the game, including how players navigate the environment and interact with objects and characters.
    - Examples: RulesOfMovement, ConstraintsOnAction.

Illustration:

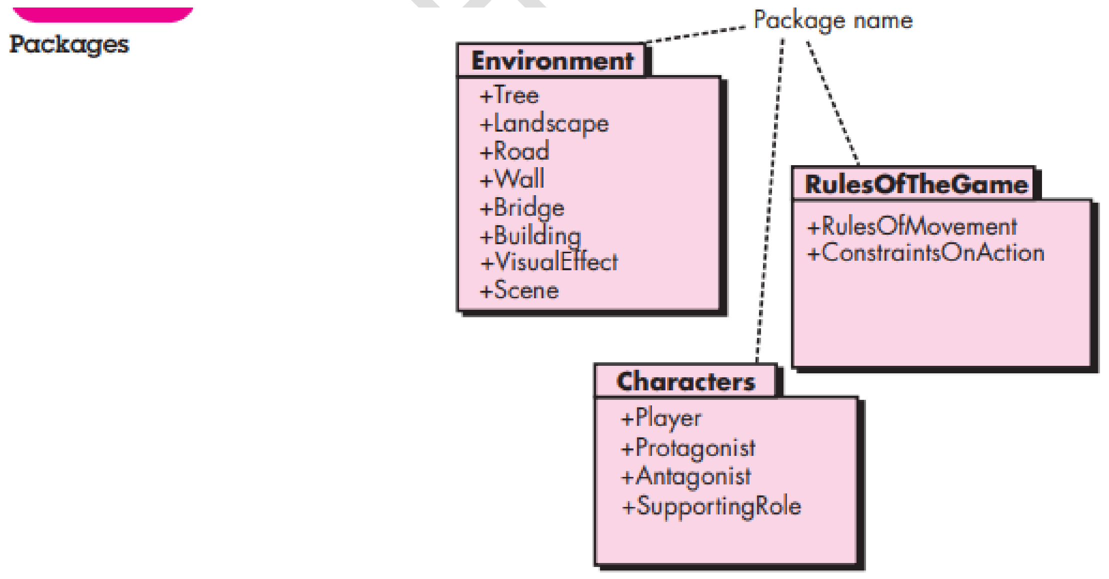
Figure 6.15: Analysis Packages for a Video Game

Visibility Symbols

In UML, symbols are used to indicate the visibility of elements within and across packages:

- **Public (+)**: The element is accessible from other packages.
- **Private (-)**: The element is hidden from other packages.
- **Protected (#)**: The element is accessible only to packages contained within a given package.

In **Figure 6.15**, the plus sign (+) before the class names indicates that these classes have public visibility, meaning they can be accessed from other packages. Other symbols (minus sign - for private and hash # for protected) are not shown in this figure but are used similarly to control the visibility of elements.


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