Module - II

Understanding Requirements: Requirements Engineering, Establishing the ground work, Eliciting Requirements, Developing use cases, Building the requirements model, Negotiating
Requirements, Validating Requirements
Textbook 1: Chapter 5: 5.1 to 5.7
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.
Textbook 1: Chapter 6: 6.1 to 6.5

REQUIREMENTS ENGINEERING

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called ***requirements engineering***. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirements engineering builds a bridge to design and construction. Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: ***inception, elicitation, elaboration, negotiation, speci□ication, validation,*** and ***management***.

a) **Inception.** In general, most projects begin when a business need is identified or a potential new market or service is discovered. Stakeholders from the business community define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.

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

b) **Elicitation**. Ask the customer, what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. A number of problems that are encountered as elicitation occurs.

- **Problems of scope**. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding**. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or un testable.
- **Problems of volatility**. The requirements change over time. To help overcome these problems, you must approach requirements gathering in an organized manner.

c) **Elaboration.** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.

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

d) **Negotiation**. It usual for customers, to given limited business resources. It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to *rank requirements* and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

e) **Specification.** Specification means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. Some suggest that a "standard template" should be developed and used for a specifcation, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.

ESTABLISHING THE GROUNDWORK

In an ideal setting, stakeholders and software engineers work together on the same team. In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team.

We discuss the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.

Identifying Stakeholders: Stakeholder is "anyone who benefits in a direct or indirect way from the system which is being developed." The usual stakeholders are: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers. Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.

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 constituencies 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. You should categorize all stakeholder information in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

Working toward Collaboration: If five stakeholders are involved in a software project, you may have five different opinions about the proper set of requirements. Customers must collaborate among themselves and with software engineering practitioners if a successful system is to result. The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" may make the final decision about which requirements make the cut.

Asking the First Questions: Questions asked at the inception of the project should be "context free. The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. You might ask:

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

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

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

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

The final set of questions focuses on the effectiveness of the communication activity itself.

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

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

ELICITING REQUIREMENTS

Requirements elicitation combines elements of problem solving, elaboration, negotiation,

and specification.

Collaborative Requirements Gathering: Many different approaches to collaborative requirements gathering have been proposed.

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts etc) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions.

The mini-specs are presented to all stakeholders for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is maintained so that these ideas will be acted on later.

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:

Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.

Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

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

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

Usage Scenarios: As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed.

Elicitation Work Products: The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

DEVELOPING USE CASES

In essence, a 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.

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. It is important to note that an actor and an end user are not necessarily the same thing. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system.

Primary actors interact to achieve required system function and derive the intended benefit from the system. They 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. *Jacobson* suggests a number of questions that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?

- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

BUILDING THE REQUIREMENTS MODEL

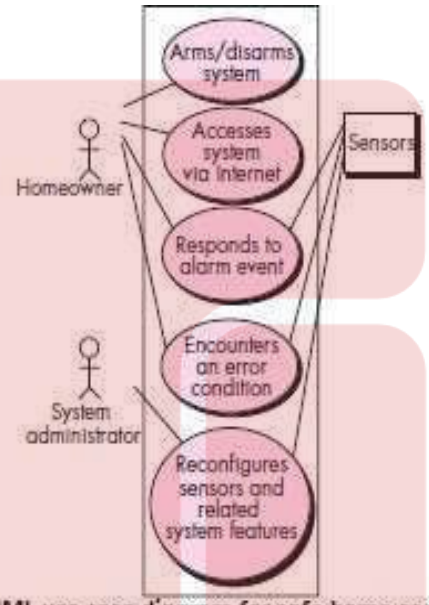


Fig 5.2: UML use case diagram for safe home security function

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand

more about what they really require. For that reason, the *analysis model* is a snapshot of requirements at any given time.

Elements of the Requirements Model: There are many different ways to look at the requirements for a computer-based system. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use-case diagrams evolve into more elaborate template-based use cases. Scenario-based elements

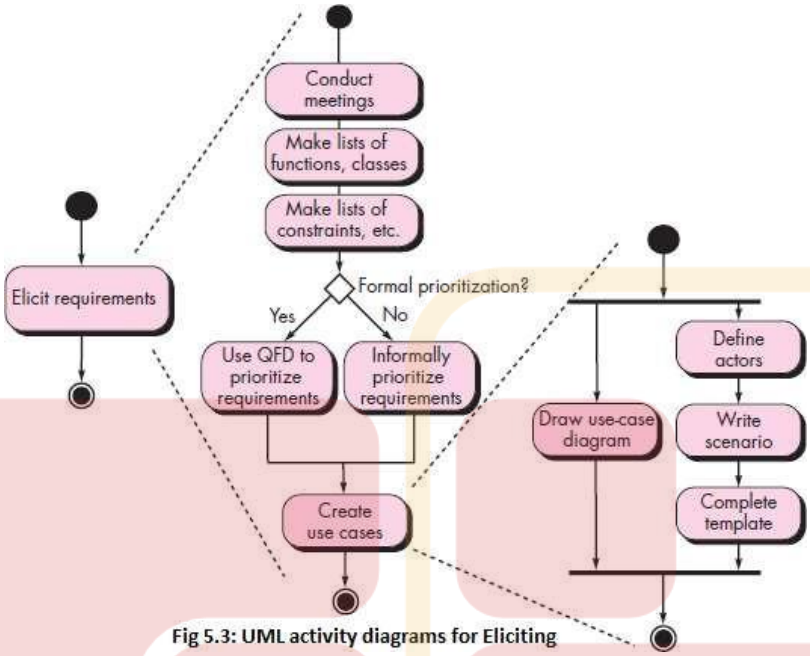


Fig 5.3: UML activity diagrams for Eliciting

of the requirements model are often the first part of the model that is developed. Three levels of elaboration are shown, culminating in a scenario-based representation.

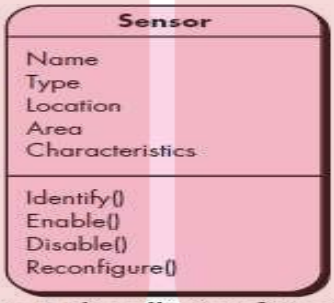


Fig 5.4: Class diagram for sensor

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. Example shown in figure.

Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A state is any externally observable mode of behavior. In addition, the state diagram indicates actions taken as a consequence of a particular ***event***.

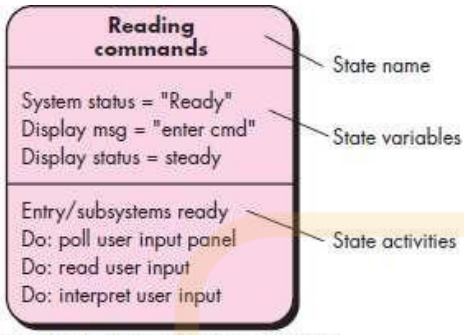


Fig 5.5: UML State diagram notation

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system.

Analysis Patterns: Anyone who has done requirements engineering on more than a few software projects begins to notice that certain problems reoccur across all projects within a specific application domain. These analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications. Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them. Information about an analysis pattern (and other types of patterns) is presented in a standard template.

NEGOTIATING REQUIREMENTS

In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities. You may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market. The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real- world constraints (e.g., time, people, budget) that have been placed on the software team.

The best negotiations strive for a "win-win" result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to

realistic and achievable budgets and deadlines.

Boehm [Boe98] defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.

VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

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

Have all patterns been properly validated? Are all patterns consistent with customer requirements?

Requirements Modeling (Scenarios, Information and Analysis Classes)

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an "analyst") builds the model using
requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. In this chapter you'll consider requirements modeling from three different perspectives: scenario-based models, data (information) models, and class- based models. Each represents requirements in a different "dimension," thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions willbe uncovered.

What are the steps? Scenario-based modeling represents the system from the user's point of view. Data modeling represents the information space and depicts the data objects thatthe software will manipulate and the relationships among them. Class-based modeling defines objects, attributes, and relationships. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency.

What is the work product? A wide array of text based and diagrammatic forms may be chosen for the requirements model. Each of these representations provides a view of one or more of the model elements.

How do I ensure that I've done it right? Requirements modeling work products must bereviewed for correctness, completeness, and consistency. They must reflect the needs of all stakeholders and establish a foundation from which design can be conducted.

2.0.1 REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering. The requirements modeling action results in one or more of the following types of models:

- Scenario-based models of requirements from the point of view of various system "actors"
- Data models that depict the information domain for the problem
- Class-oriented models that represent object-oriented classes (attributes and operations) andthe manner in which classes collaborate to achieve system requirements
- Flow-oriented models that represent the functional elements of the system and how theytransform data as it moves through the system

• Behavioral models that depict how the software behaves as a consequence of external
"events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

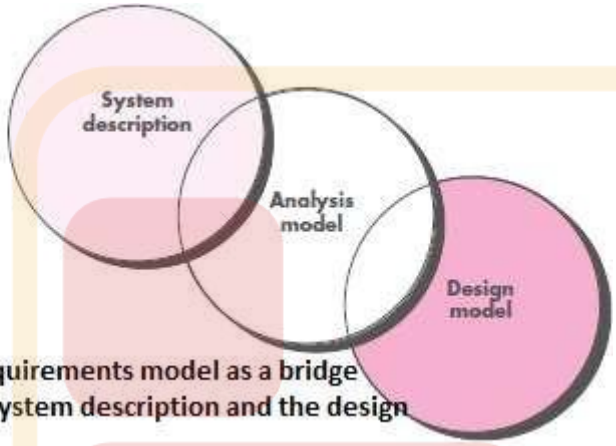


Fig 6.1: The requirements model as a bridge between the system description and the design model

Overall Objectives and Philosophy: Throughout requirements modeling, your primary focus is on ***what***, not ***how***. The requirements model must achieve three primary objectives: (1)to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design This relationship is illustrated in Figure 6.1

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

• *The model should focus on requirements that are visible within the problem or business domain.* The level of abstraction should be relatively high. "Don't get bogged down in details"that try to explain how the system will work.

• Each element of the requirements *model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system*.

• *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

• *Minimize coupling* throughout the system. It is important to represent relationships

between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

• *Be certain that the requirements model provides value to all stakeholders*. Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

• *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new

information. Don't use complex notational forms, when a simple list will do.

Domain Analysis: The analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems. This improves time-to- market and reduces development costs.

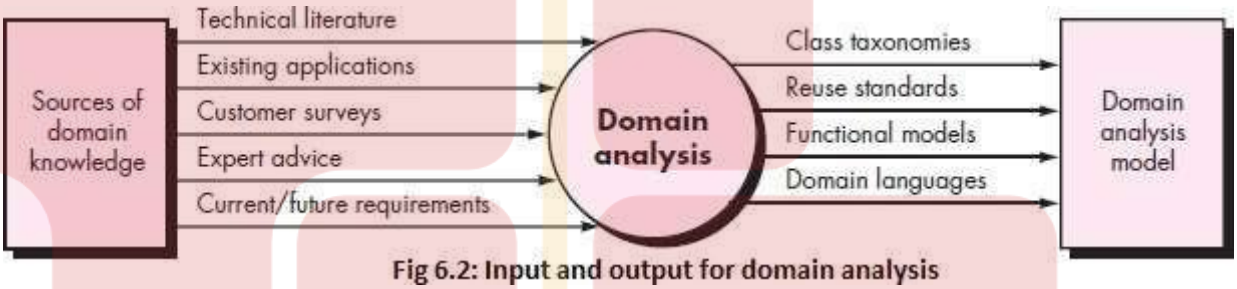


Fig 6.2: Input and output for domain analysis

Domain analysis doesn't look at a specific application, but rather at the domain in which theapplication resides. The intent is to identify common problem solving elements that areapplicable to all applications within the domain. Domain analysis may be viewed as an umbrellaactivity for the software process. Figure 6.2 illustrates key inputs and outputs for the domainanalysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

Requirements Modeling Approaches: One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their *attributes* and *relationships*. *Processes* that manipulate data objects are modeled in a manner that shows how they transformdata as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. *Scenario-based elements* depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

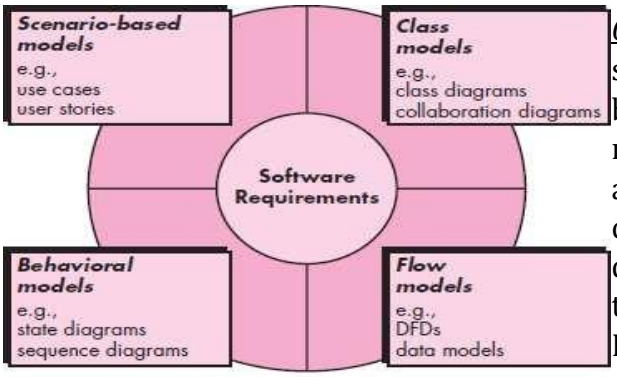


Fig 6.3: Elements of Analysis model

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. *Behavioral elements* depict how external events change the state of the system or the classes that reside within it. Finally, *flow- oriented elements* represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions. *Analysis modeling* leads to the derivation of each of these modeling elements. However, the specific content of each element may differ from project to project.

SCENARIO-BASED MODELING

Requirements modeling with UML begins with the creation of scenarios in the form of *use cases,activity diagrams, and swimlane diagrams.*

Creating a Preliminary Use Case: A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. But how do you know (1) whatto write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description?

What to write about? The first two requirements engineering tasks—inception and elicitation— provide you with the information you'll need to begin writing use cases. Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, establish priorities, outline all known functional requirements, and describe the things (objects) that will be manipulated by the system.
To begin developing a set of use cases, list the functions or activities performed by a specific actor. You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.

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 it be?

• Is it possible that the actor will encounter some other behavior at this point. If so, what it be? Answers to these questions result in the creation of a set of secondary scenarios that are part ofthe original use case but represent alternative behavior.
Can the actor take some other action at this point? The answer is "yes."
Is it possible that the actor will encounter some error condition at this point? Any number of errorconditions can occur as a computer-based system operates.
Is it possible that the actor will encounter some other behavior at this point? Again the answer to the question is "yes."
In addition to the three generic questions suggested, the following issues should also beexplored:
• Are there cases in which some "validation function" occurs during this use case? This implies that validation function is invoked and a potential error condition might occur.
• Are there cases in which a supporting function (or actor) will fail to respond appropriately? Forexample, a user action awaits a response but the function that is to respond times out.
• Can poor system performance result in unexpected or improper user actions?

Writing a Formal Use Case: The informal use cases presented 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.

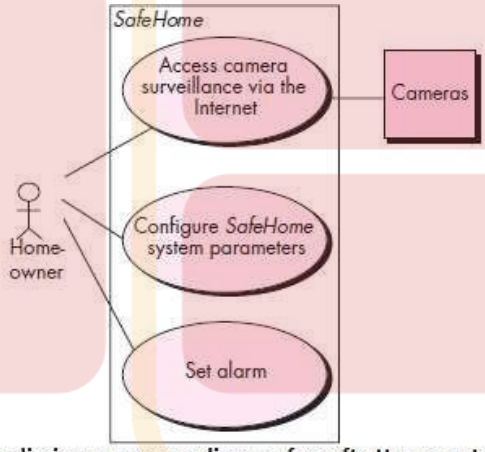


Fig 6.4: Preliminary use-case diagram for saftwHome system

UML MODELS THAT SUPPLEMENT THE USE CASE

A broad array of UML graphical models are as follows
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. An activity diagram for the ACS-DCV use case is shown in Figure 6.5.

Swimlane Diagrams: The UML swimlane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class has responsibility for the action described by anactivity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool. Refer Figure 6.6.

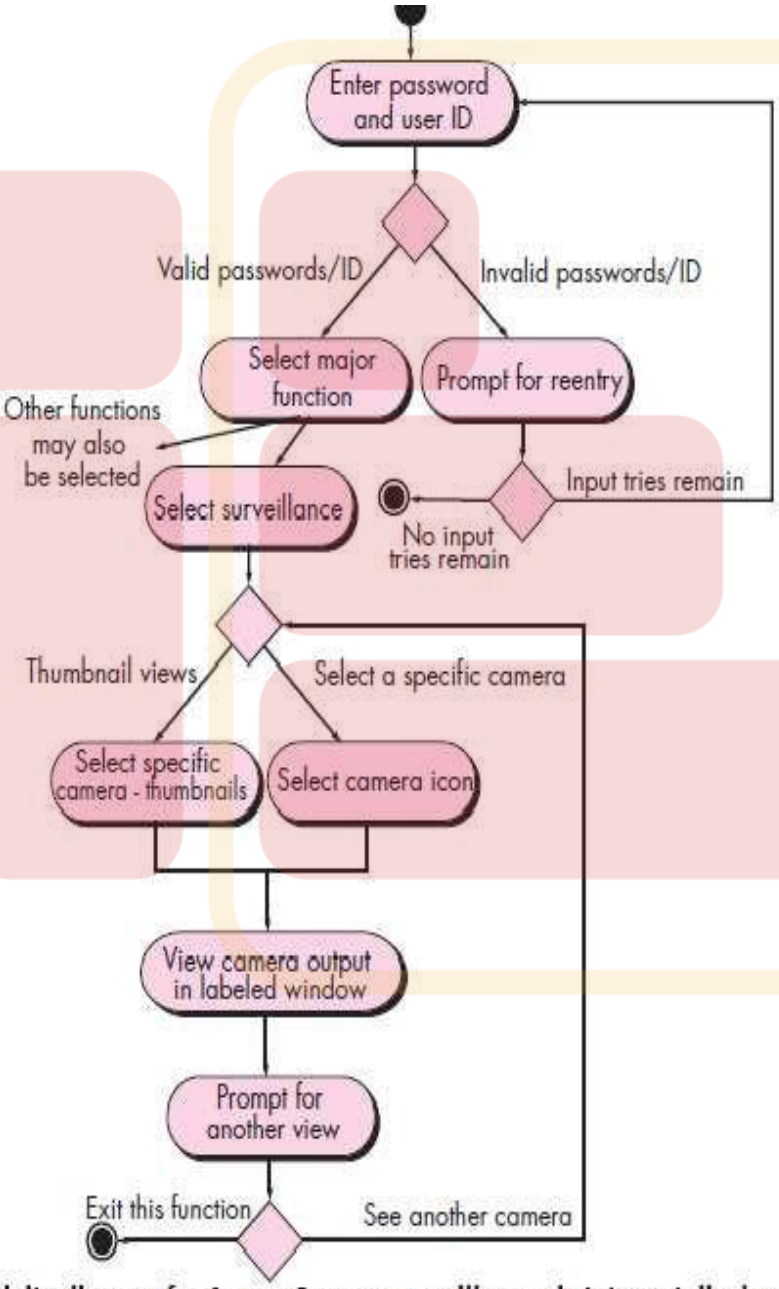
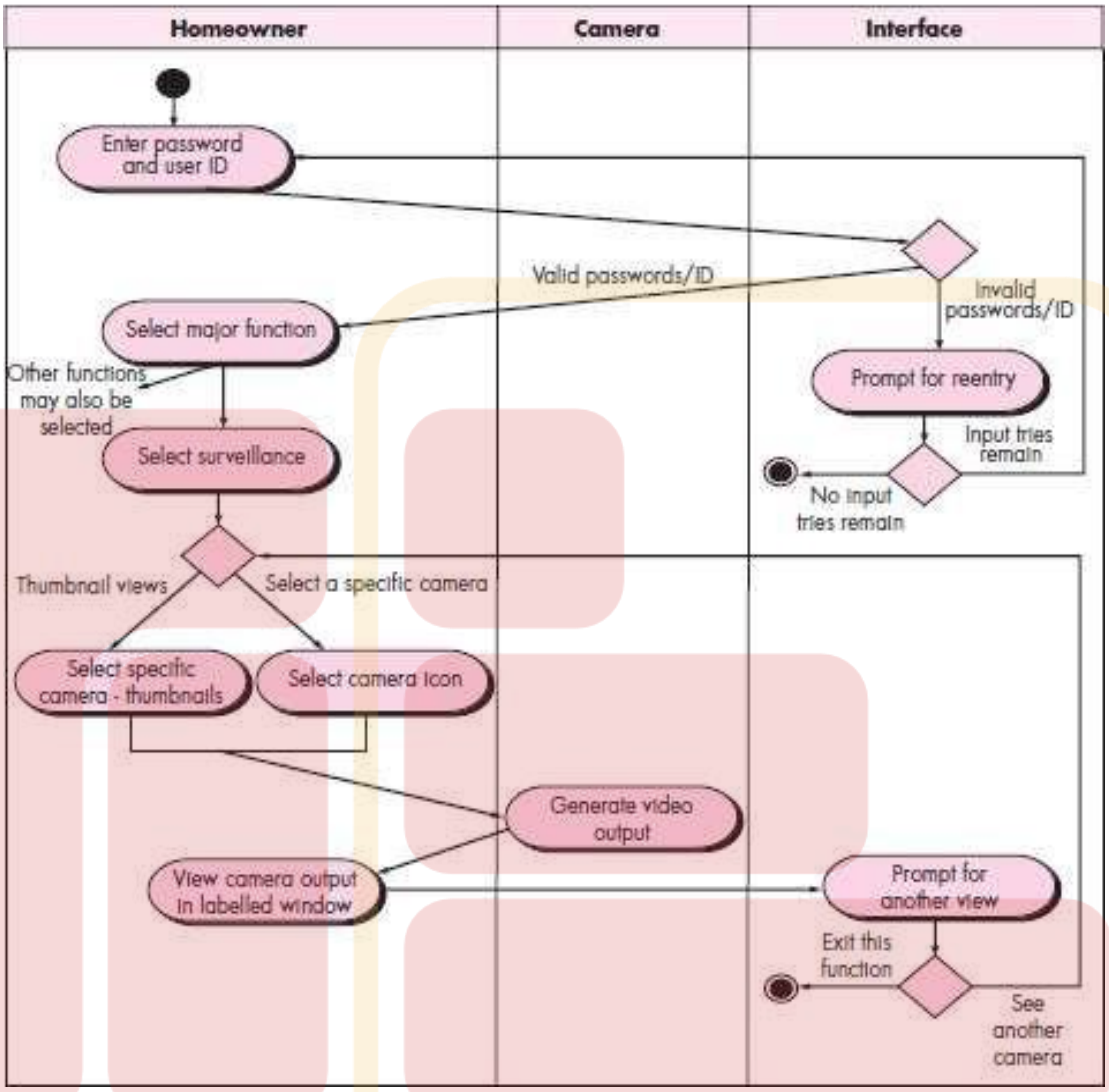


Fig 6.5: Activity diagram for Access Camera surveillance via Internet display cams

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



DATA MODELING CONCEPTS

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may chooseto 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.

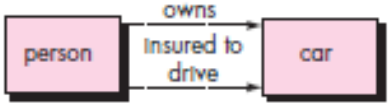
Data Objects: A *data object* is a representation of composite information that must be understood by software. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.
A data object can be an *external entity* (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). The description of the data object incorporates the data object and all of its *attributes.*

A data object *encapsulates* data only—there is no reference within a data object to operationsthat act on the data.

Data Attributes: Data attributes define the properties of a data object and take on one ofthree 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. In addition,one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement.

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 illustrated in Figure 6.8a. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you shouldunderstand the role of people (owners, in this case) and cars within the context of the softwareto be built. You can establish a set of object/relationship pairs that define the relevant relationships. For example,
• A person owns a car.
• A person is insured to drive a car.

(a) A basic connection between data objects



(b) Relationships between data objects

Fig 6.8: Relationships between data objects

The relationships owns and insured to drive define the relevant connections between person and car. Figure 6.8b illustrates these object-relationship pairs graphically. The

arrows noted in Figure

6.8b provide important information about the directionality of the relationship and often reduceambiguity or misinterpretations.

CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations thatwill be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class responsibility, collaborator(CRC) models, collaboration diagrams, and packages.

Identifying Analysis Classes

Classes are determined by underlining each *noun* or *noun phrase* and entering it into a simple table. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

Analysis classes manifest themselves in one of the following ways:

- *External entities* that produce or consume information to be used by a computer-basedsystem.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for theproblem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem andthe overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects orrelated classes of objects.

Specifying Attributes: Attributes are the set of data objects that fully define the class within the context of the problem. Attributes describe a class that has been selected for inclusionin the requirements model. In essence, it is the attributes that define the class—that clarify whatis meant by the class in the context of the problem space.

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

Defining Operations: Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way. (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event. These functions are accomplished by operating on attributes and/or associations. Therefore, an operation must have "knowledge" of the nature of the class' attributes and associations.

As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected

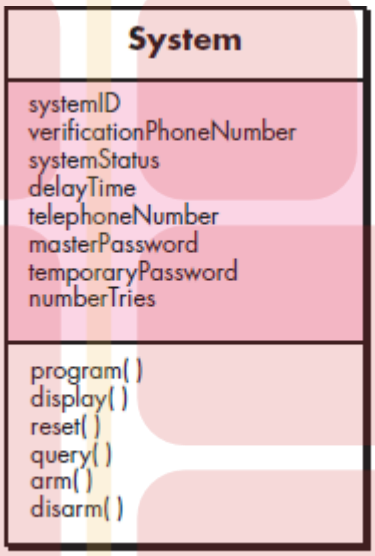


Fig 6.9: Class diagram for the System class

to a specific 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.

One purpose of CRC cards is to fail early, to fail often, and to fail inexpensively. It is a lot cheaper to tear up a bunch of cards than it would be to reorganize a large amount of source code.

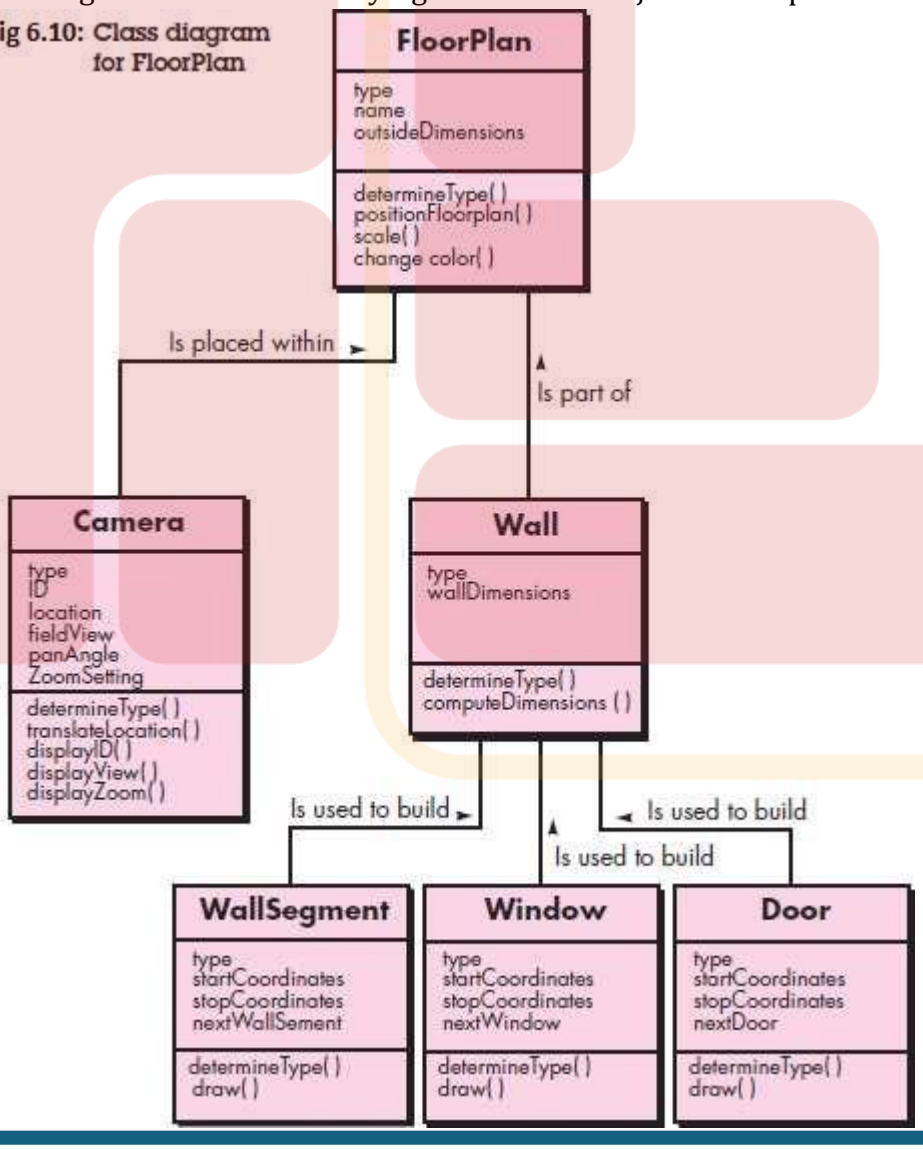
Ambler describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the *name of the class*. In the body of the card you list the *class responsibilities* on the left and the *collaborators* on the right.

In reality, the CRC model may make use of actual or virtual index cards. The intent is to developan organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does". Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.
In general, a collaboration implies either a request for information or a request for some action.

A simple CRC index card for the FloorPlan class is illustrated in Figure 6.11. The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes Wall and Camera are noted next to the responsibility that will require their collaboration.
Classes. Basic guidelines for identifying classes and objects were presented earlier in

Fig 6.10: Class diagram for FloorPlan



this chapter. The taxonomy of class types presented in Section 6.5.1 can be extended by considering the following categories:

• *Entity classes*, also called model or business classes, are extracted directly from the statement of the problem These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.

• *Boundary classes* are used to create the interface (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 are designedwith the responsibility of managing the way entity objects are represented to

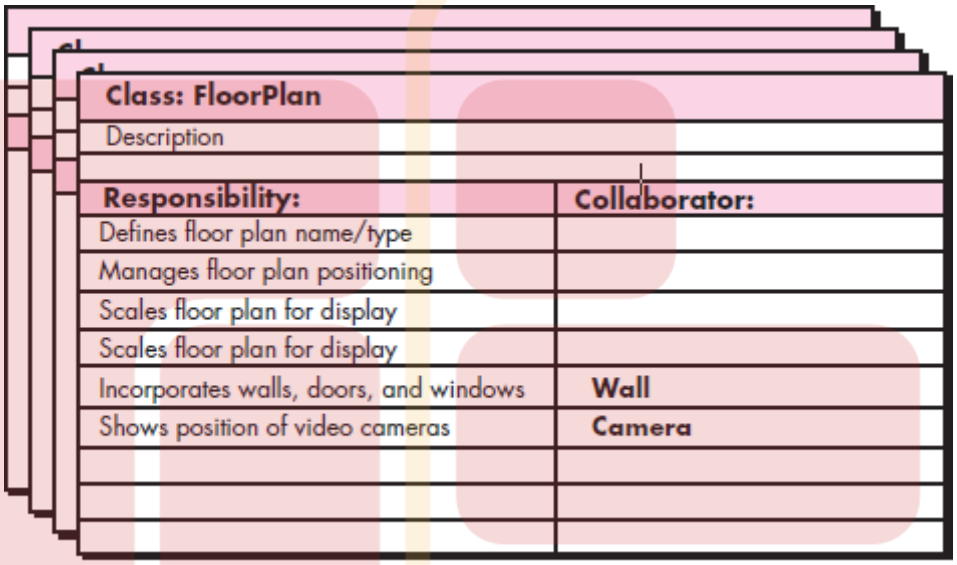


Fig 6.11: A CRC model index card

users.

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

Responsibilities. The five guidelines for allocating responsibilities to classes:

1. System intelligence should be distributed across classes to best address the needs ofthe problem. Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed acrosss classes in a umber of different ways. "Dumb" classes can be modeled to act as servants to a few "smart" classes.

To determine whether system intelligence is properly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has

an extraordinarilylong list of responsibilities. Example is aggregation of classes.

2. Each responsibility should be stated as generally as possible. This guideline implies thatgeneral responsibilities (both attributes and operations) should reside high in the class hierarchy

3. Information and the behavior related to it should reside within the same class. This achieves the object-oriented principle called encapsulation. Data and the processes that manipulate the data should be packaged as a cohesive unit.

4. Information about one thing should be localized with a single class, not distributed across multiple classes. A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared

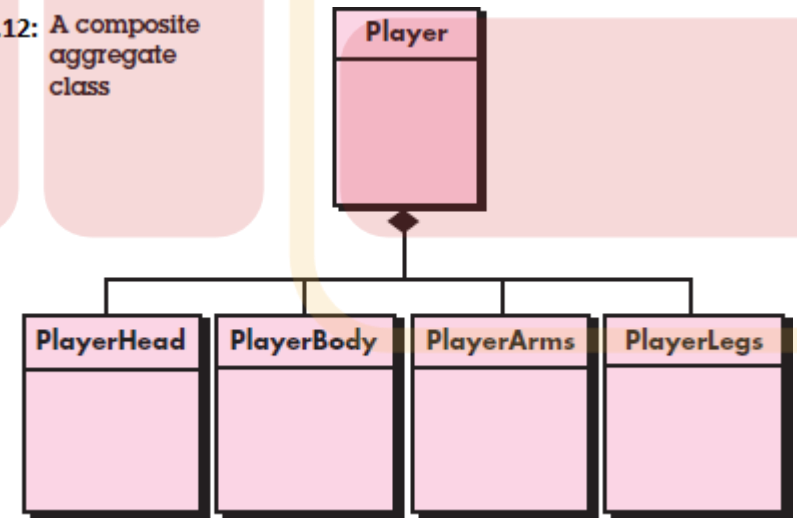
across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.

5. Responsibilities should be shared among related classes, when appropriate. There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

Collaborations. Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2)a class can collaborate with other classes.

To help in the identification of collaborators, you can examine three different generic relationships between classes (1) the is-part-of relationship, (2) the has-knowledge-of relationship, and (3) the depends-upon relationship.

Fig 6.12: A composite aggregate class



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

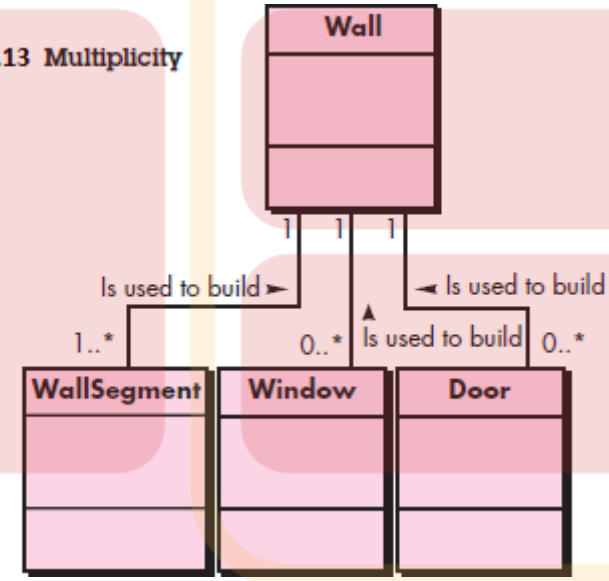
1. All participants in the review are given a subset of the CRC model index cards. Cards thatcollaborate should be separated

2. All use-case scenarios (corresponding use-case diagrams) should be organized intocategories.
3. The review leader reads the use case deliberately.
4. When the token is passed, the holder of the Sensor card is asked to describe the responsibilities noted on the card.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate theuse case, modifications are made to the cards.

This continues until the use case is finished. When all use cases have been reviewed, requirements modeling continue.

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*.
In some cases, an association may be further defined by indicating multiplicity. Referring to The multiplicity constraints are illustrated in Figure 6.13, where "one or more" is represented
using 1. .*, and "0 or more" by 0 . .*. In UML, the asterisk indicates an unlimited upper bound onthe range.

Fig 6.13 Multiplicity



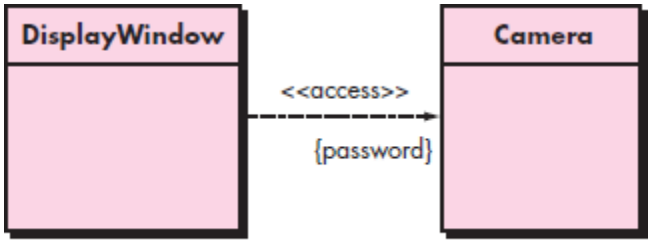


Fig 6.14: Dependencies

Analysis Packages: An important part of analysis modeling is *categorization*. That is, various elements of the analysis model are categorized in a manner that *packages* them as a grouping—called an *analysis package*—that is given a representative name. For example, Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and VisualEffectmight fall within this category. Others focus on the characters within the game, describing their physical features, actions, and constraints.

These classes can be grouped in analysis packages as shown in Figure 6.15. The *plus sign*preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.

Although they are not shown in the figure, other symbols can precede an element within a package. A *minus sign* indicates that an element is hidden from all other packages and a *#symbol* indicates that an element is accessible only to packages contained within a given package.

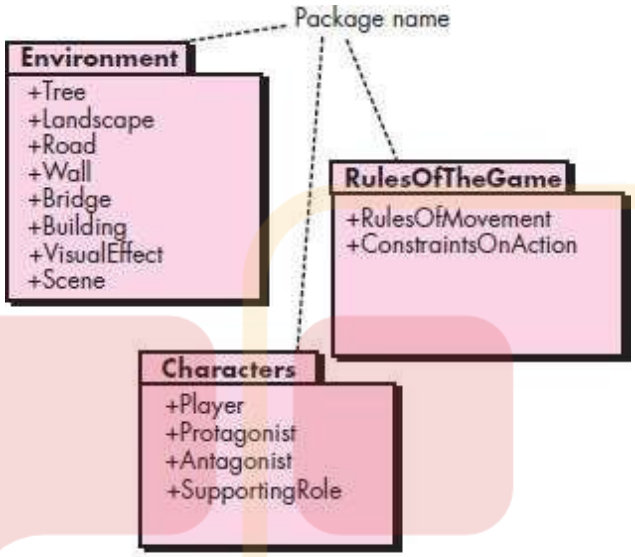


Fig 6.15: Packages