Tutorial - Building a Use Case Diagram

1. Introduction

A **Use Case diagram** is a graphical representation of the high-level system scope. It includes **use cases**, which are pieces of functionality the system will provide, and **actors**, who are the users of the system. Looking at a Use Case diagram, you should easily be able to tell what the system will do and who will interact with it.

You can create one or more Use Case diagrams for a single system. If you create more than one, each will show a subset of the actors and/or use cases in the system. You can also group the use cases and actors into **packages** to help organise the model.

A Use Case diagram can be helpful in communicating with the end users of the system. It is designed to be straightforward and non-technical so that everyone on the team can come to a common understanding of the system scope. It is usually created by the technical team, but in conjunction with an end user representative.

To create a new Use Case diagram:

1. Right-click a package in the Use Case view.
2. Select New → Use Case Diagram.

A Use Case diagram shows a subset of the use cases and actors in the system. You can create as many Use Case diagrams as you need to fully document the system scope.

2. Actors

An actor is a person, system, piece of hardware, or other thing that interacts with your system. It is denoted with a stick figure. We differentiate **active** actors, who initiate interactions with a system, and **passive** actors, who are targets of requests or who are activated by the system.

Follow these steps to add actors to the diagram:

1. Determine the actors for your system.
2. Select the Actor toolbar button.
3. Click in the Use Case diagram to add the actor.
4. Name the actor.

3. Use cases

A use case is a piece of functionality the system will provide. It is usually named in the format <verb><noun>, such as "Deposit Check" or "Withdraw Cash." Use cases are high-level and implementation-independent. It is denoted with a horizontal ellipse:

Follow these steps to add use cases to the diagram:

1. Determine the use cases of the diagram.
2. Select the Use Case toolbar button.
3. Click in the Use Case diagram to add the use case.
4. Give the use case a name.

4. Relationships

4.1 Communicates relationship

A **communicates** relationship between an actor and a use case indicates that the actor initiates the use case. An actor may initiate one or more use cases.

To add relationships between the actors and use cases:

1. Select the Unidirectional Association toolbar button.
2. Drag an arrow from the actor to the use case.

Example: Consider the following requirements of a flight booking and hotel reservation system. A customer can purchase a ticket for a flight without having to book a hotel. They also have the opportunity to reserve a hotel room or change a reservation without

having to book a flight. Finally, they can hire a rental car for their holiday. Create a simple Use Case diagram for the above problem.

4.2 Includes relationship

An **includes** relationship suggests that one use case must include another. In other words, running one use case means that the other must be run as well. One use case may be included by one or more other use cases. It is denoted by a dashed unidirectional arrow from the source use case(s) to the target use case.

To add includes relationships between appropriate use cases:

1. Select the Dependency or instantiates toolbar button.
2. Drag an arrow from a use case to the use case it includes.
3. Double-click the arrow.
4. Assign the stereotype as include.

4.3 Extends relationship

An **extends** relationship is used when one use case optionally extends the functionality provided by another. In other words, if one use case runs, an extending use case may or may not run.

To add extends relationships between appropriate use cases:

1. Select the Dependency or instantiates toolbar button.
2. Drag an arrow from the extending use case to the use case it extends.
3. Double-click the arrow.
4. Assign the stereotype as extend.

Example: Consider the following flight check-in subsystem requirements. In order to successfully check-in passengers, the attendant must weigh the passengers' luggage and assign them a seat. There are two ways to assign a seat: assigning a window seat and assigning an aisle seat, but only one need be completed in the process of assigning the passenger a seat. Create a simple Use Case diagram for the above problem.

4.4 Generalisation relationship

A **generalisation** relationship between actors or use cases indicates that one actor or use case (the child) inherits the characteristics of another actor or use case (the parent). The child actor may initiate all of the use cases that the parent can initiate.

To add generalization relationships between appropriate actors:

1. Select the Generalization toolbar button.
2. Drag an arrow from the child actor/use case to the parent actor/use case.

Example: Consider the following credit granting system for a bank where a customer can apply for different types of credit. The types of customer are:

- Home Owner
- Home Buyer
- Auto Buyer
- Credit Card Applicant

If the customer is a home owner then a loan officer must process the customer's home equity loan application but if the customer is a home buyer then a loan officer must process the customer's home mortgage loan application. In both cases an assessor (independent of the bank) is required to assess the home value. If the customer is an auto buyer then a loan officer must process the customer's auto loan application. Finally, if the customer is a credit card applicant, a loan officer must process the credit card application. For all types of customers (except for home owners) it is required that a credit history check is made on the customer. This information can be obtained by the local credit bureau. Create a Use Case diagram for the above problem.

Building a Class Diagram

A Class diagram is used to show a subset of the classes, interfaces, packages of classes, and relationships in the system. A typical system will have many different Class diagrams.

In Rose, different icons are used to represent different kinds of classes on a Class diagram. For example, Rose contains icons for interfaces, client pages, session EJBs, COM objects, and many other types of classes. Rose also contains icons that distinguish analysis classes from design classes. An *analysis class* is an implementation-independent view of the system, intended to be an initial sketch of the system design. *Design classes* are implementation-specific and correlate to the classes that will eventually be created in the source code.

Rose can generate code that will include the class name, attribute types, default values, operation signatures, and class relationships. Developers use the Class diagrams to see the system structure and to know what operations to create for a given class.

A *Class diagram* includes a subset of the classes, attributes, operations, relationships, and packages of classes in the system. You can create as many Class diagrams as you need to fully document the system design.

Follow these steps to create a new Class diagram:

1. Click the + (plus sign) next to the Logical View entry in the browser to open it.
2. The Main Class diagram will be visible. Double-click the Main Class diagram to open it.

To add a Class to the diagram:

1. Select the Class button from the toolbar. The cursor changes to a plus sign (+) when moved to the diagram.
2. Click anywhere inside the Class diagram. The new class will be named NewClass by default.
3. To create a new class, replace the word NewClass with the new class name. Note that the new class has also been automatically added to the browser in the Logical view.

An *attribute* is a piece of information associated with a class. All objects in a given class will share the same attributes, but each object may have its own attribute values.

Follow these steps to add attributes to the classes:

1. Right-click a class on the diagram.
2. Select New → Attribute.
3. Type the attribute name, followed by a colon, and then the attribute's data type (i.e., **Address:Strin**g). Optionally enter a default value for the attribute, by following the data type with an equals sign and then the default value (i.e., **Address:String = 123 Main S**t.).
4. Right-click the attribute in the browser window and select Open Specification.
5. Set the attribute visibility (public, private, protected).

An *operation* is a method within the class. In Rose, you can define the operation name, parameters, visibility, return type, and parameter data types. Certain operations, such as Get() and Set() methods for attributes, can be automatically generated by Rose.

Follow these steps to add operations to the classes:

1. Right-click a class on the diagram.
2. Select New → Operation.
3. Enter the operation signature, including parameters and a return type. Use the format
4. OpName(Parm1:Parm1DataType, Parm2:Parm2DataType):ReturnType (for example,

5. **AddNumbers(X:Int, Y:Int): Lon**g).
6. Right-click the operation in the browser window and select Open Specification.
7. Set the operation visibility (public, private, protected).

Adding Class Relationships

There are many different types of relationships between classes. An *association relationship* indicates that one class needs to communicate with another. Associations may be unidirectional or bidirectional. An *aggregation relationship* suggests a whole/part relationship between two classes. A *generalization relationship* indicates a parent/child inheritance relationship between two classes. Finally, a *dependency relationship* is a weaker form of association, suggesting that a change to one class may affect another.

Association names can be added to a relationship to clarify the relationship's purpose. Role names can also be used that show what role each class plays in the relationship. Multiplicity settings show how many instances of one object are related to a single instance of the other object.

Relationships are drawn on Class diagrams as arrows between the two related classes. Different types of arrows are used to indicate different types of relationships.

An ***association*** relationship is a semantic connection between classes. It indicates that one class needs to communicate with another (for example, one class needs to send a message to the other). Unidirectional associations suggest that the messages can be sent in only one direction, while bidirectional associations suggest that messages can be sent in both directions (i.e., each class can call a method of the other).

Follow these steps to add association relationships between the classes:

1. Select the Unidirectional Association or the Bidirectional Association toolbar button.
2. Drag an arrow from one class to the other.
3. Type an association name, if needed.

An ***aggregation*** relationship is used to denote a whole/part relationship between classes. In this situation, one class logically contains another. Association and aggregation relationships are created identically during code generation.

To add aggregation relationships:

1. Select the Aggregation toolbar button.
2. Drag an arrow from the "whole" class to the "part" class.
3. Type an association name, if needed.

A ***generalization*** relationship is used to show an inheritance relationship between two classes. The child class inherits all attributes, operations, and relationships of the parent

To add generalization relationships:

1. Select the Generalization toolbar button.
2. Drag an arrow from the child class to the parent class.

Multiplicity shows how many instances of one class are related to a single instance of another class.

Multiplicity indicators are placed at both ends of a relationship to show the number of instances in both directions. Multiplicity is not included on a generalization relationship.

Follow these steps to add multiplicity to the relationships:

1. Right-click one end of the relationship.
2. Select Multiplicity and then the appropriate multiplicity setting for that end of the relationship.
3. Right-click the other end of the relationship and set its multiplicity.

Class diagrams: an example

A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static -- they display what interacts but not what happens when they do interact.

The class diagram below models a customer order from a retail catalog. The central class is the **Order**. Associated with it are the **Customer** making the purchase and the **Payment**. A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.

UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as ***Payment***, are in italics. Relationships between classes are the connecting links.

Our class diagram has three kinds of relationships.

- **association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.
- **aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, **Order** has a collection of **OrderDetails**.
- **generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.

A **navigability** arrow on an association shows which direction the association can be traversed or

queried. An **OrderDetail** can be queried about its **Item**, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, **OrderDetail** has an **Item**. Associations with no navigability arrows are bi-directional.

The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.

This table gives the most common multiplicities.

Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n* . . *m*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Every class diagram has classes, associations, and multiplicities. Navigability and roles are optional items placed in a diagram to provide clarity.

Building an Interaction Diagram

1. Introduction

An **Interaction diagram** is a graphical representation of how the objects and actors in a system interact with one another to achieve the desired goal of the system. There are two types of Interaction diagrams, *Sequence diagrams* and *Collaboration diagram*s. Sequence diagrams illustrate the interactions of objects along a timeline. Collaboration diagrams show the interactions, but without the timeline. These two diagrams display the same information, just in different ways.

You can create multiple Interaction diagrams for each use case in a system. More than one is typically created to illustrate the interaction given different scenarios.

2. Sequence Diagrams

Sequence diagrams are usually created to show the flow of functionality and control throughout the objects in the system. Collaboration diagrams are typically used to illustrate which objects communicate with other objects. Sequence diagrams answer questions about how the system will work. Collaboration diagrams answer questions about the soundness of the structure of the system.

To create a new Sequence diagram:

1. Right-click a use case in the Use Case view.
2. Select New → Sequence Diagram.

Once you have created the Sequence diagram, you next need to determine which actors and objects to place on it. One method is to drag all actors involved in the use case to the Sequence diagram, then walk through the functionality, adding objects to facilitate that functionality as needed.

To add actors and objects to the Sequence diagram:

1. Select any actors/classes involved in the use case/class diagram and drag them into the new Sequence diagram.
2. To add additional objects, select the Object toolbar button.
3. Click in the Sequence diagram to add the object.
4. Name the object.
5. Add more objects as needed.

At this point, the objects and actors are on the Sequence diagram, but Interaction diagrams would be fairly useless without showing the interactions. Messages are used to accomplish this. A message is simply some form of communication between one object or actor and another.

Messages can also be *reflexiv*e, meaning that the object communicates some information to itself.

Follow these steps to add messages to the diagram:

1. Select the Message toolbar button.
2. Click on the actor or object to initiate the message.
3. Drag the message to the actor or object to receive the message.
4. Name the message.

Tip: Pressing F5 on a Sequence diagram will create the corresponding Collaboration diagram. You can also use F5 to toggle between a Sequence diagram and its Collaboration diagram.

3. Collaboration Diagrams

Collaboration diagrams are not time-based, but show the interactions of the objects as a whole. This is useful for pointing out potential bottlenecks (objects that individually communicate with a large number of objects). Sequencing of messages is shown on a Collaboration diagram by numbering the messages.

To create a new Collaboration diagram:

1. Right-click a use case in the Use Case view.
2. Select New → Collaboration Diagram.

Once you have created the Collaboration diagram, you then need to determine which actors and objects to place on it. One method is to drag all actors involved in the use case to the Collaboration diagram, and then walk through the functionality, adding objects as needed to facilitate that functionality.

To add actors and objects to a Collaboration diagram:

1. Select any actors involved in the use case and drag them into the new Collaboration diagram.
2. To add additional objects, select the Object toolbar button.
3. Click in the Collaboration diagram to add the object.
4. Name the object.
5. Add more objects as needed.

Before adding messages, we must link the objects or actors that will communicate. This is done by adding an object link using the toolbar. Once the object link is established, messages can be added. On Collaboration diagrams, the messages are either *link* messages or *links to self*. A link message is analogous to a message in a Sequence diagram. A link to self is analogous to a reflexive message in a Sequence diagram.

To add object links to the Collaboration diagram:

1. Select the Object Link toolbar button.
2. Click on one actor or object to be linked.
3. Drag the object link to the actor or object to be linked.

After the object link is established, messages can be created. Link messages are created either using the Link Message toolbar button or the Reverse Link Message toolbar button, depending on the direction of the message.

Follow these steps to add messages to the Collaboration diagram:

1. Select the Link Message or Reverse Link Message toolbar button.
2. Click the object link on which to place the message.
3. Name the message.

To add reflexive links to a Collaboration diagram:

1. Select the Link to Self toolbar button.
2. Click on the object to be linked.
3. Name the reflexive link.

Tip: Pressing F5 on a Collaboration diagram will create the corresponding Sequence diagram.

Which of the following pieces of sequence diagrams represents this action?

"Object **A** sends object **B** several messages named `msg`."

a)

b)

c)

d)

Sequence diagrams: an example

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is **a Reservation window**.

The **Reservation window** sends a `makeReservation()` message to a **HotelChain**. The **HotelChain** then sends a `makeReservation()` message to a **Hotel**. If the **Hotel** has available rooms, then it makes a **Reservation** and a **Confirmation**.

Each vertical dotted line is a **lifeline**, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the **activation bar** of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message.

In our diagram, the **Hotel** issues a **self call** to determine if a room is available. If so, then the **Hotel** creates a **Reservation** and a **Confirmation**. The asterisk on the self call means **iteration** (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, [ ], is a **condition**.

Collaboration diagrams: an example

The object-role rectangles are labeled with either class or object names (or both). Class names are preceded by colons (:).

Each message in a collaboration diagram has a **sequence number**. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Sequence diagrams: a second example

Messages are indicated on UML sequence diagrams as labeled arrows, when the source and target of a message is an object or class the label is the signature of the method invoked in response to the message. However, if either the source or target is a human actor, then the message is labeled with brief text describing the information being communicated. For example, in the figure below the *EnrollInSeminar* object sends the message *isEligibleToEnroll(theStudent)* to the instance of *Seminar*. Notice how I include both the method's name and the name of the parameters, if any, passed into it. The *Student* actor provides information to the *SecurityLogon* object via the messages labeled *name* and *student number* (these really aren't messages, they are actually user interactions).

Return values are optionally indicated using a dashed arrow with a label indicating the return value. For example, the return value *theStudent* is indicated coming back from the *Student* class as the result of invoking a message, whereas no return value is indicated as the result of sending the message *isEligibleToEnroll(theStudent)* to *Seminar*. My style is not to indicate the return values when it's obvious what is being returned, so I don't clutter my sequence diagrams (as you can see, sequence diagrams get complicated fairly quickly).

Enroll In Seminar
Basic Course of Action
SD#: UC17-01
A Student
<<actor>>
:MainMenu
<<UI>>
:EnrollInSeminar
<<controller>>
:SecurityLogon
<<UI>>
:SeminarSelector
<<UI>>
:FeeDisplay
<<UI>>
Student
seminar:Seminar
theStudent
:Student
schedule
:StudentSchedule
:StudentFees
1. Student indicates wish to enroll
wish to enroll
2. Student inputs name and number
<<create>>
name
student number
3. System verifies student
isEligible(name, studentNumber)
<<create>>
theStudent
theStudent
<<destroy>>
Note: Need to flesh this message out more.
4. System displays seminar list
<<create>>
5. Students picks seminar
selection
6. System determines eligibility to enroll
isEligibleToEnroll(theStudent)
qualifications()
getSchedule()
7. System determines schedule fit
determineFit(seminar)
8. System calculates fees
calculateFees(seminar, theStudent)
9. System displays fees
<<create>>
10. System verifies student wishes to enroll
11. Students indicates yes.
verification
12. System enrolls student in seminar
enrollStudent(theStudent)