OBJECT MODELLING USING UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed.

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system. The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

UML Diagrams

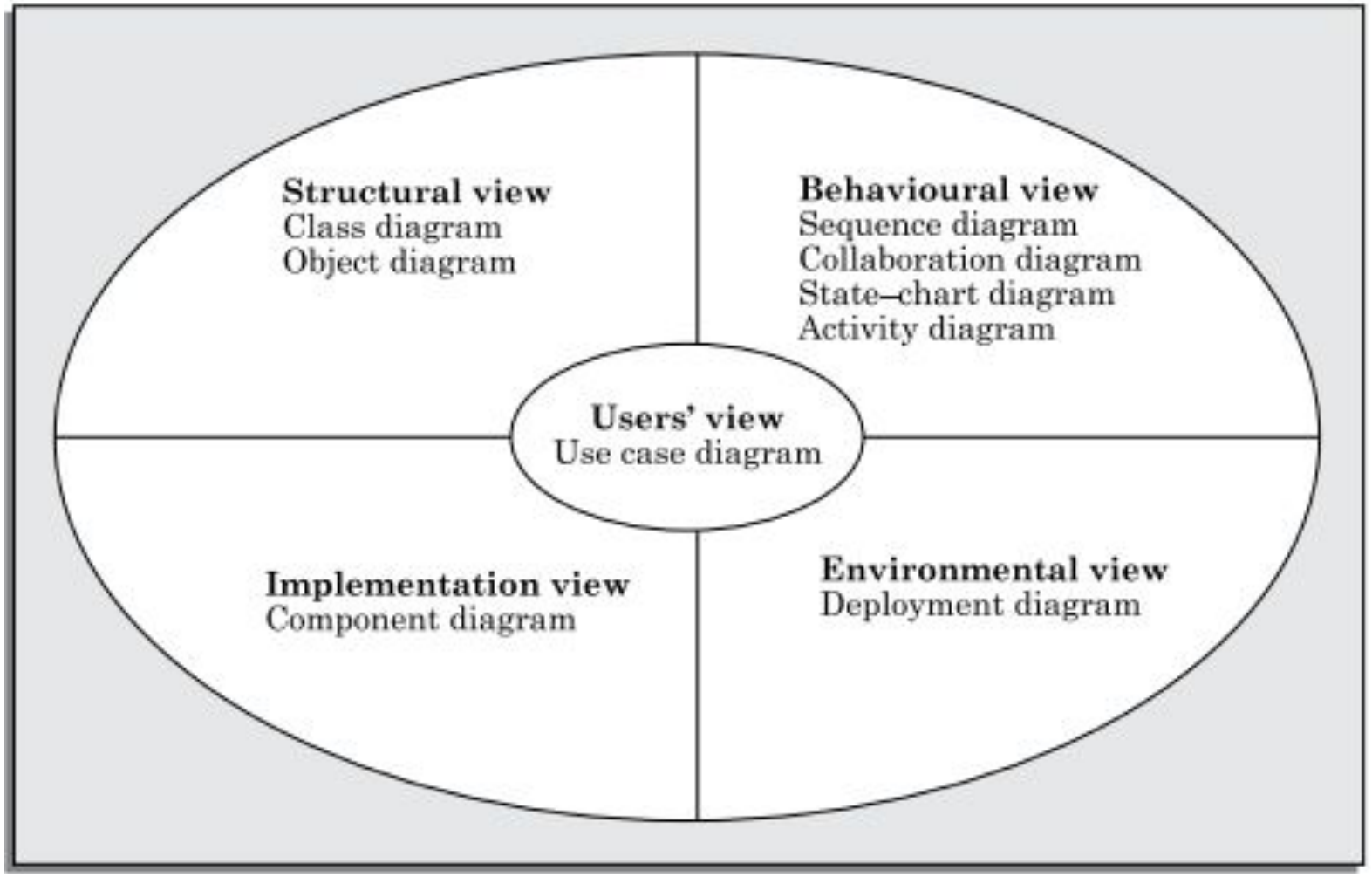


FIGURE: Different types of diagrams and views supported in UML.

UML Diagrams

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

UML Diagrams

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?”

Example: For the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use Case Model

- Use cases correspond to the high-level functional requirements.
- The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view.
- To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.
- The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.
- The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software.
- A use case typically involves a sequence of interactions between the user and the system.

Use Case Model

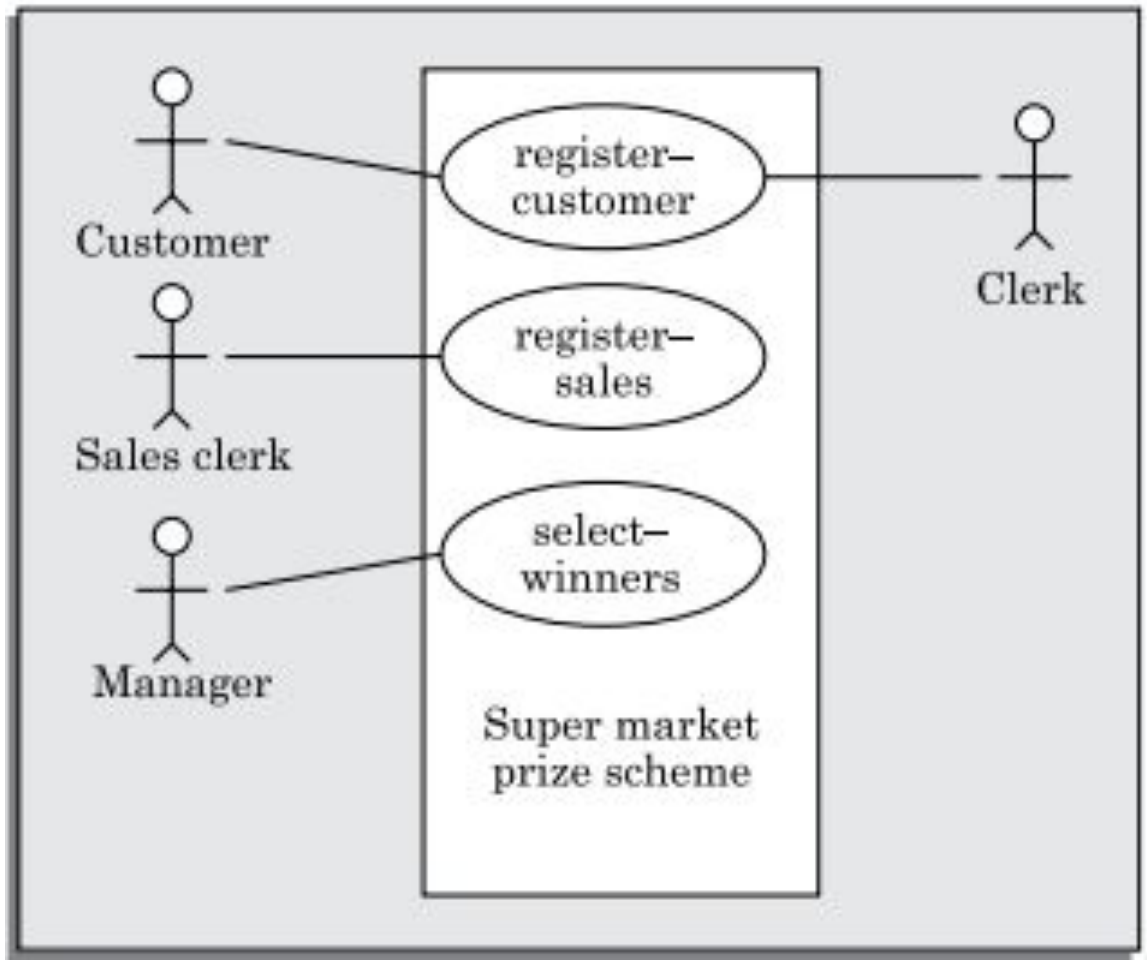


Figure: The use case diagram of the Super market prize scheme

Use Case Model

The Main Line Sequence:

- The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction.
- For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount.
- Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

Use Case Model

Representation of Use Cases:

- Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing.
- In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse.
- All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary.
- The name of the system being modeled (such as Library Information System) appears inside the rectangle.

Use Case Model

Representation of Use Cases:

- The different users of the system are represented by using the stick person icon.
- Each stick person icon is normally referred to as an actor.
- An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases.
- The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case.
- Both the human users and the external systems can be represented by stick person icons.
- When a stick person icon represents an external system, it is annotated by the stereotype <>.

Use Case Model

Text Description:

Each ellipse on the use case diagram should be accompanied by a text description to define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc.

The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Use Case Model

Text Description:

The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Use Case Model

Text Description:

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation

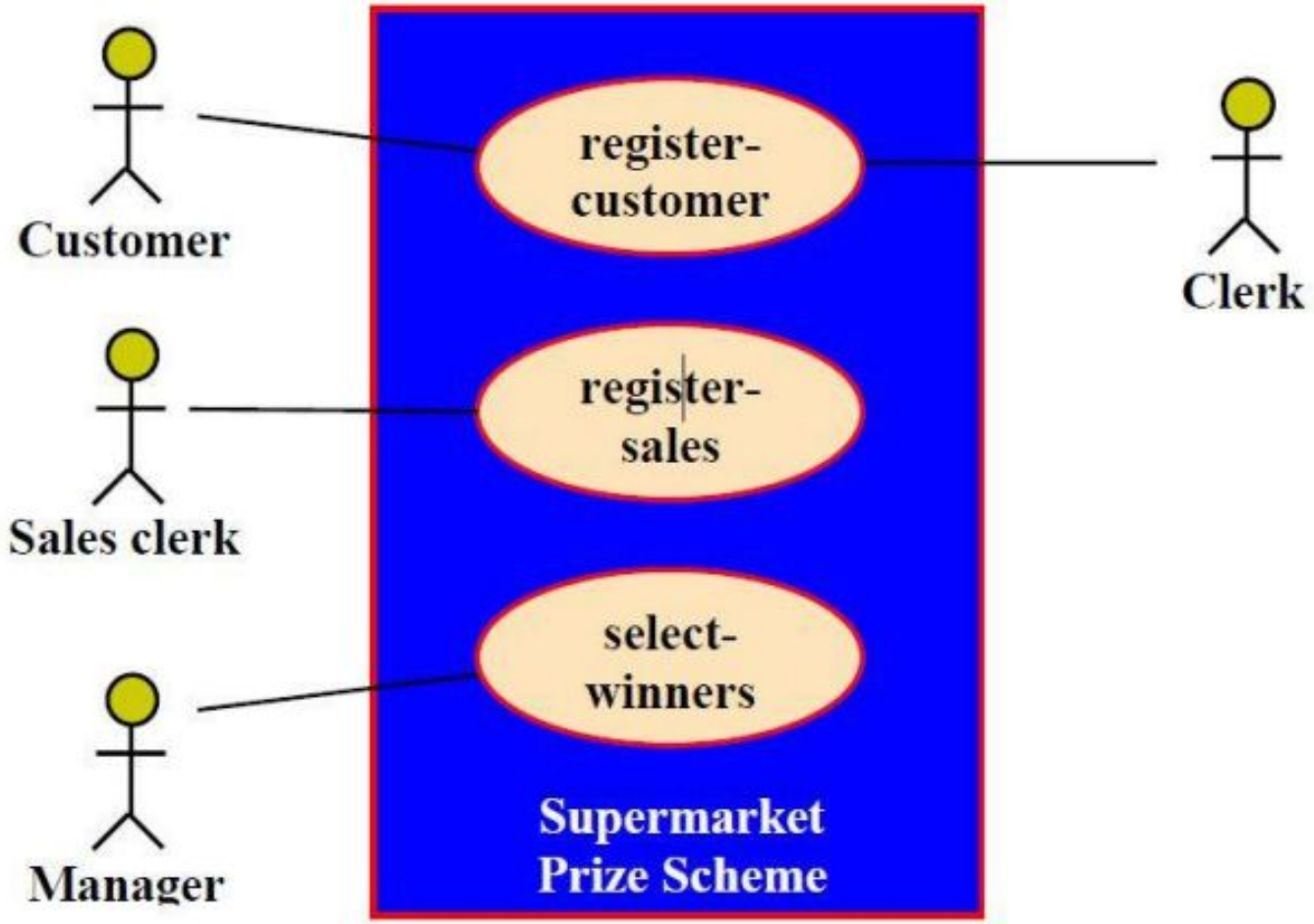
Use Case Model

Example: A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the checkout staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Tk.100,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

Use Case Model

Example:

The use case model for the Supermarket Prize Scheme is shown in the figure. We can identify three use cases: “register-customer”, “register-sales”, and “select-winners”.



Use Case Model

Text description:

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

1. Customer: select register customer option.
2. System: display prompt to enter name, address, and telephone number.
3. Customer: enter the necessary values.
4. System: display the generated id and the message that the customer has been successfully registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

1. System: displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

The description for other use cases is written in a similar fashion.

How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the high-level requirements specified in the SRS document correspond to the use cases. In the absence of a well-formulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the different functions that they might initiate or participate are identified.

Example: For a Library Automation System, the categories of users can be members, librarian, and the accountant. Each user typically focuses on a set of functionalities. For example, the member typically concerns himself with book issue, return, and renewal aspects. The librarian concerns himself with creation and deletion of the member and book records. The accountant concerns itself with the amount collected from membership fees and the expenses aspects.

Factoring of Use Cases

It is often desirable to factor use cases into component use cases.

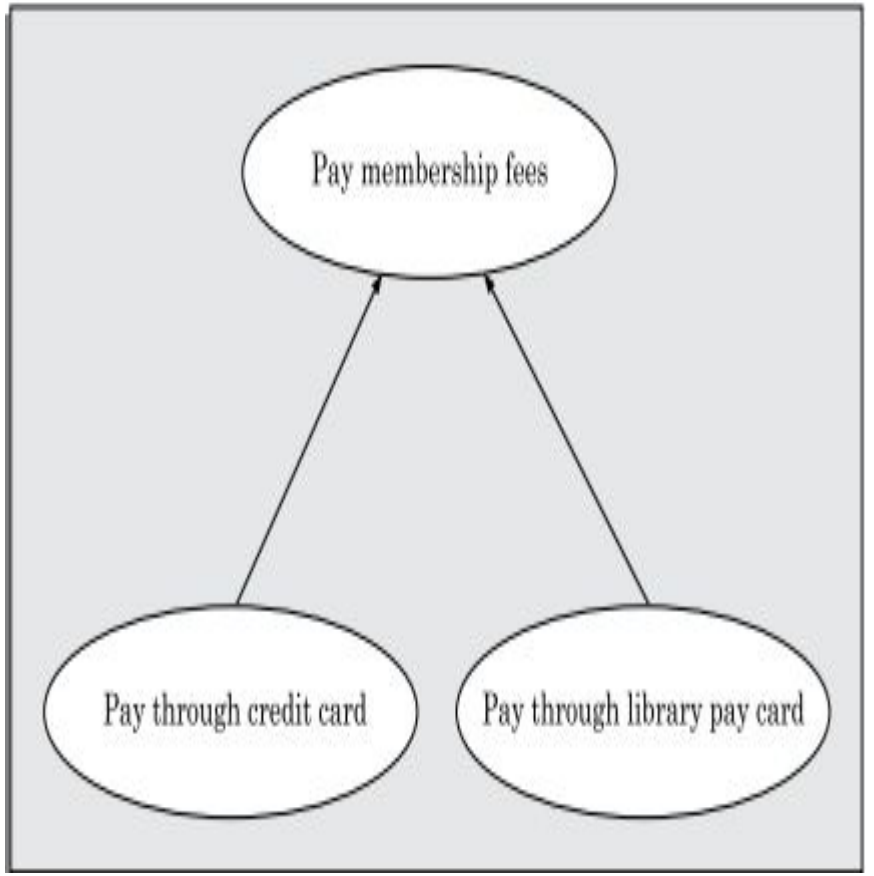
Actually, factoring of use cases are required under two situations-

Decompose complex use cases: Complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper.

Identify commonality: Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant.

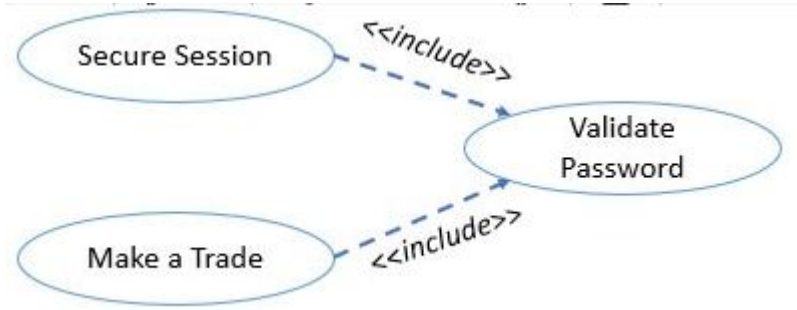
Factoring of Use Cases

Generalization: Use case generalization can be used when one use case is quite similar to another, but does something slightly different or something more. That is one use case is a special case of another use case. In this sense, generalization works the same way with use cases as it does with classes. The child use case inherits the behavior of the parent use case. The notation is the same too (See Figure). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.



Factoring of Use Cases

Includes: The includes relationship implies one use case includes the behavior of another use case in its sequence of events and actions. The includes relationship is appropriate when you have a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use case into more manageable parts. The includes relationship is represented using a predefined stereotype ≪include≫. In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use case.



Factoring of Use Cases

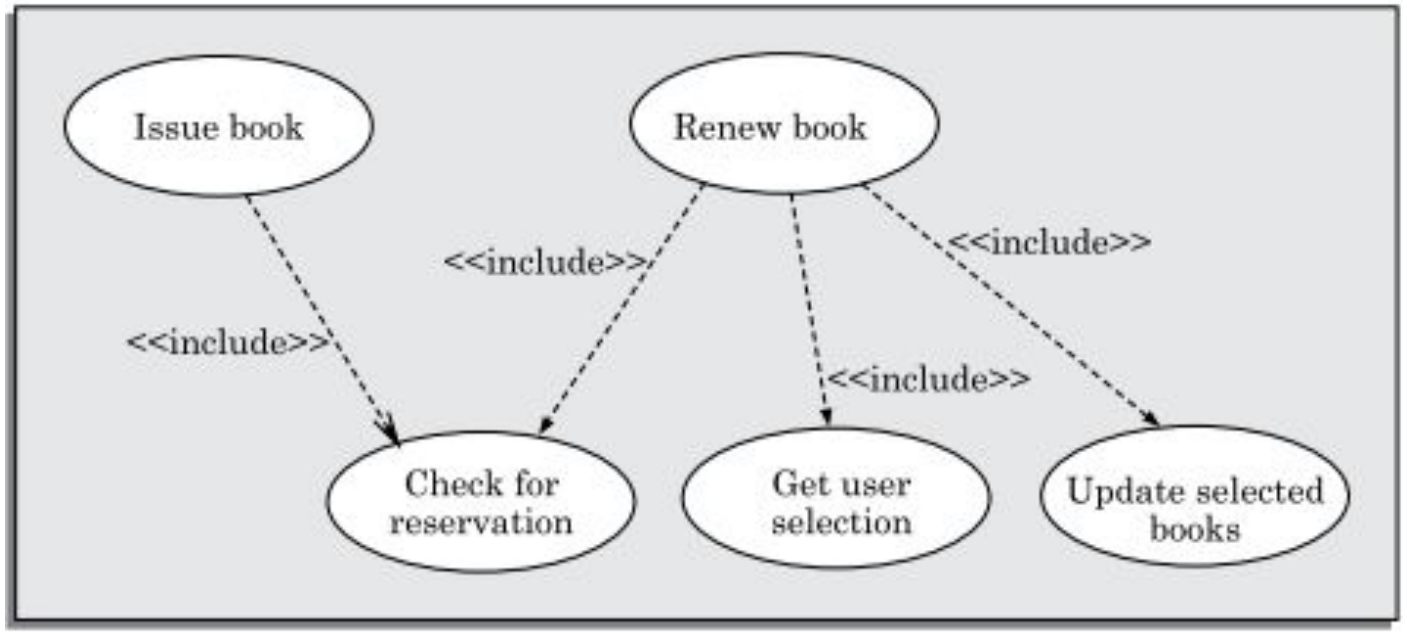
Includes Example:

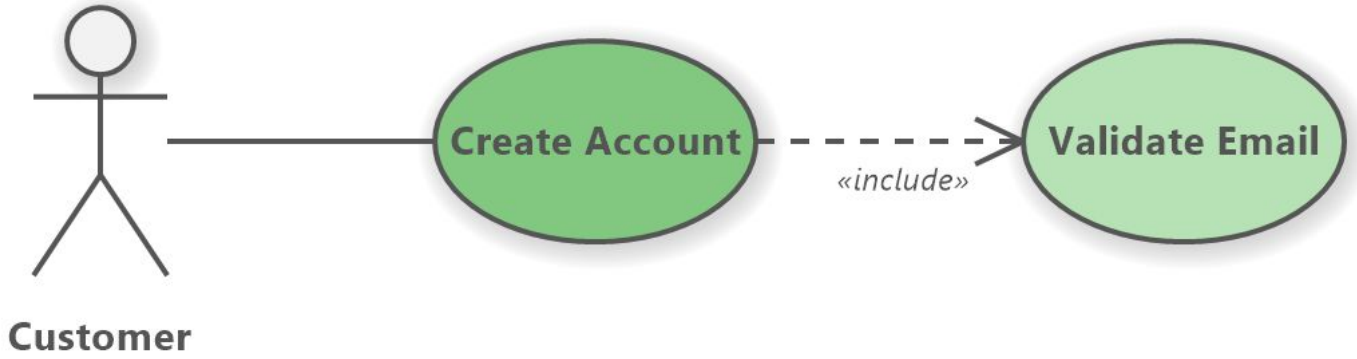
1) The use cases issue-book and renew-book both include check-reservation use case. The implication is that during execution of either of the use cases issue-book and renew-book, the use case check-reservation is executed. The base use case may include several use cases.

2) We have a use case called "Create Account" which represents the process of creating a new account on a website. This use case might include the use case "Validate Email," which represents the process of verifying that the email address entered by the user is valid. In this case, the "Validate Email" use case would be included in the "Create Account" use case.

Factoring of Use Cases

Includes Example:





Factoring of Use Cases

Extends: The extends relationship among use cases is that it allows one to show optional behavior that may occur during the execution of the use case. An optional system behavior is executed only if certain conditions hold, otherwise the optional behavior is not executed.

The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied.

Factoring of Use Cases

Extends Example:

- We have a use case called "Place Order" which represents the process of placing an order on an online store. This use case might be extended by the use case "Apply Discount," which represents the process of applying a discount to the order. In this case, the "Apply Discount" use case would extend the "Place Order" use case.
- We have also a use case called "Update Delivery Address" which represents the process of updating the delivery address for an existing order. This use case can be extended by the "Place Order" use case, as the customer may want to update their delivery address while placing a new order.

Factoring of Use Cases

Extends Example:

However, "Update Delivery Address" can also be used directly by the customer actor, as they may need to update their delivery address at any time before or after placing an order. In this case, the "Update Delivery Address" use case extends the "Place Order" use case, but can also be used independently by the customer actor.