UNIFIED MODELING LANGUAGE - UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

- UML stands for Unified Modeling Language.
- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to *visualize, specify, construct, and document software system*.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization, UML has become an OMG standard.

Goals of UML

A picture is worth a thousand words, this idiom absolutely fits describing UML. The most important goal is to define some general purpose modeling language, which all modelers can use and it also needs to be made simple to understand and use.

UML diagrams are not only made for developers but also for business users, common people, and anybody interested to understand the system. Thus it must be clear that UML is not a development method rather it accompanies with processes to make it a successful system.

The primary goals in the design of the UML are as follows:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.

6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

Uses of UML

UML is quite useful for the following purposes −

- Modeling the business process
- Describing the system architecture
- Showing the application structure
- Capturing the system behavior
- Modeling the data structure
- Building the detailed specifications of the system
- Sketching the ideas
- Generating the program code

Static Models

Static models show the structural characteristics of a system, describe its system structure, and emphasize on the parts that make up the system.

- They are used to define class names, attributes, methods, signature, and packages.
- UML diagrams that represent static model include class diagram, object diagram, and use case diagram.

Dynamic Models

Dynamic models show the behavioral characteristics of a system, i.e., how the system behaves in response to external events.

- Dynamic models identify the object needed and how they work together through methods and messages.
- They are used to design the logic and behavior of system.
- UML diagrams represent dynamic model include sequence diagram, communication diagram, state diagram, activity diagram.

USE CASE DIAGRAM

A use case diagram is a dynamic or behavior diagram in UML. Use case diagrams are valuable for visualizing the functional requirements of a system that will translate into design choices and development priorities. They also help identify any internal or external factors that may influence the system and should be taken into consideration.

They model the functionality of a system using actors and use cases. Use cases are a set of actions, services, and functions that the system needs to perform. In this context, a "system" is something being developed or operated, such as a web site. The "actors" are people or entities operating under defined roles within the system.

They provide a good high level analysis from outside the system. Use case diagrams specify how the system interacts with actors without worrying about the details of how that functionality is implemented.

UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

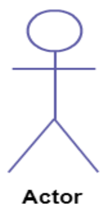
Use case diagram components

Common components include:

Actor

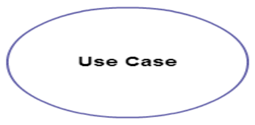
Actor in a use case diagram is any entity that performs a role in one given system.

An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data. They are stick figures that represent the people actually employing the use cases.



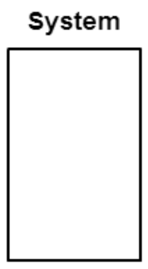
Use Case

A use case represents a function or an action within the system (different uses that a user might have). It’s drawn as an oval and named with the function.



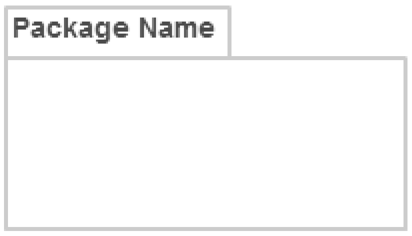
System

The system is a specific sequence of actions and interactions between actors and the system. It is used to define the scope of the use case.
A system may also be referred to as a scenario.
It is drawn as a rectangle.



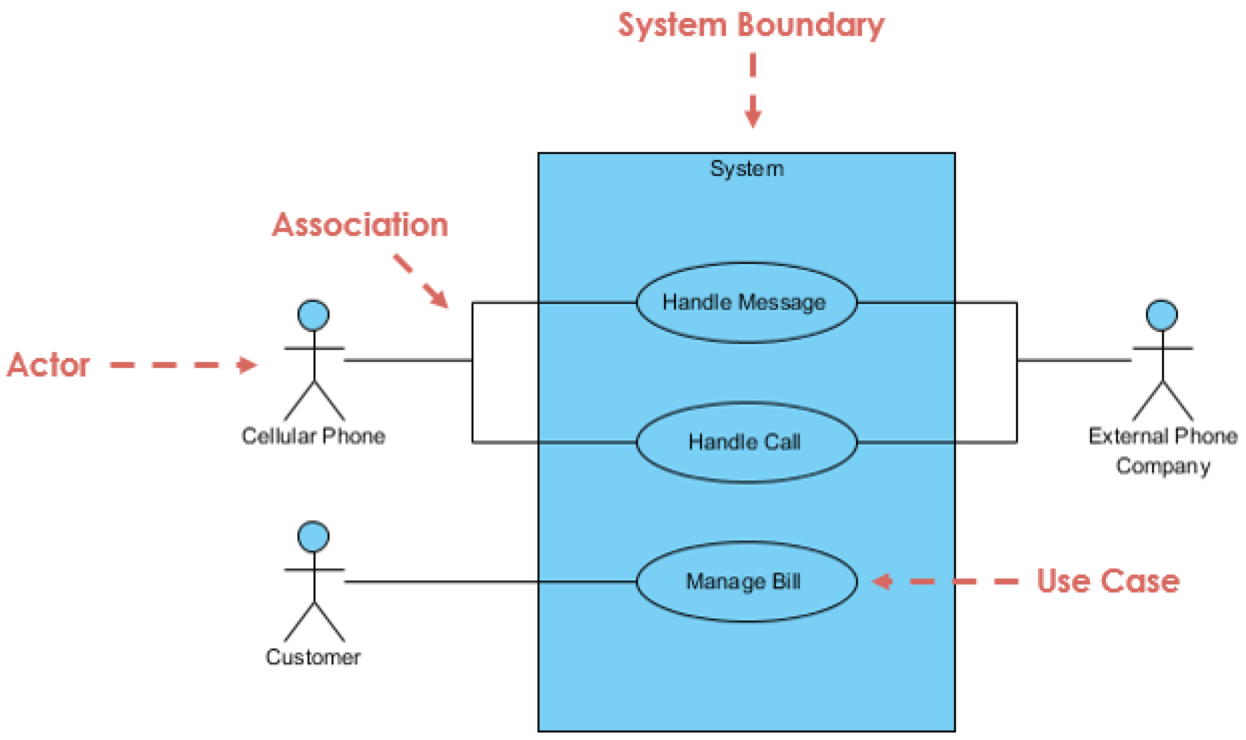
Package

The package is an optional element that is extremely useful in complex diagrams. Similar to class diagrams and component diagrams, packages are used to group together use cases. These groupings are represented as file folders.



- *Associations*: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- *Goals*: The end result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

Use Case Diagram at a Glance



Use Case Diagram Guidelines

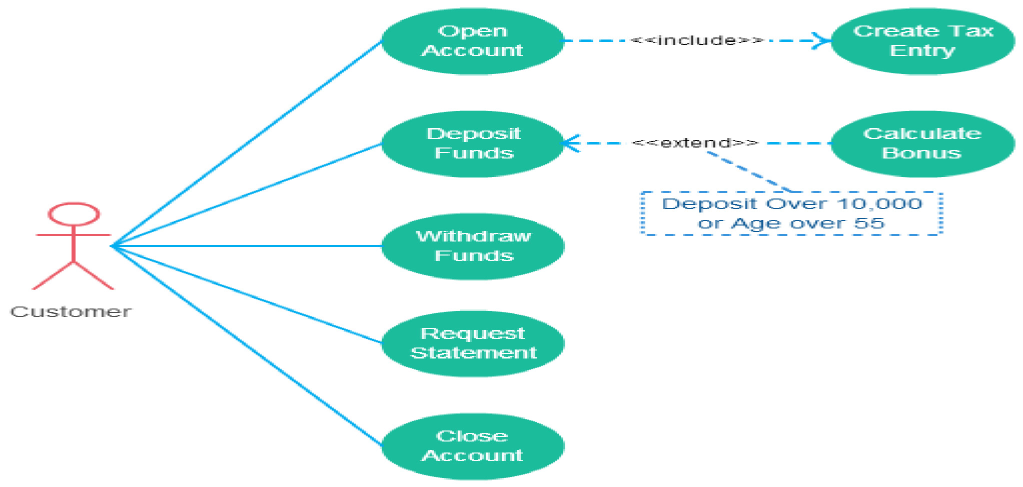
A use case diagram mainly consists of actors, use cases and relationships. More complex larger diagrams can include systems and boundaries.

Actors

- Give meaningful business relevant names for actors – For example, if your use case interacts with an outside organization its much better to name it with the function rather than the organization name.
- Primary actors should be to the left side of the diagram – This enables you to quickly highlight the important roles in the system.
- Actors model roles (not positions)
- External systems are actors
- Actors don't interact with other actors – In case actors interact within a system you need to create a new use case diagram with the system in the previous diagram represented as an actor.
- Place inheriting actors below the parent actor – This is to make it more readable and to quickly highlight the use cases specific for that actor.

Use Cases

- Names begin with a verb – A use case models an action so the name should begin with a verb.
- Make the name descriptive – This is to give more information for others who are looking at the diagram. For example "Print Invoice" is better than "Print".
- Highlight the logical order – For example, if you're analyzing a bank customer typical use cases include open account, deposit and withdraw. Showing them in the logical order makes more sense.
- Place included use cases to the right of the invoking use case – This is done to improve readability and add clarity.
- Place inheriting use case below parent use case – Again this is done to improve the readability of the diagram.



Systems / Packages

- Use them sparingly and only when necessary
- Give meaningful and descriptive names to these objects

Relationships

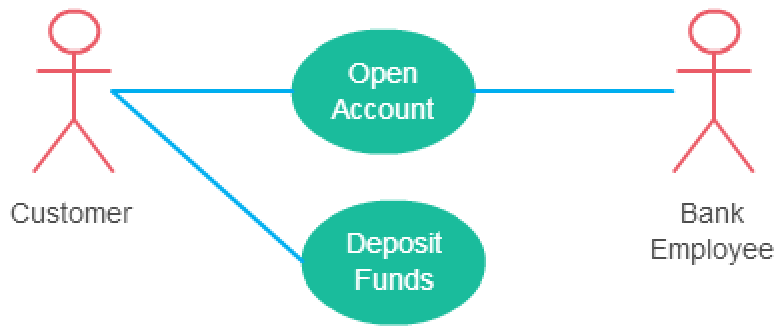
There can be 5 relationship types in a use case diagram.

- Association between actor and use case
- Generalization of an actor
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

1. Association between Actor and Use Case

This one is straightforward and present in every use case diagram.

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.

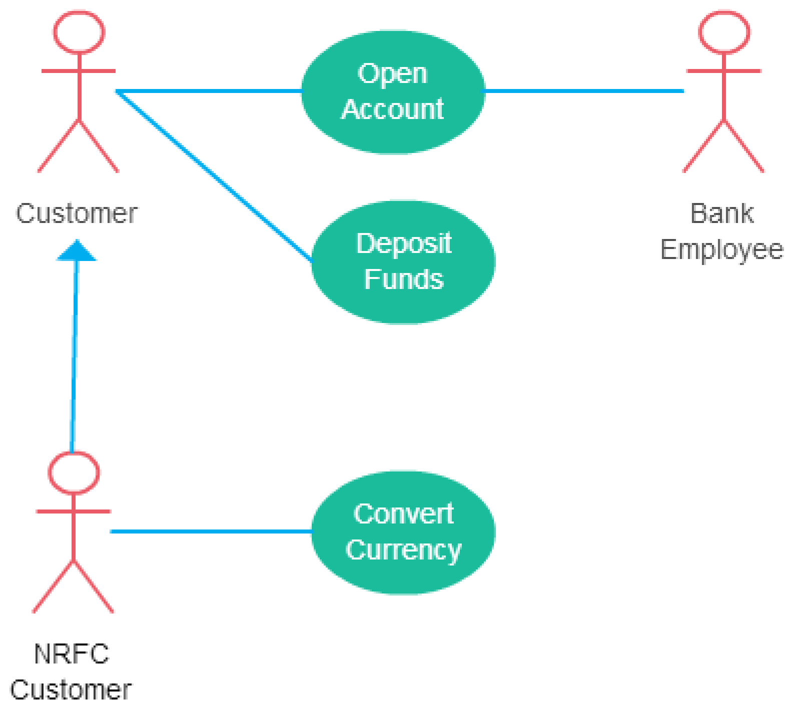


Different ways association relationship appears in use case diagrams

2. Generalization of an Actor

Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role.

Let's expand the previous use case diagram to show the generalization of an actor.

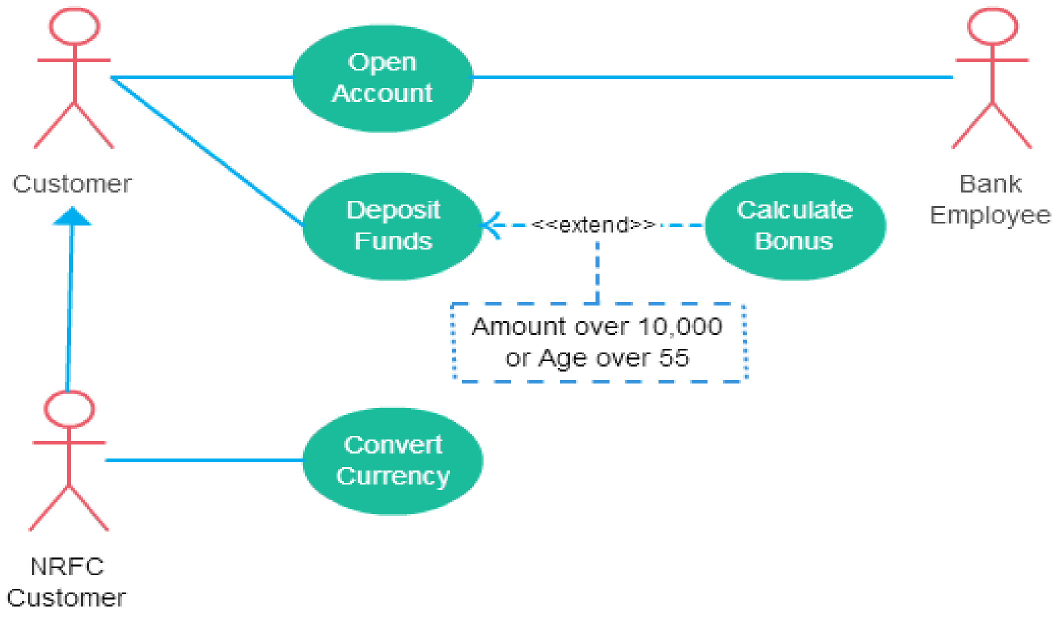


A generalized actor in a use case diagram

3. Extend Relationship between Two Use Cases

Extend relationship extends the base use case and adds more functionality to the system. Consider the following when using the <<**extend**>> relationship.

- The extending use case is dependent on the extended (base) use case. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- The extending use case is usually optional and can be triggered conditionally. In the diagram, extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- The extended (base) use case must be meaningful on its own. This means it should be independent and must not rely on the behavior of the extending use case.



Extend relationship in use case diagrams

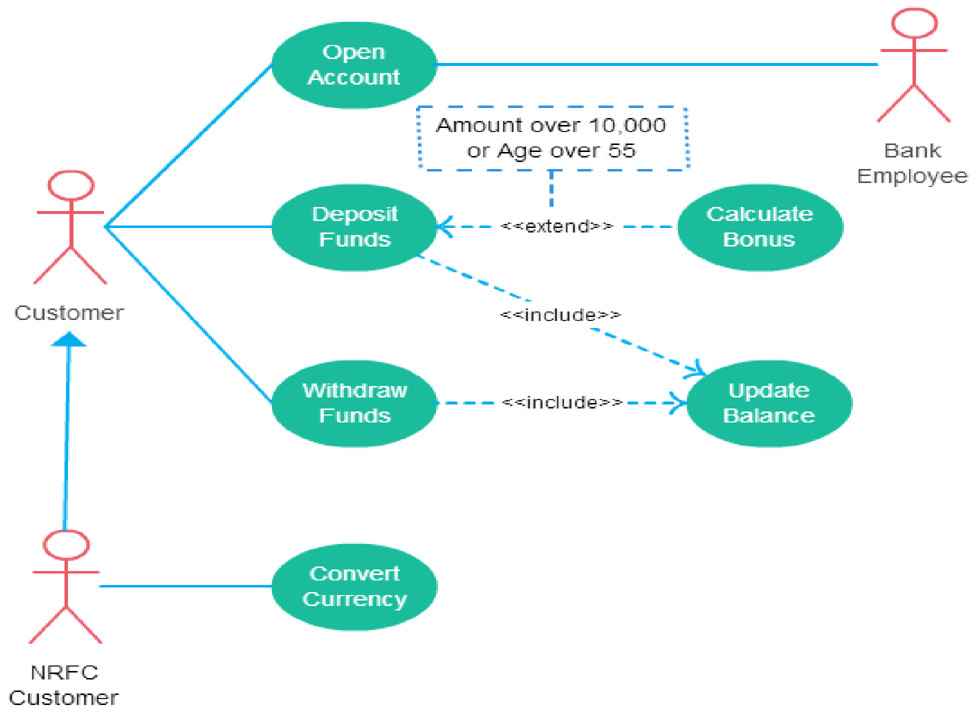
4. Include Relationship between Two Use Cases

Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases.

In some situations, this is done to simplify complex behaviors.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

Let’s expand our banking system use case diagram to show include relationships as well.



Includes is usually used to model common behavior

5. Generalization of a Use Case

This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.

For example, in the previous banking example, there might be a use case called “Pay Bills”. This can be generalized to “Pay by Credit Card”, “Pay by Bank Balance” etc.

How to Create a Use Case Diagram

Identifying Actors

The following questions can help you identify the actors of your system

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a present time?

Identifying Use Cases

A good way to identify use cases is to identify what the actors need from the system.
Top level use cases should always provide a complete function required by an actor. You can extend or include use cases depending on the complexity of the system.
Once you identify the actors and the top level use case you have a basic idea of the system. Now you can fine tune it and add extra layers of detail to it.

The following questions can be asked to identify use cases, once your actors have been identified:

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about changes in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?

Look for Common Functionality to use Include

Look for common functionality that can be reused across the system. If you find two or more use cases that share common functionality you can extract the common functions and add it to a separate use case. Then you can connect it via the include relationship to show that it’s always called when the original use case is executed.

Is it Possible to Generalize Actors and Use Cases

There may be instances where actors are associated with similar use cases while triggering a few use cases unique only to them. In such instances, you can generalize the actor to show the inheritance of functions. You can do a similar thing for use case as well.

Optional Functions or Additional Functions

There are some functions that are triggered optionally. In such cases, you can use the extend relationship and attach an extension rule to it.
Extend doesn’t always mean it’s optional. Sometimes the use case connected by extending can supplement the base use case.
The thing to remember is that the base use case should be able to perform a function on its own even if the extending use case is not called.

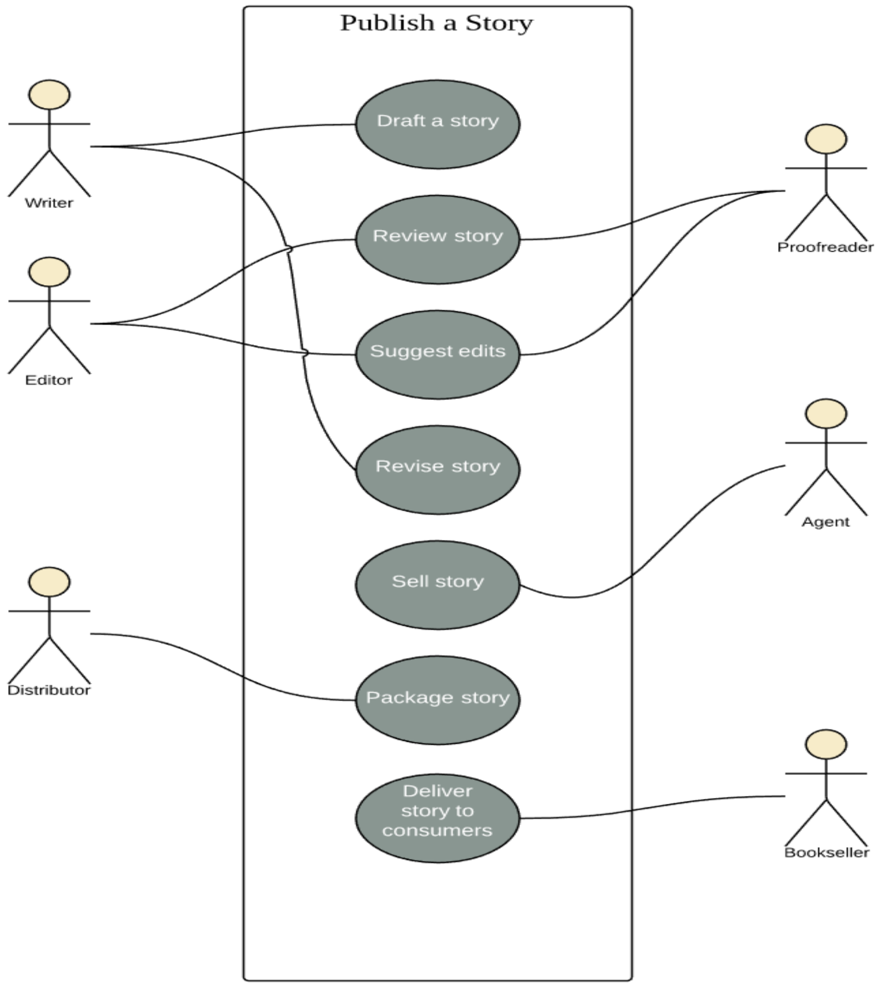
Use Case Diagram Tips

- Always structure and organize the use case diagram from the perspective of actors.
- Use cases should start off simple and at the highest view possible. Only then can they be refined and detailed further.
- Use case diagrams are based upon functionality and thus should focus on the "what" and not the "how".

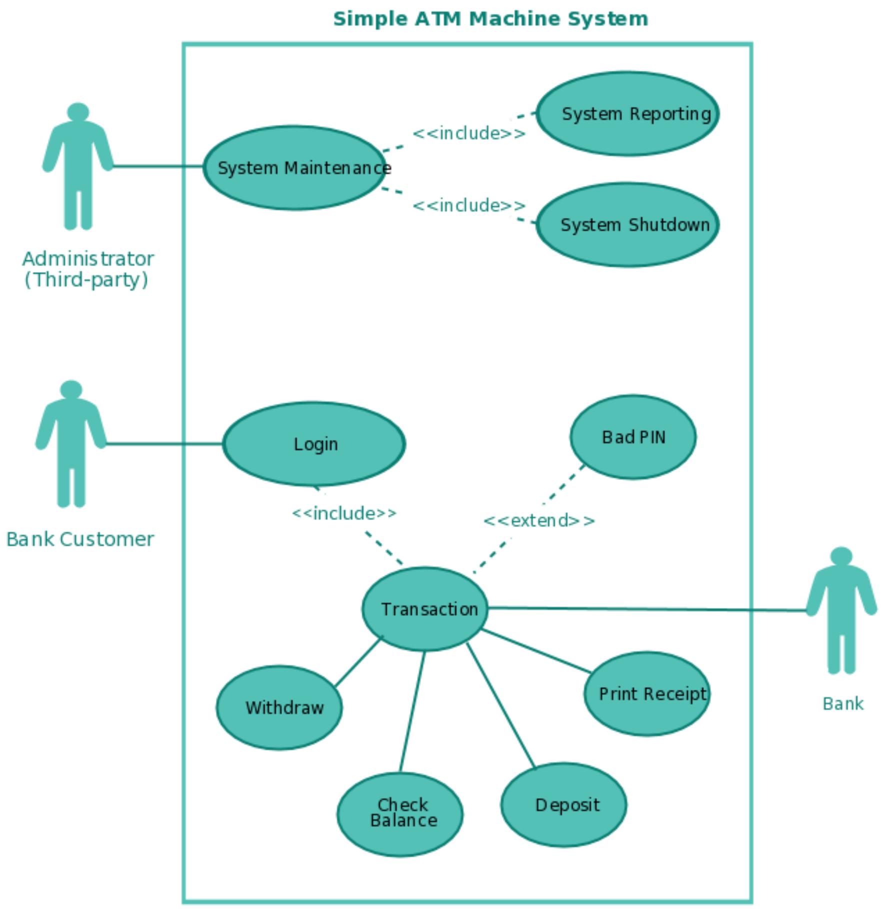
Use case diagram examples

Book publishing use case diagram example

This use case diagram is a visual representation of the process required to write and publish a book. Whether you're an author, an agent, or a bookseller, inserting this diagram into your use case scenario can help your team publish the next big hit.

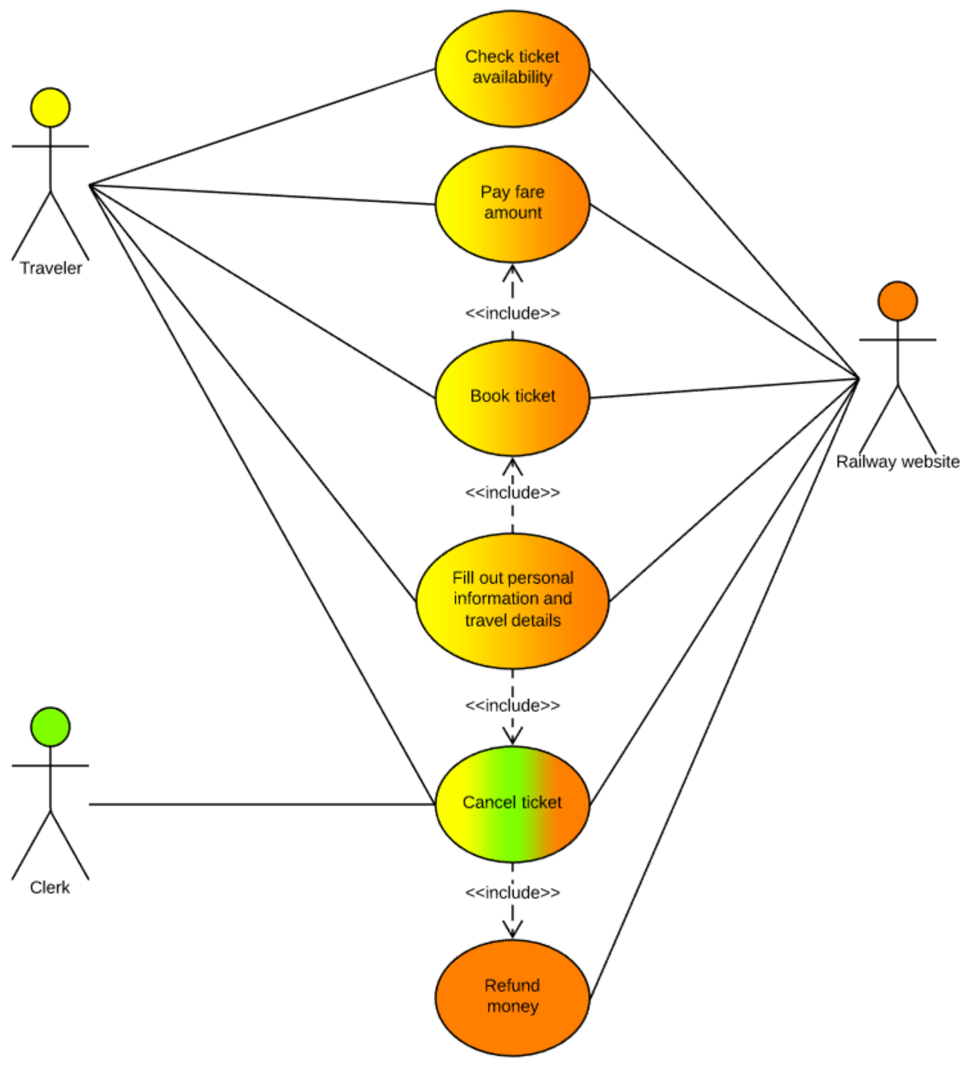


Use Case Diagram Templates



A use case template for an ATM system

Railway reservation use case diagram example


Check ticket availability
Pay fare amount
Traveler
<<include>>
Book ticket
Railway website
<<include>>
Fill out personal information and travel details
<<include>>
Cancel ticket
Clerk
<<include>>
Refund money