Use-Case Diagrams

A use-case diagram is a UML diagram that provides a high-level view of the behaviors in a system or in part of a system. Use case diagrams also identify users of a system and their interactions with the system. A use-case diagram can depict all or some of the use cases in a system.

The major behaviors of a system are specified with use cases. Use cases indicate that the behaviors exist and describe the value that the results give to users. They do not describe how the behaviors are implemented. The external users of the system are presented as actors. Actors present the different roles people, hardware, and other systems play when they interact with the system. In addition, connectors in use case diagrams identify the relationships between the model elements.

The use cases and actors in use-case diagrams describe a system from an outsider's perspective. They do not describe how the system operates internally.

Using Use-Case Diagrams

Use-case diagrams are helpful because they illustrate and define the context and requirements of either an entire system or important parts of it. Use-case diagrams also identify the interactions between the system and its various users. Use-case diagrams are usually developed in the early phases of a software project and used throughout the development process. Possible uses of use-case diagrams include the following:

- Before starting the software project, you can create use-case diagrams to model a business so that all participants in the software project share an understanding of the workers, customers, entities, and activities of the business.
- While gathering requirements, you can create use-case diagrams to capture the system requirements and to present to others what the system should do.
- During analysis and design, you can use the use cases and actors in existing use-case diagrams to identify the classes that are required by the system.
- During testing, you can use existing use-case diagrams to identify tests that should be performed on the system.

Shapes and Connectors

A use-case diagram can contain shapes and connectors that represent:

- actors
- use cases
- access relationships
- association relationships

- dependency relationships
- directed association relationships
- extend relationships
- generalization relationships
- import relationships
- include relationships

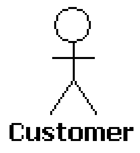
Actors

An actor is a model element that describes a role that a user plays when interacting with the system being modeled. Actors, by definition, are external to the system. Although an actor typically represents a human user, it can also represent an organization, system, or machine that interacts with the system. An actor can correspond to multiple real users, and a single user may play the role of multiple actors.

For example, a use-case model for an e-commerce system may contain a "Customer" actor to represent a user who uses the system to buy products.

Shape

An actor usually appears as a "stick man" shape.



By default, actor shapes show only their names. If desired, you can also show the actor's attributes, operations, and signal receptions under its shape using the Appearance Toolbar.

Features of Actors

If desired, you can add the following to actors:

- Attributes – Identify states in the actor.
- Operations – Identify the work that the actor can perform.
- Signal Receptions - Identify signals from the system to which the actor responds.
- Documentation – Defines what the actor does and how the actor interacts with the system. For details, see Documenting Model Elements.

Using Actors

You can add actors to your model to represent the following:

- In models depicting businesses, actors represent the types of individuals and machines that interact with a business. Examples include customers, suppliers, partners, and external computer systems.
- In models depicting software applications, actors represent the users of the system. Examples include end users, external computer systems, and system administrators.

The actors in a model usually appear in use-case diagrams. They sometimes appear in class and sequence diagrams.

Note If you want to nest an actor within a classifier, drag the actor on top of the classifier in the Model Explorer.

Naming Conventions

Each actor has a unique name that describes the role the user plays when interacting with the system.

Attributes

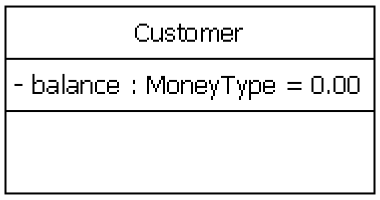
An attribute is a model element that represents a data definition for a classifier. It also describes a range of values that instances of the classifier may have for the data definition. Usually, an attribute has a type-expression, which defines the type of value (for example, Boolean or integer) and the range (for example, -32,768 to 32,767). A constraint may also be attached to an attribute to define the range of values it can hold.

A classifier may have any number of attributes or none at all. A class's attributes describe its structure, and the value of an instance's attributes define its state. For example, in an e-commerce application, one of the attributes that a "Customer" class may have is a "balance" attribute that holds the amount of money in the customer's balance.

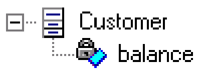
Shape

In the diagram window, attributes appear (if the attribute compartment is displayed) in the classifier shape for which they are defined. In the Model Explorer, attributes appear under the classifier for which they are defined.

Attributes in the Diagram Window



Attributes in the Model Explorer



You can show or hide attributes in the classifier shapes for which they are defined. For details, see Showing and Hiding Attributes in Classifiers. You can show the visibility of attributes as text symbols (for example, -) or icons (for example,). For details, see Specifying Visibility Styles.

Using Attributes

You can add attributes to some types of classifiers in your model to identify their properties. For details about attributes in each type of classifier, see Actors, Classes, Signals, and Uses Cases.

Naming Conventions

Attributes have names that are short nouns or noun phrases describing their contents. Additional information, including the visibility, type, and initial value, can be provided about the attribute in its name. The syntax for an attribute name is:

visibility «stereotype» name : type-expression = initial-value

Note If a stereotype is not used, the guillemets (« ») do not appear, except when you are editing the operation name.

For example, in an e-commerce application, a "Customer" class may have a "- balance : MoneyType = 0.00" attribute. The attribute name provides the following information.

Part of Syntax	**Example**	**Description**
visibility	-	The minus sign indicates the attribute has private visibility. Only operations in the class can access it.
name	balance	The name describes the information about the customer that the attribute holds.
type-expression	MoneyType	The attribute is an instance of the "MoneyType" class, which holds an amount of money.
initial-value	0.00	The initial value assigned to the attribute is "0.00."

Operations

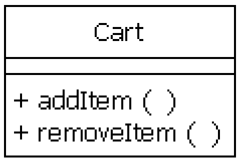
An operation is a model element that represents a service that a classifier or its instances may be requested to perform. Operations are most often contained by classes, but they can also be contained by other classifiers (for example, actors, use cases, or components). A classifier may have any number of operations or none at all.

In classes, operations are implementations of functions or queries that an object may be called to perform. Well-defined operations do only one thing. For example, in an e-commerce system, you may make a "Cart" class responsible for adding and removing merchandise that a customer plans to buy. To do so, you can add an "addItem()" operation that adds merchandise to the cart and a "removeItem()" operation that removes merchandise.

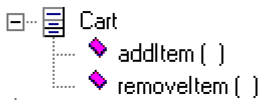
Shape

In the diagram window, operations appear (if the operation compartment is displayed) in or under the classifier shape for which they are defined. In the Model Explorer, operations appear under the classifier for which they are defined.

Operations in the Diagram Window



Operations in the Model Explorer



You can show or hide operations in the classifier shapes for which they are defined. For details, see Showing and Hiding Operations in Classifiers. You can show the visibility of operations as text symbols (for example, +) or icons (for example, ◆) in classifier shapes that show operations. For details, see Specifying Visibility Styles.

Using Operations

You can add operations to many types of classifiers in your model to identify their behaviors. For details about operations in each type of classifier, see Actors, Classes, Enumerations, Interfaces, Signals, Subsystems, and Uses Cases.

The operations in a model usually appear in classifier shapes in various diagrams, including class, component, and use-case diagrams.

Naming Conventions

Each operation in a classifier must have a unique signature. A signature includes the operation name and its ordered list of parameter types. The syntax for an operation name is:

visibility «stereotype» name(parameter list) : return-type

Note If a stereotype is not used, the guillemets (« ») do not appear, except when you are editing the operation name.

For example, in an e-commerce application, a "Customer" class may have a "+ getBalance([in] day: Date) : MoneyType" operation. The operation name indicates the following information.

Part of Syntax	Example	Description
visibility	+	The plus sign indicates the operation has public visibility. Operations that are in other classes can call it.
name	getBalance	The name describes the operation according to its outcome and from its user's perspective.
parameter list	([in] day: Date)	The operation has one input parameter named "day" followed by its type ("Date"). You can display the complete signature, which includes the parameter list, or just the operation name.
return-type	: MoneyType	The type of value returned by the operation is an instance of the "MoneyType" class, which specifies an amount of money.

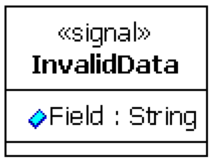
Signals

A signal is a model element that specifies a one-way, asynchronous communication from one object to one or more other objects.

Signals are often used in event-driven systems. For example, a communications system may contain a "Pager" class, whose objects wait for and respond to "Page" signals.

Shape

A signal usually appears as a rectangle with three compartments. Its upper compartment contains the «signal» keyword and the signal name. Its middle compartment contains the attributes. Its lower compartment contains the operations.



You can show or hide the attribute and operation compartments. For details, see Showing and Hiding Attributes in Classifiers and Showing and Hiding Operations in Classifiers.

Features of Signals

If desired, you can add the following to signals:

- Attributes – Identify the parameters that the signal provides.
- Operations – Identify the operations that can access and modify the attributes of the signal. All signals are assumed to have a "send( )" operation.

Using Signals

You can add signals to your model to represent the following:

- In models depicting event-driven systems, signals represent the signals that objects instantiated from classes are prepared to handle, thus triggering events.
- In models depicting software systems, signals can represent exceptions that an operation can throw when something unexpected occurs.

The signals in a model usually appear in class diagrams.

Naming Conventions

A signal has a name that describes its purpose in the system.

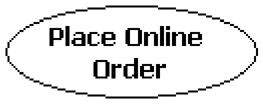
Use Cases

A use case is a model element that describes behaviors that a system performs to yield observable, valuable results for actors. The use case specifies the behaviors of the system that the actor sees, but it does not reveal the internal structure that implements those behaviors.

For example, an e-commerce application may contain a "Place Online Order" use case to describe the actions that a customer follows to buy an item. In most cases, you link documents or diagrams (usually activity diagrams) to the use case to detail its flow of events, preconditions, and so on.

Shape

A use case usually appears as an oval.



Features of Use Cases

If desired, you can add the following to use cases:

- Attributes – Identify objects in a use case that occur during its execution.
- Operations – Describe the work that can take place in a use case and its effect on the system. These operations cannot be called from a client.
- Signal Receptions - Identify external stimuli that can trigger the use case.
- Documentation – Details the purpose and flow of events of a use case. One possible document includes a report that describes the use case, its flows of events, its preconditions, its postconditions, and special requirements. Other possible documents include activity diagrams that illustrate flows of events and screen captures of user-interface prototypes.

Using Use Cases

You can add use cases to your model to represent the following:

- In models depicting businesses, use cases represent the processes and activities in the business.
- In models depicting software systems, use cases represent the functionality of the system from the actor's point of view.

The use cases in a model usually appear in use-case diagrams.

Naming Conventions

Each use case has a unique name in its enclosing namespace. The name typically describes the action that the system performs from the actor's perspective. Use-case names are often short phrases that start with a verb.

Access Relationships (Java)

An access relationship to a modeled element results in an Import statement being generated in the source code for the element. It displays on a diagram as a Dependency with an «Access» stereotype.

During reverse-engineering, an Import or Import on Demand statement in the Java source results in an Access relationship between the modeled element and the Imported element.

Association Relationships

An association is a structural relationship showing that objects of one classifier (actor, use case, class, interface, node, or component) are connected and can navigate to objects of another classifier. An association connects two classifiers—supplier classifier consumer classifier—even in bi-navigable associations. Associations help you make design decisions about the structure of your data—not only the classes needed to contain the data, but also which classes need to share data with other classes. An association supports data sharing between classes or, in the case of a self-association, between objects of the same class.

For example, in an e-commerce application, a "Customer" class may have a single association (1) to an "Account" class, indicating that each "Account" instance is owned by one "Customer" instance. Also, if you have an "Account," you can locate its owning "Customer." Furthermore, with a given "Customer," you can navigate to each "Account." The association between the "Customer" class and the "Account" class is important because it shows the structure between each classifier.

Association Ends

An association end contains the properties that specify how one classifier structurally relates to the other classifier in the association. Both ends of an association can connect to the same class to show a self-association, indicating that given an object of this class, you can navigate to other objects of the same class.

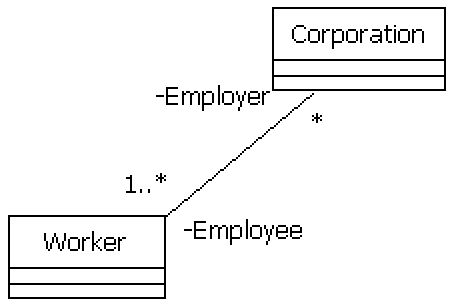
Appearance and Semantics of Association Relationships

Association appearance and semantics are determined by the properties of the association and its ends. The toolbox contains several associations tools with predefined properties. After an association is created, you can modify its appearance and semantics by changing its properties in the Properties window.

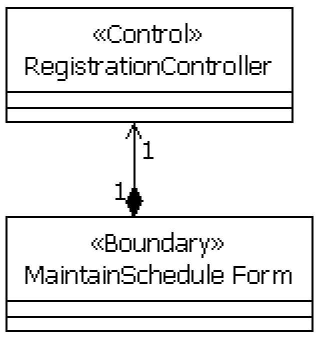
Connector

A bi-navigable association appears as a solid line between two classifiers.

Association relationship with adornments between two classifiers



Composition association relationship between two classifiers



Naming Conventions

You can name an association to describe the nature of the relationship between the two classifiers; however, names are typically not needed, especially if you use association end names.

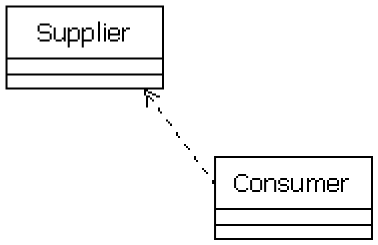
Dependency Relationships

A dependency relationship indicates that a change to one model element (the supplier) may cause a change in the other model element (the consumer). The supplier model element is independent because a change in the consumer does not affect it. In contrast, the consumer model element is dependent on the supplier model element because a change to the supplier may also affect it.

In an e-commerce application, for example, a "Cart" class depends on a "Product" class because the "Product" class is used as a parameter for an "add" operation in the "Cart" class. In a class diagram, a dependency relationship points from the "Cart" class to the "Product" class. In other words, the "Cart" class is the consumer model element, and the "Product" class is the supplier model element. This relationship indicates that a change to the "Product" class may require a change to the "Cart" class.

Connector

A dependency appears as a dashed line with an open arrow. It points from the consumer model element to the supplier model element.



Types of Dependency Relationships

Because a dependency relationship may represent several different types of relationships, you typically use a keyword or stereotype to show the precise nature of the dependency.

Type	**Keyword/Stereotype**	**Description**
Abstraction	«abstraction», «derive», «realize», «refine», or «trace»	Relates two model elements or sets of model elements that represent the same concept at different levels of abstraction or from different viewpoints.

Binding	«bind»	Binds template arguments to template parameters in order to create model elements from templates.
Permission	«access», «friend», or «import»	Grants permission for one model element to reference the model elements owned by another model element.
Usage	«call», «create», «friendusage», «instantiate», «send», or «use»	Indicates that one model element requires the presence of another model element for its correct implementation or functioning.

Note A number of relationships use a connector similar to that of a dependency relationship but are not considered types of dependency relationships. These relationships include Extend Relationships, Include Relationships, and Note Attachments

Using Dependency Relationships

You can add dependency relationships to your model to do the following:

- Connect two packages to indicate that at least one model element in the consumer package is dependent on one model element in the supplier package. The dependency relationship does not indicate that all model elements in the consumer package are dependent.
- Connect two classes to indicate that there is a connection between them that is more temporary than an association relationship. For details on association relationships, see Association Relationships. Specifically, a dependency relationship indicates that the consumer class does one of the following: temporarily uses a supplier class that has global scope, temporarily uses a supplier class as a parameter for one of its operations, temporarily uses a supplier class as a local variable for one of its operations, or sends a message to a supplier class.
- Connect components to interfaces or other components to indicate that they use one or more of the operations specified by the interface or depend on the other component during compilation.

The dependency relationships in a model appear in a variety of diagrams, including class, component, deployment, and use-case diagrams.

Naming Conventions

Dependency relationships typically do not have names.

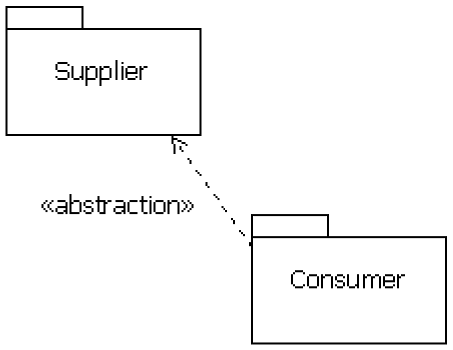
Abstraction Relationships

An abstraction relationship is a type of dependency relationship that relates two model elements (or sets of model elements) that represent the same concept at different levels of abstraction or from different viewpoints. In most cases, one of the model elements is more detailed than the other one. The more detailed model element is the consumer in the relationship, and the less detailed model element is the supplier. Types of abstraction relationships include derivation, realization, refinement, and trace relationships.

All types of abstraction relationships can connect model elements that are in the same model; realization, refinement, and trace relationships can also connect model elements in different models. For example, if you develop an analysis model and then a design model, you can connect them with a trace relationship pointing from the analysis model to the design model to indicate that the design model provides a different level of abstraction about the same system.

Connector

An abstraction relationship appears as a dashed line with an open arrow. It points from the consumer model element to the supplier model element. When you first create an abstraction relationship, the «abstraction» keyword appears next to the connector.



You should assign a stereotype to the abstraction relationship to indicate the type of abstraction.

Types of Abstraction Relationships

The type of abstraction is defined by a stereotype. The following table identifies the predefined stereotypes for abstraction relationships.

Name	**Keyword**	**Description**
Derivation	«derive»	Indicates that a consumer model element, such as a constraint, contains the details or formula for computing a derived model element (the supplier). For details on derived model elements, see Specifying Derived for Attributes and Associations.
Realization	«realize»	Indicates that the supplier model element provides a specification that the consumer model element implements.

Refinement	«refine»	Indicates that the consumer model element represents a more developed specification than the supplier model element.
Trace	«trace»	Indicates that the consumer model element is a historical development of the supplier model element.

Using Abstraction Relationships

You can add abstraction relationships to your model to indicate the following:

- Derivation abstraction relationships connect constraints that detail a derivation to the derived model element. You can also use a constraint attachment relationship to connect the constraint to the derived model element.
- For details about using realization relationships, see Realization Relationships.
- Refinement abstraction relationships indicate iterative development, optimization, and transformations between the model elements that they connect.
- Trace abstraction relationships track requirements and changes across models.

The abstraction relationships in a model appear in a variety of diagrams, including use-case, class, and component diagrams.

Naming Conventions

Abstraction relationships typically do not have names.

Binding Relationships

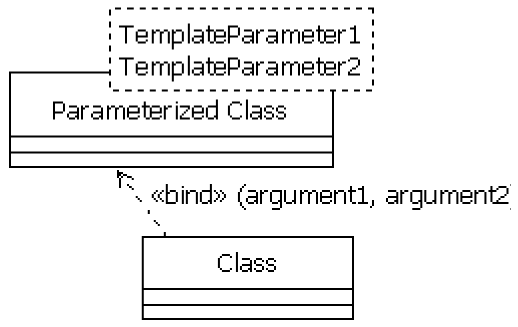
A binding relationship is a type of dependency relationship that assigns values to template parameters in order to generate a new model element (a class or collaboration) from the template (a template class or template collaboration). The template is the supplier in the relationship, and the model element is the consumer. The binding does not affect the template, so the template can be bound to any number of model elements. The binding does affect the model element, however, because the model element is defined by replacing the template parameters with the template arguments provided by the binding relationship.

For example, to create a "BookList" class from a "List" template class, you can add a binding relationship pointing from the "BookList" class to the "List" template class. The binding relationship provides "Book" (a class) and "50" (an integer value) as template arguments for the "List" template's "ItemType" and "Size" template parameters. (Note that these arguments must match the type specified by the template parameters. For details, see Understanding Template Arguments.) The new "BookList" class can thus contain 50 instances of the "Book" class.

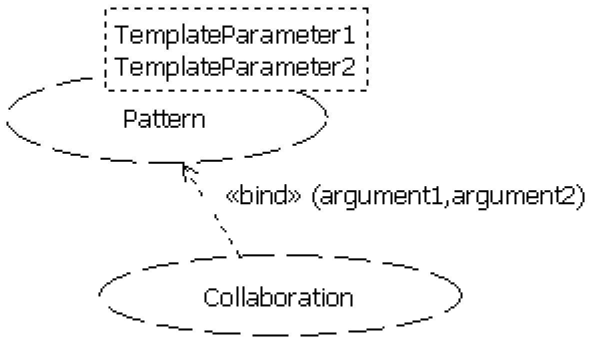
Connector

A binding relationship appears as a dashed line with an arrow pointing from the model element to the template. The keyword «bind» appears next to the connector, and template arguments are placed in parentheses after the keyword.

Binding Relationship Between a Template Class and Class



Binding Relationship Between a Pattern and Collaboration



Using Binding Relationships

You can add binding relationships to your model to create new classes from template classes or to create new collaborations from template collaborations. The binding relationship allows you to assign values to the template parameters in the template class or template collaboration. The values assigned to template parameters are called template arguments. A template argument must correspond in type to the template parameter. For example, you must assign a template argument with a class value to a template parameter defined as a class type.

In most cases, a binding relationship provides one value for each template parameter. There are two exceptions: First, if a template parameter has been assigned a default template argument, you do not have to provide a template argument for it if you want the new model element to be defined by replacing the template parameter with its default. Second, some template collaborations have template parameters to which you can assign template arguments with multiple values. For instance, you can bind multiple classes to the observer template parameter in the observer pattern. For details, see Example Pattern and Understanding Template Arguments.

The binding relationships in a model usually appear in class diagrams.

Naming Conventions

Binding relationships typically do not have names.

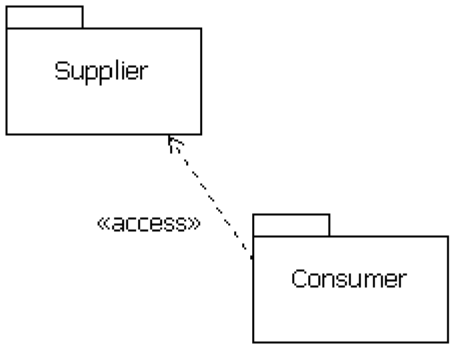
Permission Relationships

A permission relationship is a type of dependency relationship that gives one model element access to the contents of another model element. The model element that gains permission is the consumer in the relationship, and the model element that grants permission is the supplier. Permission relationships include access, friend, and import relationships.

For example, a model may be separated into layers to group classes or subsystems that are at the same levels of abstraction. The architect places each layer in a package, so the model elements specific to the application (for example, the classes that implement the user interface) are in one package, and the model elements specific to the type of business (for example, a subsystem that is used to process credit card transactions) are in another package. An access relationship pointing from the application package to the business package indicates that one or more model elements in the application package can access the model elements in the business package.

Connector

A permission relationship appears as a dashed line with an open arrow. It points from the consumer model element to the supplier model element. A stereotype appears next to the connector to indicate the type of permission.



Types of Permission Relationships

The type of access that the consumer model element has to the contents of the supplier model element is defined by a stereotype. The following table identifies the predefined types of permission relationships.

Name	**Keyword**	**Description**
Access	«access»	Indicates that the consumer model element can access the public contents of the supplier model element. This type of permission relationship typically connects packages.
Friend	«friend»	Indicates that the consumer model element has access to the contents of the supplier model element regardless of the content's visibility. This type of permission relationship typically connects either operations or classes to other classes.
Import	«import»	Indicates that the consumer model element adds the public contents of the supplier model element to its namespace. This type of permission relationship typically connects packages.

Naming Conventions

Permission relationships typically do not have names.

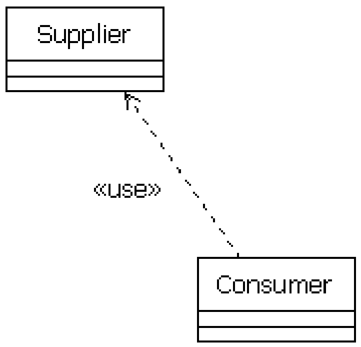
Usage Relationships

A usage relationship is a type of dependency relationship in which one model element requires the presence of another model element (or set of model elements) for its full implementation or operation. The model element that requires the presence of another model element is the consumer in the relationship, and the model element whose presence is required is the supplier.

Although a usage relationship indicates an ongoing need, it also indicates that the connection between the two model elements is not always meaningful or present.

Connector

A usage relationship appears as a dashed line with an open arrow. It points from the consumer model element to the supplier model element. The «use» keyword appears next to the connector.



Naming Conventions

Usage relationships typically do not have names.

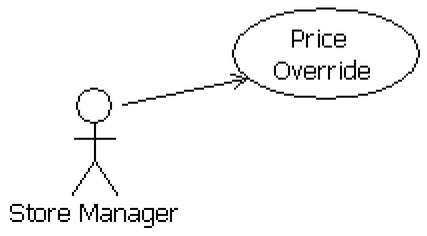
Directed Association Relationships

A directed association is an association relationship that is navigable in only one direction, indicating that the control flows from one classifier to another (for example, from an actor to a use case). This means that only one of the association ends has navigability set for it. For details, see Specifying Navigability in Associations.

An example directed association can be found in an e-commerce application. A "Store Manager" actor connects to a target classifier "Price Override" with a directed association. Control can only flow in one direction; only the store manager can override the price of an item.

Connector

A directed association appears as a solid line with an arrowhead near the shape to which navigation flows.



Naming Conventions

You can name any association to describe the nature of the relationship between the two classifiers; however, names are typically not needed, especially if you use association end names.

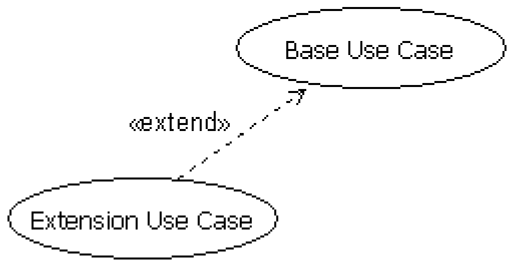
Extend Relationships

An extend relationship specifies that the behaviors defined in one use case (extension use case) can be inserted into another use case (base use case). The extend relationship signifies that the incorporation of the extension use case is conditional, meaning its execution is dependent on what has happened while executing the base use case. You can define the exact location where the extension use case is inserted in the documentation for the base use case.

For example, an e-commerce system may have a base use case called "Place Online Order" that has an extension use case called "Specify Shipping Instructions." An extend relationship points from the "Specify Shipping Instructions" use case to the "Place Online Order" use case to indicate that the behaviors in the "Specify Shipping Instructions" use case are optional and only occur in certain circumstances.

Connector

An extend relationship appears as a dashed line with an arrow pointing from the extension use case to the base use case. The keyword «extend» is attached to the connector.



Using Extend Relationships

You can add extend relationships to your model to indicate the following:

- A part of a use case that is optional system behavior
- A subflow that is executed only under certain conditions

- A set of behavior segments, of which one or more may be inserted in a base use case

The extend relationships in a model usually appear in use-case diagrams.

Naming Conventions

Extend relationships are not named.

Imports (Java)

An Import declaration, in a Java source file, maps to an Access relationship in a Java code model. It displays on a model diagram as a Dependency relationship with an «Access» stereotype

Generalization Relationships

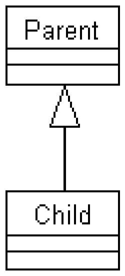
A generalization relationship (sometimes called an is-a relationship) indicates that a specialized (child) model element is based on a general (parent) model element. The parent model element can have one or more children, and any child model element can have one or more parents. It is more common to have a single parent and multiple children. To comply with UML semantics, the related model elements must be the same type; a generalization relationship can be created from actor to actor or from use case to use case; however, it cannot be created from actor to use case.

For example, in an e-commerce application for a site that sells a variety of merchandise, an "inventory" class may be a parent class (also called a superclass). This class contains the attributes, such as "price," and operations, such as "setPrice," that all pieces of merchandise use. After defining this parent class, a child class (also called a subclass) is created for each type of merchandise, such as books and VCRs. The "book" class uses the attributes and operations in the "inventory" class and then adds attributes such as "author" and operations such as "setAuthor." A VCR class also uses the attributes and operations in the "inventory" class, but it adds attributes such as "manufacturer" and operations such as "setManufacturer," which are different from those in the book class.

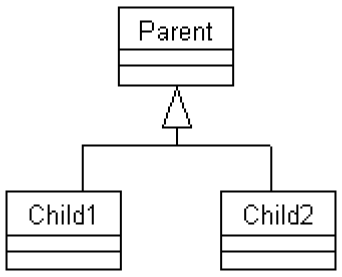
Connector

A generalization appears as a solid line with an unfilled arrowhead pointing from the specialized (child) model element to the general (parent) model element.

Single Parent and Single Child



Single Parent with Multiple Children (Inheritance Tree)



Using Generalization Relationships

You can add generalization relationships to your model to capture attributes, operations, and relationships in a parent model element and then reuse them in one or more child model elements. Because the child model elements in generalizations inherit the attributes, operations, and relationships of the parent, you need to define for the child only those attributes, operations, or relationships that are distinct from its parent.

The generalization relationships in a model appear in a variety of diagrams, including class, component, and use-case diagrams.

Naming Conventions

Generalization relationships usually do not have names. If you name a generalization relationship, the name appears next to the generalization connector on the diagram.

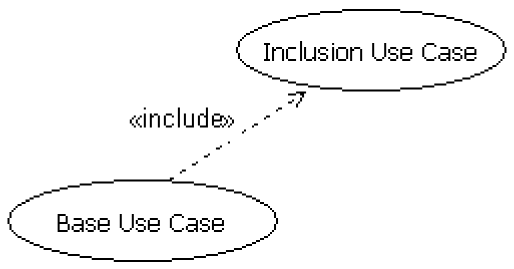
Include Relationships

An include relationship specifies that a base use case uses the behavior defined in another use case (inclusion use case). The inclusion relationship signifies that the incorporation of the inclusion use case is unconditional, meaning the inclusion use case is always executed when the spot in the base use case where it is inserted is reached. The inclusion use case is abstract.

For example, an e-commerce application may provide customers with the option of checking the status of their orders. One way to model this behavior is with a base use case called "Check Order Status" that has an inclusion use case called "Log In." The "Log In" use case is a separate inclusion use case because it contains behaviors that are used by a number of other use cases in the system. In the use-case diagram, an include relationship points from the "Check Order Status" use case to the "Log In" use case to indicate that the behaviors in the "Log In" use case are always included in the "Check Order Status" use case.

Connector

An include relationship appears as a dashed line with an arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



Using Include Relationships

You can add include relationships to your model to show the following:

- The behavior of the inclusion use case is common to two or more use cases.

- The result of the behavior specified in the inclusion use case, not the behavior itself, is important in the base use case.

The extend relationships in a model usually appear in use-case diagrams.

Naming Conventions

Include relationships are not named.