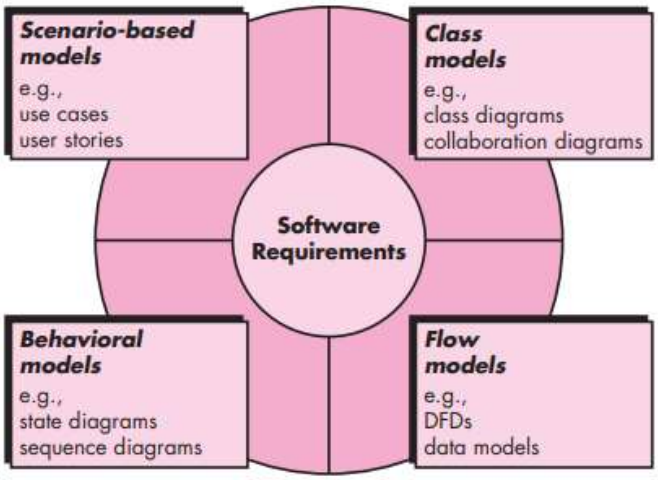
1. **Briefly explain the types of models in requirements modelling with relevant figures.**

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



2. **Explain the requirements modelling / analysis along with its primary objectives**

Throughout requirements modeling, your primary focus is on what, not how.

The customer may be unsure of precisely what is required for certain aspects of the system.

The developer may be unsure that a specific approach will properly accomplish function and performance.

These realities mitigate in favor of an iterative approach to requirements analysis and modeling.

The analyst should model what is known and use that model as the basis for design of the software increment.

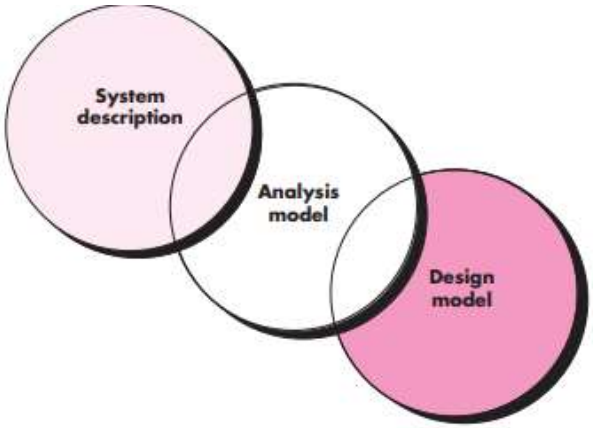
The requirements model must achieve three primary objectives:

(1) to describe what the customer requires,

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure



3. Explain various Analysis Rules of Thumb

• The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.

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

• Delay consideration of infrastructure and other non-functional 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.

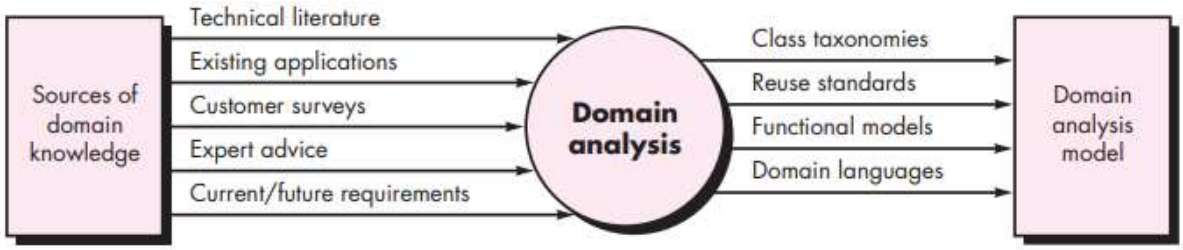
• Minimize coupling throughout the system. It is important to represent relationships between classes and functions.

• Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model.

• Keep the model as simple as it can be. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

4. Explain Domain Analysis in understanding requirements phase

applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.



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

The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment.

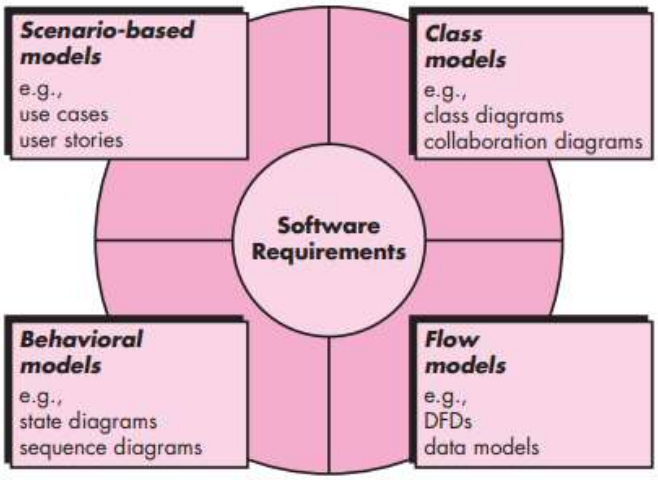
The job of the tool smith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs.

The role of the domain analyst is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

5. Explain various approaches in Requirements Modeling.

One view of requirements modelling, called structured analysis, considers data and the processes that transform the data as separate entities. Data objects are modelled in a way that defines their attributes and relationships. Processes that manipulate data objects are modelled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modelling, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.



Each element of the requirements model (Figure above) presents the problem from a different point of view.

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

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) 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.

6. Comment on Creating a Preliminary Use Case

A use case captures the interactions that occur between producers and consumers of information and the system itself. A few major questions to be addressed:

(1) what to write about

(2) how much to write about it

(3) how detailed to make your description

(4) how to organize the description?

The first two requirements engineering tasks—inception and elicitation—provide with the information 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.

Take the example of **SafeHome home surveillance** function (subsystem) identifies the following functions (an abbreviated list) that are performed by the homeowner actor:

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

Consider the function access camera surveillance via the Internet— display camera views (ACS-DCV). The stakeholder who takes on the role of the homeowner actor might write the following narrative:

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

If he is at a remote location, he can use any PC with appropriate browser software to log on to the SafeHome Products website.

He enter his user ID and two levels of passwords and once he is validated, he has access to all functionality for my installed SafeHome system.

To access a specific camera view, he selects “surveillance” from the major function buttons displayed.

He then selects “pick a camera” and the floor plan of the house is displayed.

He then selects the camera that he is interested in.

Alternatively, he can look at thumbnail snapshots from all cameras simultaneously by selecting “all cameras” as his viewing choice.

Once he chooses a camera, he selects “view” and a one-frame-per-second view appears in a viewing window that is identified by the camera ID.

If he wants to switch cameras, he selects “pick a camera” and the original viewing window disappears and the floor plan of the house is displayed again.

He then selects the camera that he is interested in.

A new viewing window appears.

A variation of a narrative use case presents the interaction as an ordered sequence of user actions.

Each action is represented as a declarative sentence.

Revisiting the ACS-DCV function, you would write:

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

Actor: homeowner

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

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

7. Explain how a Preliminary Use Case can be refined.

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?*

The answer is "yes." Referring to the free-flowing narrative, the actor may choose to view thumbnail snapshots of all cameras simultaneously. Hence, one secondary scenario might be "View thumbnail snapshots for all cameras."

• *Is it possible that the actor will encounter some error condition at this point? If so, what might it be?*

Again the answer to the question is "yes." A floor plan with camera icons may have never been configured. Hence, selecting "pick a camera" results in an error condition: "No floor plan configured for this house."9 This error condition becomes a secondary scenario.

• *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)? If so, what might it be?*

Again the answer to the question is "yes." This would result in the system displaying a special alarm notification (type, location, system action) and providing the actor with a number of options relevant to the nature of the alarm. Because this secondary scenario can occur at any time for virtually all interactions, it will not become part of the ACS-DCV use case. Rather, a separate use case—Alarm condition encountered—would be developed and referenced from other use cases as required.

• *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.

• *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.

• *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

8. Describe how to write a Formal Use Case

The ACS-DCV use case specified in the previous answer follows a typical outline for formal use cases.

The goal in context identifies the overall scope of the use case.

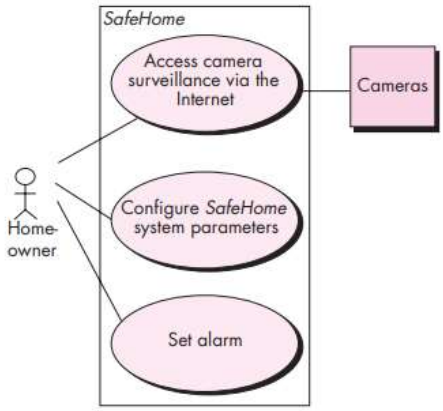
The precondition describes what is known to be true before the use case is initiated.

The trigger identifies the event or condition that "gets the use case started".

The scenario lists the specific actions that are required by the actor and the appropriate system responses.

Exceptions identify the situations uncovered as the preliminary use case is refined

Additional headings may or may not be included and are reasonably self-explanatory.



Every modeling notation has limitations, and the use case is no exception.

Like any other form of written description, a use case is only as good as its author(s).

If the description is unclear, the use case can be misleading or ambiguous.

A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements. For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient. However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer.

If developed properly, the use case can provide substantial benefit as a modeling tool.

9. Describe the way of building requirements model.

or

10. Explain the elements of the requirements model.

As the requirements model evolves, certain elements will become relatively stable, providing a solid foundation for the design tasks that follow.

However, other elements of the model may be more volatile, indicating that stakeholders do not yet fully understand requirements for the system.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. Figure below depicts it and evolve into more elaborate template-based use cases.

Scenario-based elements of the requirements model are often the first part of the model that is developed.

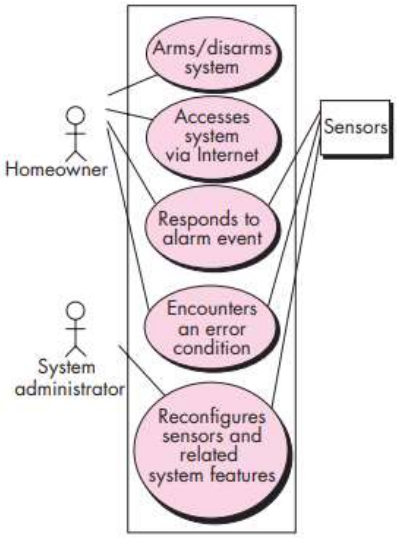
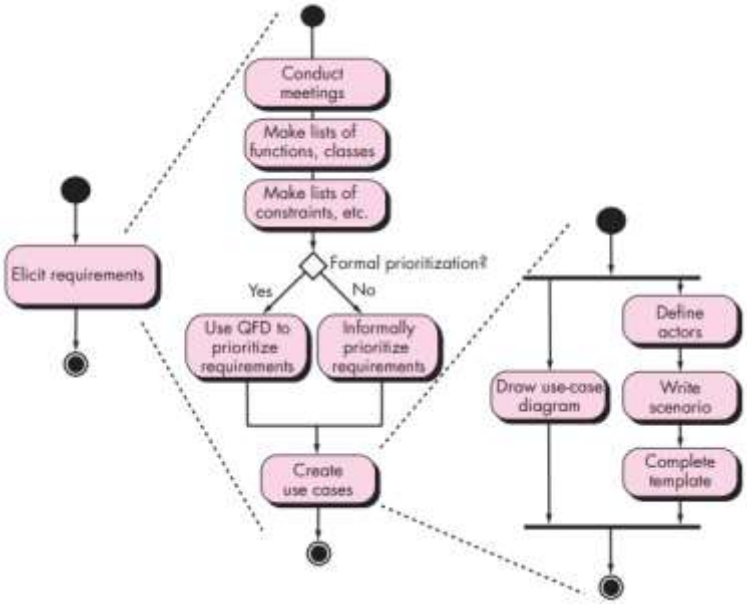


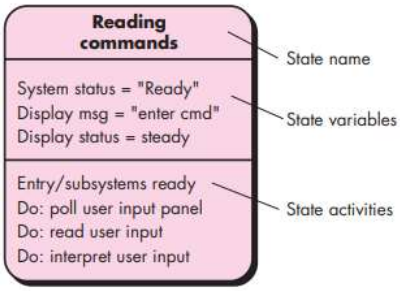
Figure below depicts a UML activity diagram for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.



Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.

Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., identify, enable) that can be applied to modify these attributes as shown in the below diagram.

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



The state diagram is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state.

A state is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.

A simplified UML state diagram is shown in above Figure.

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.

Output may light a single LED or produce a 200-page report.

11. Comment on requirements negotiation in software engineering practice.

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.

Unfortunately, this rarely happens. In reality, 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.

Rather than a single customer communication activity, the following activities are defined:
1. Identification of the system or subsystem's key stakeholders.

2. Determination of the stakeholders' "win conditions."

3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

11. Write a note on Validating Requirements

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

Few questions need to be addressed.

• Is each requirement consistent with the overall objectives for the system/product?

• Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

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

• Is each requirement bounded and unambiguous?

• Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

• Do any requirements conflict with other requirements?

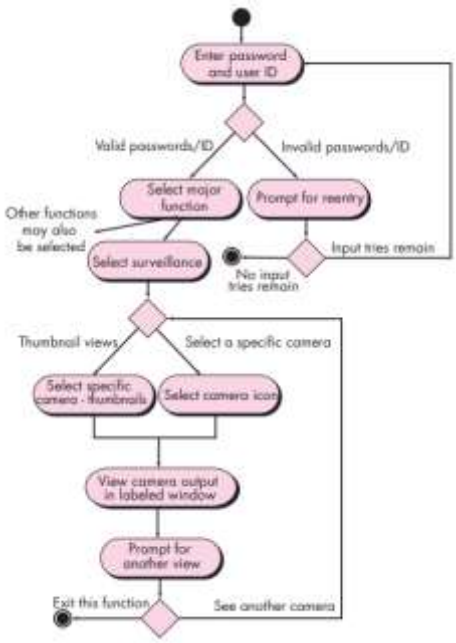
• Is each requirement achievable in the technical environment that will house the system or product? • Is each requirement testable, once implemented?

• Does the requirements model properly reflect the information, function, and behavior of the system to be built?

• Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

• Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

12. Describe how an activity diagram is generated with an example.



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 (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring.

An activity diagram for the ACS-DCV use case is shown in above Figure.

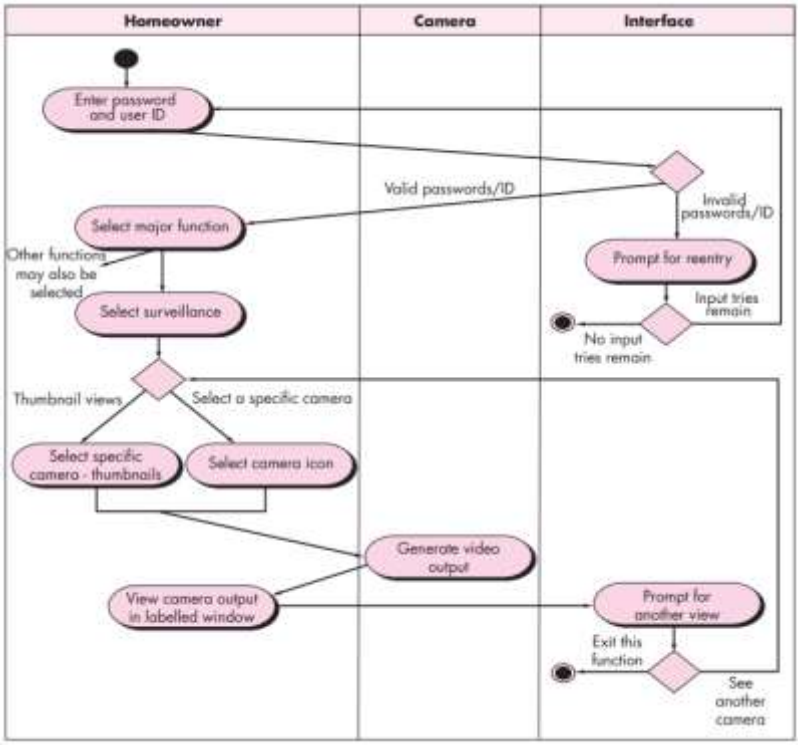
For example, a user may only attempt to enter userID and password a limited number of times. This is represented by a decision diamond below "Prompt for reentry."

13. Explain the concept of swimlane diagram with an example

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 (if there are multiple actors involved in a specific use case) or analysis has responsibility for the action described by an activity rectangle.

Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

Three analysis classes—Homeowner, Camera, and Interface—have direct or indirect responsibilities in the context of the activity diagram represented in Figure.



14. Explain the data modelling concept considering various factors in it.

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 choose to create a data model as part of overall requirements modeling.

A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships.

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

Data Objects

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

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).

For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes.

The description of the data object incorporates the data object and all of its attributes.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table.

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.

Referring to the data object car, a reasonable identifier might be the ID number.

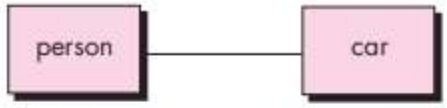
The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context.

The attributes for car might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software.

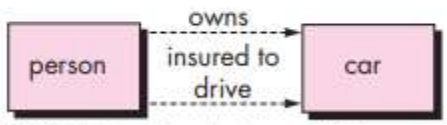
In the latter case, the attributes for car might also include ID number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

Relationships

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation



(a) A basic connection between data objects



(b) Relationships between data objects

illustrated in Figure a.

A connection is established between person and car because the two objects are related.

But what are the relationships? To determine the answer, we should understand the role of people (owners, in this case) and cars within the context of the software to be built.

We 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.

The relationships owns and insured to drive define the relevant connections between person and car.

Figure b illustrates these object-relationship pairs graphically.

The arrows noted in Figure b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

15. Explain class-based modelling covering various factors of it.

16. Narrate how analysis classes manifest themselves with supporting points.

Class-based modelling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined.

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

Identifying Analysis Classes

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

Classes are determined by underlining each noun or noun phrase and entering it into a simple table. Synonyms should be noted.

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 (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.

• Things (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.

• Occurrences or events (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.

• Roles (e.g., manager, engineer, salesperson) played by people who interact with the system.

• Organizational units (e.g., division, group, team) that are relevant to an application.

• Places (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.

• Structures (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

17. Describe how analysis classes can be defined during modelling.

To illustrate how analysis classes might be defined during the early stages of modeling, consider a grammatical parse for a processing narrative for the SafeHome security function.

Extracting the nouns, we can propose a number of potential classes:

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

18. What are the six selection characteristics to be used for including classes in analysis model?

Coad and Yourdon suggest six selection characteristics that should be used as you consider each potential class for inclusion in the analysis model:

1. Retained information. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.

2. Needed services. The potential class must have a set of identifiable operations that can change the value of its attributes in some way.

3. Multiple attributes. During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.

4. Common attributes. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.

5. Common operations. A set of operations can be defined for the potential class and these operations apply to all instances of the class.

6. Essential requirements. External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

19. Comment on Specifying attributes regarding analysis model

Specifying Attributes

Attributes describe a class that has been selected for inclusion in the requirements model.

In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

For a cricketer, attributes such as name, position, batting average, fielding percentage, years played, and games played might be relevant.

But for the same, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.

In addition, the following question should be answered for each class: “What data items (composite and/or elementary) fully define this class in the context of the problem at hand?”

To illustrate, we consider the System class defined for SafeHome.

A homeowner can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth.

We can represent these composite data items in the following manner: 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

In general, we avoid defining an item as an attribute if more than one of the items is to be associated with the class.

20. Explain about defining operations in class (analysis) modelling.

Defining Operations

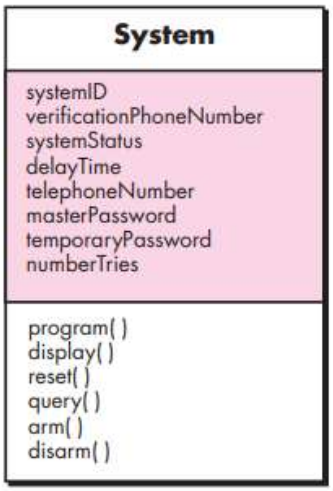
Operations define the behaviour of an object.

Although many different types of operations exist, they can generally be divided into four broad categories:

(1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting),

(2) operations that perform a computation,

(3) operations that inquire about the state of an object,



(4) operations that monitor an object for the occurrence of a controlling event.

Therefore, an operation must have "knowledge" of the nature of the class' attributes and associations.

For example, from the SafeHome processing narrative presented earlier in this content, we see that "sensor is assigned a number and type" or "a master password is programmed for arming and disarming the system."

These phrases indicate a number of things:

• That an assign() operation is relevant for the Sensor class.

• That a program() operation will be applied to the System class.

• That arm() and disarm() are operations that apply to System class.

Upon further investigation, it is likely that the operation program() will be divided into a number of more specific suboperations required to configure the system.

program() implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s).

But for now, we specify program() as a single operation.

21. Explain CRC modelling with relevant sketches

22. Explain class modelling in CRC with an example

23. Explain Collaborator modelling in CRC with an example

24. Explain Responsibility modelling in CRC with an example

Class-Responsibility-Collaborator (CRC) Modelling

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

A CRC model is really a collection of standard index cards that represent classes.

The cards are divided into three sections.

Along the top of the card you write the name of the class.

In the body of the card you list the class responsibilities on the left and the collaborators on the right. In reality, the CRC model may make use of actual or virtual index cards.

The intent is to develop an organized representation of classes.

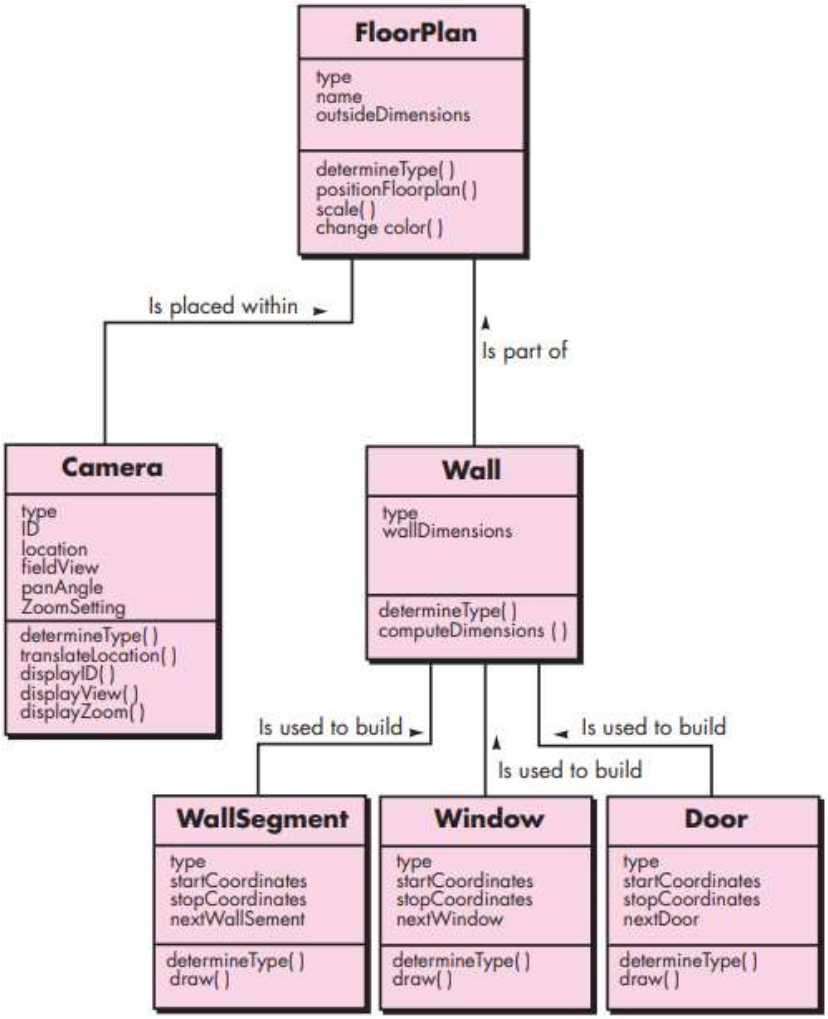
Responsibilities are the attributes and operations that are relevant for the class.

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 shown below.

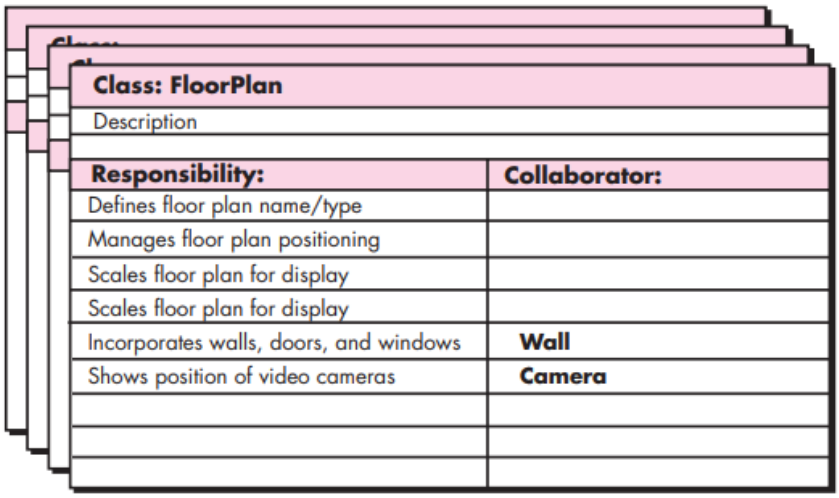
The list of responsibilities shown on the CRC card is preliminary and subject to additions or modification. The classes Wall and Camera are noted next to the responsibility that will require their collaboration.



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

• Entity classes, also called model or business classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor). These classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

• 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.



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**

Entity objects contain information that is important to users, but they do not display themselves. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

For example, a boundary class called CameraWindow would have the responsibility of displaying surveillance camera output for the SafeHome system.

• Controller classes manage a "unit of work" [UML03] 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.

Five guidelines for allocating responsibilities to classes:

1. *System intelligence should be distributed across classes to best address the needs of the problem.*

Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. "Dumb" classes (those that have few responsibilities) can be modelled to act as servants to a few "smart" classes (those having many responsibilities).

Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: it concentrates all intelligence within a few classes, making changes more difficult, and it tends to require more classes, hence more development effort. If system intelligence is more evenly distributed across the classes in an application, each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved.

This enhances the maintainability of the software and reduces the impact of side effects due to change. 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 extraordinarily long list of responsibilities.

This indicates a concentration of intelligence.18 In addition, the responsibilities for each class should exhibit the same level of abstraction.

For example, among the operations listed for an aggregate class called CheckingAccount a reviewer notes two responsibilities: balance-the-account and check-off-clearedchecks. The first operation (responsibility) implies a complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, check-off-clearedchecks should be placed within the responsibilities of CheckEntry, a class that is encompassed by the aggregate class CheckingAccount.

2. *Each responsibility should be stated as generally as possible*.

This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses).

3. *Information and the behavior related to it should reside within the same class*.

This achieves the object-oriented principle called encapsulation. Data and the processes that manipulate the data should be packaged as a cohesive unit.

4. *Information about one thing should be localized with a single class, not distributed across multiple classes.*

A single class should take on the responsibility for storing and manipulating a specific type of information. 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. As an example, consider a video game that must display the following classes: Player, PlayerBody, PlayerArms, PlayerLegs, PlayerHead.

Each of these classes has its own attributes (e.g., position, orientation, color, speed) and all must be updated and displayed as the user manipulates a joystick.

The responsibilities update() and display() must therefore be shared by each of the objects noted. Player knows when something has changed and update() is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

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

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

As an example, consider the SafeHome security function.

As part of the activation procedure, the ControlPanel object must determine whether any sensors are open. A responsibility named determine-sensor-status() is defined. If sensors are open, ControlPanel must set a status attribute to "not ready." Sensor information can be acquired from each Sensor object. Therefore, the responsibility determine sensor-status() can be fulfilled only if ControlPanel works in collaboration with Sensor.

To help in the identification of collaborators, one can examine three different generic relationships between classes:

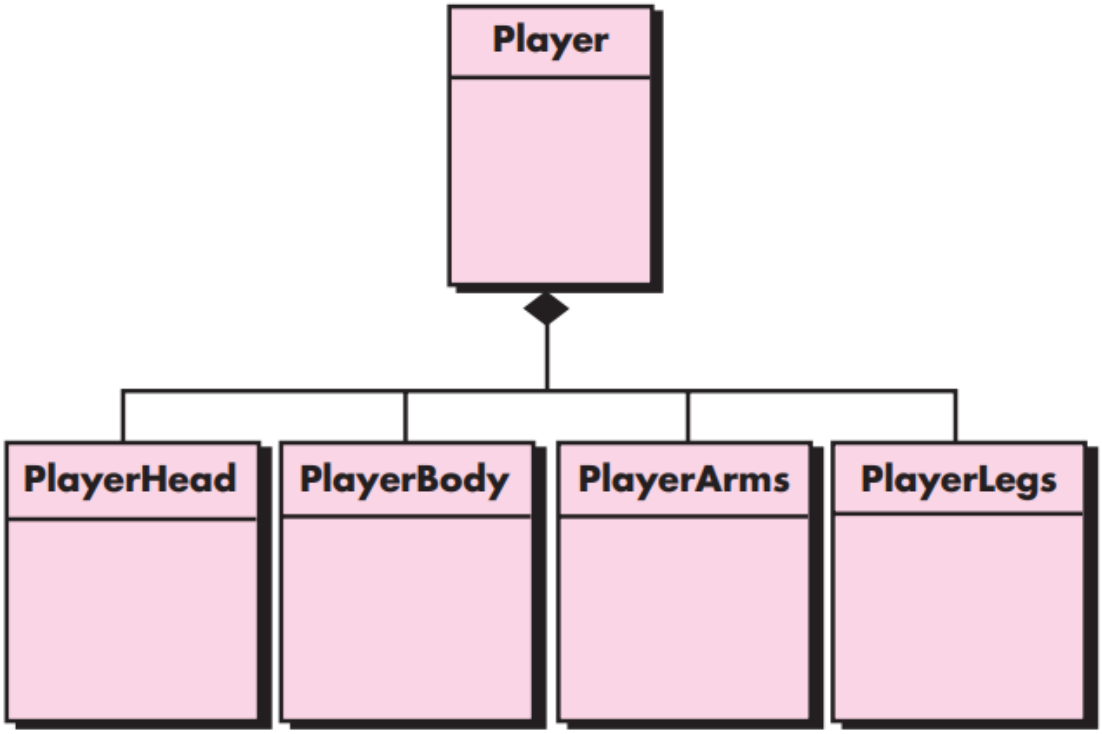
(1) the is-part-of relationship,

(2) the has-knowledge-of relationship, and

(3) the depends-upon relationship.

Consider the classes defined for the video game noted earlier, the class PlayerBody is-part-of Player, as are PlayerArms, PlayerLegs, and PlayerHead.

In UML, these relationships are represented as the aggregation shown in Figure below



When one class must acquire information from another class, the has-knowledge of relationship is established.

The determine-sensor-status() responsibility noted earlier is an example of a has-knowledge-of relationship.

The depends-upon relationship implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of.

For example, PlayerHead must always be connected to PlayerBody (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the PlayerHead object called center-position is determined from the center position of PlayerBody. This information is obtained via a third object, Player, that acquires it from PlayerBody. Hence, PlayerHead depends-upon PlayerBody.

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration.

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

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

2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.

3. The review leader reads the use case deliberately.

4. When the token is passed, the holder of the Sensor card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.

5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards

25. Explain the concept of Associations and Dependencies with an example

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 previously taken example, the FloorPlan class is defined by identifying a set of associations between FloorPlan and two other classes, Camera and Wall.

The class Wall is associated with three classes that allow a wall to be constructed, WallSegment, Window, and Door.

Objects like Wall is constructed from one or more WallSegment objects.

In addition, the Wall object may contain 0 or more Window objects and 0 or more Door objects. These multiplicity constraints are illustrated in figure (a) shown below, where “one or more” is represented using 1. .*, and “0 or more” by 0 . .*. In UML, the asterisk indicates an unlimited upper bound on the range.

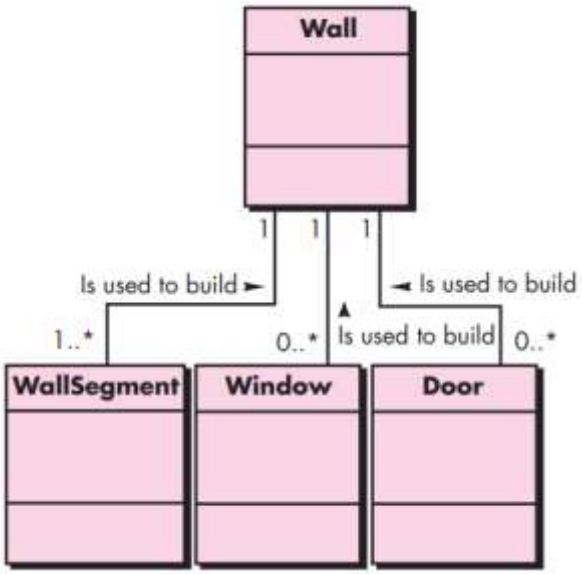


Figure a

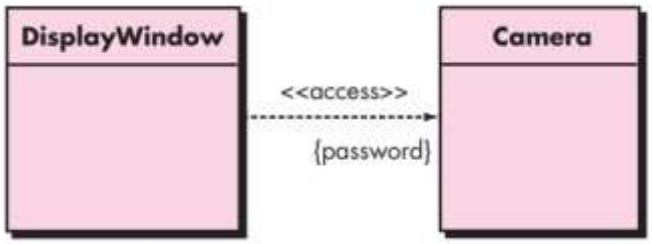


Figure b

In many instances, a client-server relationship exists between two analysis classes.

In such cases, a client class depends on the server class in some way and a dependency relationship is established. Dependencies are defined by a stereotype.

A stereotype is an "extensibility mechanism" within UML that allows you to define a special modeling element whose semantics are custom defined.

In UML stereotypes are represented in double angle brackets (e.g., <>).

As an illustration of a simple dependency within the SafeHome surveillance system, a Camera object (in this case, the server class) provides a video image to a DisplayWindow object (in this case, the client class). The relationship between these two objects is not a simple association, yet a dependency association does exist.

In a use case written for surveillance (not shown), we learn that a special password must be provided in order to view specific camera locations.

One way to achieve this is to have Camera request a password and then grant permission to the DisplayWindow to produce the video display. T

his can be represented as shown in Figure (b) shown above, where <> implies that the use of the camera output is controlled by a special password.

25. Describe Analysis Packages with the help of an example

Analysis Packages

An important part of analysis modeling is categorization.

That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an analysis package—that is given a representative name.

To illustrate the use of analysis packages, consider the video game that is introduced earlier.

As the analysis model for the video game is developed, a large number of classes are derived.

Some focus on the game environment—the visual scenes that the user sees as the game is played. Classes such as Tree, Landscape, Road, Wall, Bridge, Building, and VisualEffect might fall within this category.

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

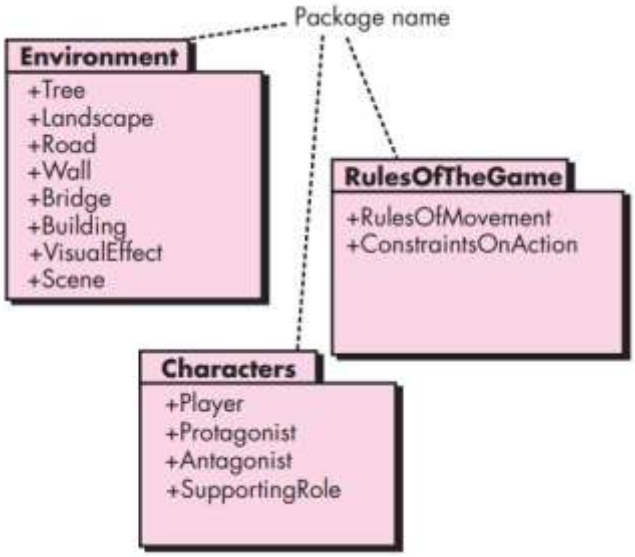
Classes such as Player (described earlier), Protagonist, Antagonist, and SupportingRoles might be defined.

Classes such as RulesOfMovement and ConstraintsOnAction are candidates here.

These classes can be grouped in analysis packages as shown in Figure shown below.

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.


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