Structuring the Use-Case Model

Maria Ericsson
Email: mariae@rational.com

Introduction

The purpose of this white paper is to summarize and exemplify how use-case relationships will be defined in the UML 1.3. It is an excerpt of what will be in the Rational Unified Process 5.0. It is assumed that the reader is familiar with the basics of use cases.

Why Structure the Use-Case Model?

There are three main reasons for structuring the use-case model:

- To make the use cases easier to understand.
- To reuse behavior that is shared among many use cases.
- To make the use-case model easier to maintain.

Structuring is, however, not the first you do. There is no point in structuring the use cases until you know a bit more about their behavior, beyond a one sentence brief description. You should at least have established a step-by-step outline to the flow of events of the use case, to make sure that you decisions are based on an accurate enough understanding of the behavior.

To structure the use cases, we have three kinds of relationships. You will use these relationships to factor out pieces of use cases that can be reused in other use cases, or that are specializations or options to the use case. The use case that represents the modification we call the *addition use case*. The use case that is modified we call the *base use case*.

- If there is a part of a base use case that represents a function of which the use case only depends on the result, not the method used to produce the result, you can factor that part out to an addition use case. The addition is explicitly included in the base use case, using the include-relationship.
- If there is a part of a base use case that is optional, or not necessary to understand the primary purpose of the use case, you can factor that part out to an addition use case in order to simplify the structure of the base use case. The addition is implicitly included in the base use case, using the extend-relationship.
- If there are use cases that have commonalties in behavior and structure and that have similarities in purpose, their common parts can be factored out to a base use case (parent) that is inherited by addition use cases (children). The child use cases can insert new

behavior and modify existing behavior in the structure they inherit from the parent use case.

You can also use actor-generalization to show how actors are specializations of one another.

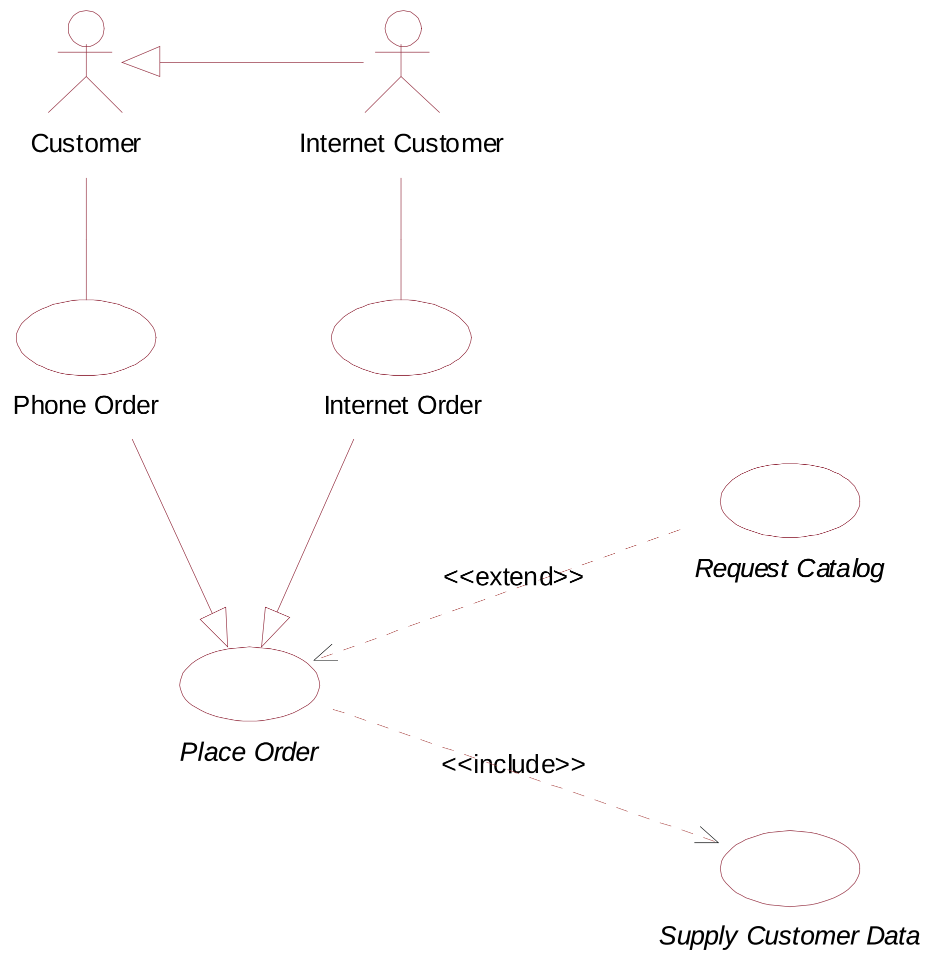
In factoring out behavior to new use cases you may create use cases that never are instantiated on their own, only as part of some other use case. Such non-instantiable use cases are referred to as *abstract* use cases. Use cases that are directly initiated by actors and instantiated on their own are called *concrete* use cases.

Example:

Consider part of the use-case model for an Order Management System.

It is useful to separate ordinary Customer from Internet Customer, since they have slightly different properties. However, since Internet Customer does exhibit all properties of a Customer, you can say that Internet Customer is a specialization of Customer, indicated with an actor-generalization.

The concrete use cases in this diagram are Phone Order (initiated by the Customer actor) and Internet Order (initiated by Internet Customer). These use cases are both variations of the more general Place Order use case, which in this example is abstract. The Request Catalog use case represents an optional segment of behavior that is not part of the primary purpose of Place Order. It has been factored out to an abstract use case to simplify the Place Order use case. The Supply Customer Data use case represents a segment of behavior that was factored out since it is a separate function of which only the result is affecting the Place Order use case and it can also be reused in other use cases. Both Request Catalog and Supply Customer Data are abstract in this example.



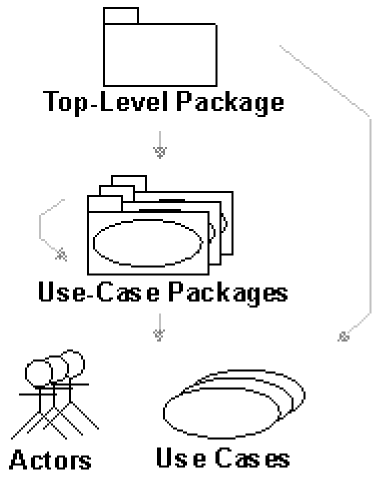
A use-case diagram showing part of the use-case model for an Order Management System.

The following table shows a more detailed comparison between the three different use-case relationships:

Question	**Extend**	**Include**	**Generalization**
What is the direction of the relationship?	The addition use case references the base use case.	The base use case references the addition use case.	The addition use case (child) references the base use case (parent).
Does the relationship have multiplicity?	Yes, on the addition side.	No. If you want to include the same segment of behavior more than once, that needs to be stated in the base use case.	No.
Does the relationship have a condition?	Yes.	No. If you want to express a condition on the inclusion you need to say it explicitly in the base use case.	No.

Is the addition use case abstract?	Often yes, but not necessarily.	Yes.	Often no, but it can be.
Is the base use case modified by the addition?	The extension implicitly modifies the behavior of the base use case.	The inclusion explicitly modifies the effect of the base use case.	If the base use case (parent) is instantiated, it is unaffected by the child. To obtain the effects of the addition, the addition use case (child) must be instantiated.
Does the base use case have to be complete and meaningful?	Yes.	Together with the additions, yes.	If it is abstract, no.
Does the addition use case have to be complete and meaningful?	No.	No.	Together with the base use case (parent), yes.
Can the addition use case access attributes of the base use case?	Yes.	No. The inclusion is encapsulated, and only "sees" itself.	Yes, by the normal mechanisms of inheritance.
Can the base use case access attributes of the addition use case?	No. The base use case must be well-formed in the absence of the addition.	No. The base use case only knows about the effect of the addition. The addition is encapsulated.	No. The base use case (parent) must in this sense be well-formed in the absence of the addition (child).

Another aspect of organizing the use-case model for easier understanding is to group the use cases into packages. The use-case model can be organized as a hierarchy of use-case packages, with "leaves" that are actors or use cases.



The use-case model hierarchy. Arrows show possible ownership.

Include-Relationship

An include-relationship is a relationship from a base use case to an inclusion use case, specifying how the behavior defined for the inclusion use case is explicitly inserted into the behavior defined for the base use case.

Notation: a dashed arrow with the text «include».

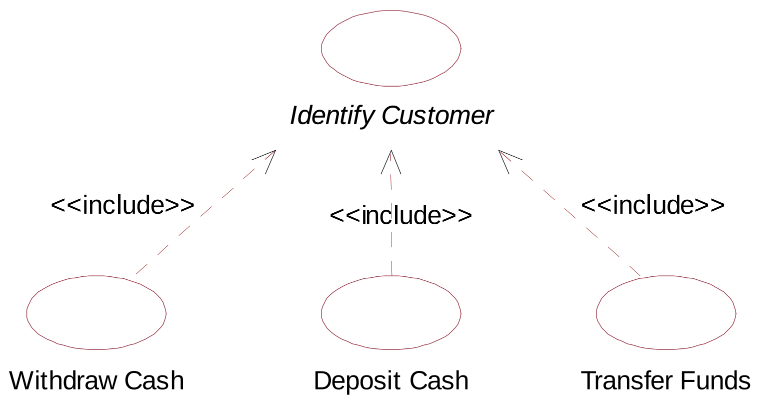
The inclusion use case is always abstract. The base use case has control of the relationship to the inclusion and can depend on the result of performing the inclusion, but neither the base nor the inclusion may access each other's attributes. The inclusion is in this sense encapsulated, and represents behavior that can be reused in different base use cases.

You can use the include-relationship to:

- Factor out behavior from the base use case that is not necessary for the understanding of the primary purpose of the use case, only the result of it is important.
- Factor out behavior that is in common for two or more use cases.

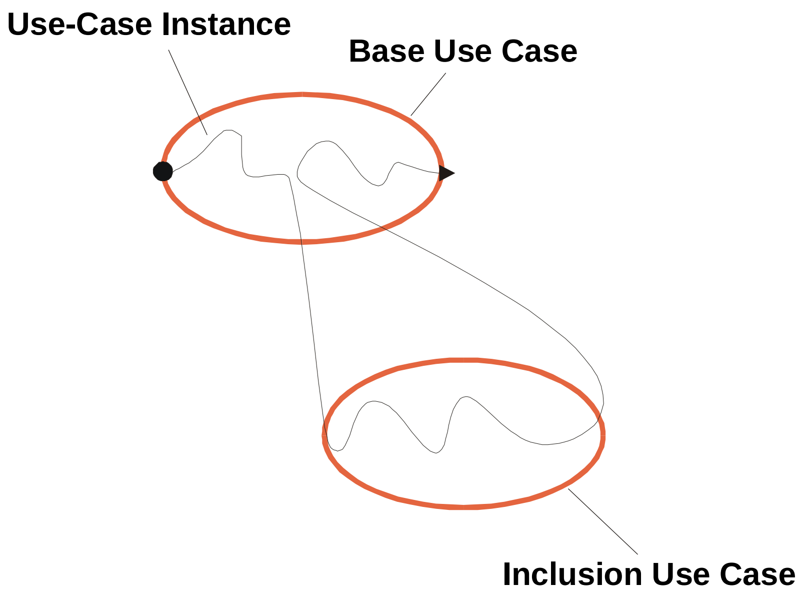
Example:

In an ATM system, the use cases Withdraw Cash, Deposit Cash, and Transfer Funds all need to include how the customer is identified to the system. This behavior can be extracted to a new inclusion use case called Identify Customer, which the three base use cases include. The base use cases are independent of the method used for identification, and it is therefore encapsulated in the inclusion use case. From the perspective of the base use cases, it does not matter whether the method for identification is to read a magnetic bank card, or to do a retinal scan. They only depend on the result of Identify Customer, which is the identity of the customer. And vice versa, from the perspective of the Identify Customer use case, it does not matter how the base use cases use the customer identity or what has happened in them before the inclusion is executed, the method for identification is still exactly the same.



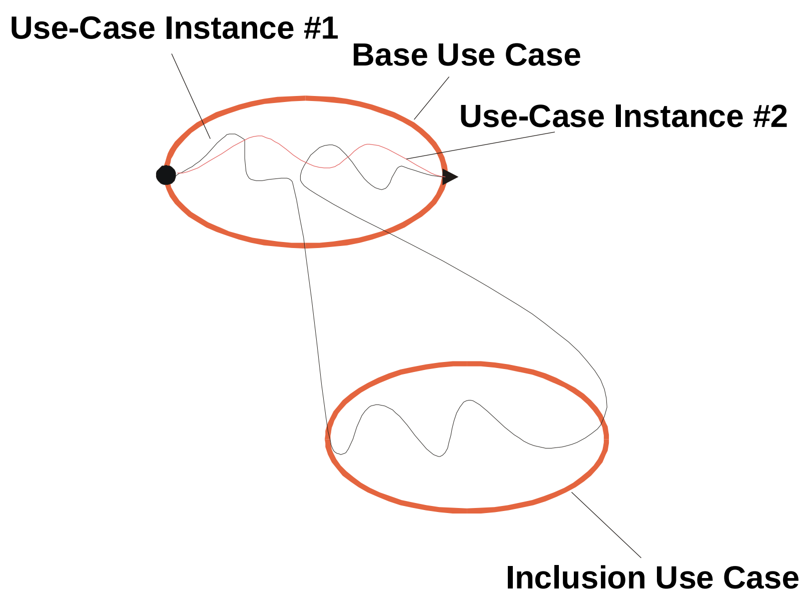
In the ATM system, the use cases Withdraw Cash, Deposit Cash, and Transfer Funds all include the use case Identify Customer.

The behavior of the inclusion is inserted in one location in the base use case. When a use-case instance following the description of a base use case reaches a location in the base use case from which include-relationship is defined, it will follow the description of the inclusion use case instead. Once the inclusion is performed, the use-case instance will resume where it left off in the base use case.



A use-case instance following the description of a base use case including its inclusion.

The include-relationship is not conditional, if the use-case instance reaches the location in the base use case it is defined for, it is always executed. If you want to express a condition, you need to do that as part of the base use case. If the use-case instance never reaches the location the include-relationship is defined for, it will not be executed.



Use-case instance #1 follows the base use case and its inclusion. Use-case instance #2 never reaches the point the inclusion is defined for, and follows only the base use case.

The inclusion use case is one continuous segment of behavior, all of which is included at one location in the base use case. If you have separate segments of behavior that need to be inserted at different locations, you should consider an extend-relationship or use-case-generalization instead.

Extend-Relationship

An extend-relationship is a relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case can be inserted into the behavior defined for the base use case. It is implicitly inserted in the sense that the extension is not shown in the base use case.

Notation: a dashed arrow with the text «extend».

You define where in the base to insert the extension by referring to extension points in the base use case. The extension use case is often abstract, but does not have to be. The extension is conditional. The base use case does not control the conditions for the execution of the extension, those conditions are described within the extend-relationship.

An extension point opens up a base use case to the possibility of an extension. It has a name, and a list of references to one or more locations within the flow of events of the use case. An extension point may reference a single location between two behavior steps within the base use case. It may also reference a set of discrete locations.

You can use the extensions for several purposes:

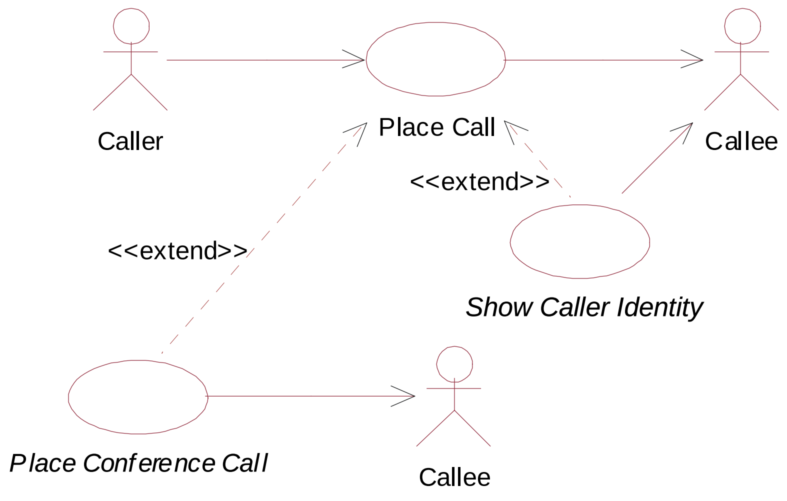
- To show that a part of a base use case is optional, or potentially optional, system behavior. In this way, you separate optional behavior from mandatory behavior in your model.
- To show that a subflow is executed only under certain (sometimes exceptional) conditions, such as triggering an alarm.

- To show that there may be a set of behavior segments of which one or several may be inserted at an extension point in a base use case. It will depend on the interaction with the actors during the execution of the base use case which of the behavior segments are inserted and in what order.

The base use case is implicitly modified by the extensions. You can also say that the base use case defines a modular framework into which extensions can be added, but the base does not have any visibility of the specific extensions.

The base use case should be complete in and of itself, meaning that it should be understandable and meaningful without any references to the extensions.

Example:

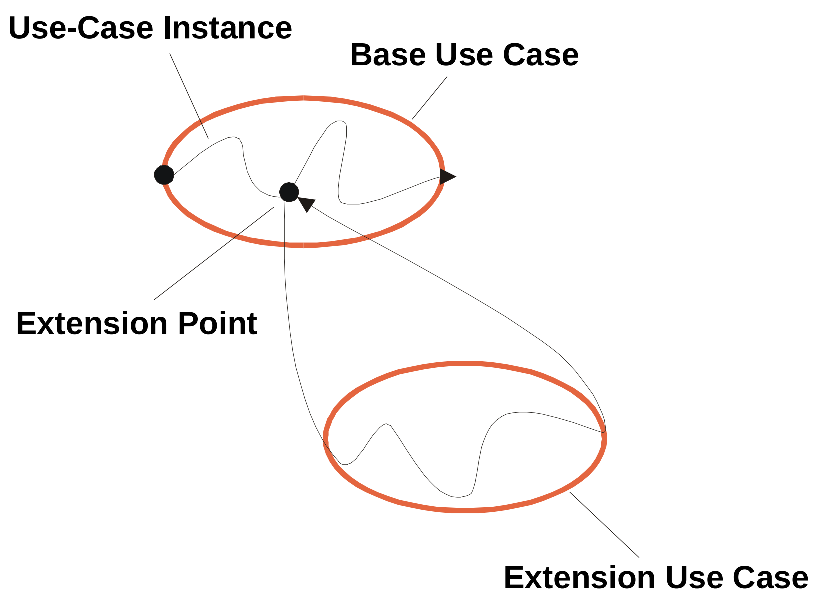


The use cases Place Conference Call and Show Caller Identity are both extensions to the base use case Place Call.

In a phone system, the primary service provided to the users is represented by the use case Place Call. Examples of optional services are to be able to add a third party to a call (Place Conference Call) and to allow the callee to see the identity of the caller (Show Caller Identity). We can represent the behaviors needed for these optional services as extension use cases to the base use case Place Call. This is a correct use of the extend-relationship, since Place Call is meaningful in itself, you do not need to read the descriptions of the extension use cases to understand the primary purpose of the base use case, and the extensions use cases have optional character.

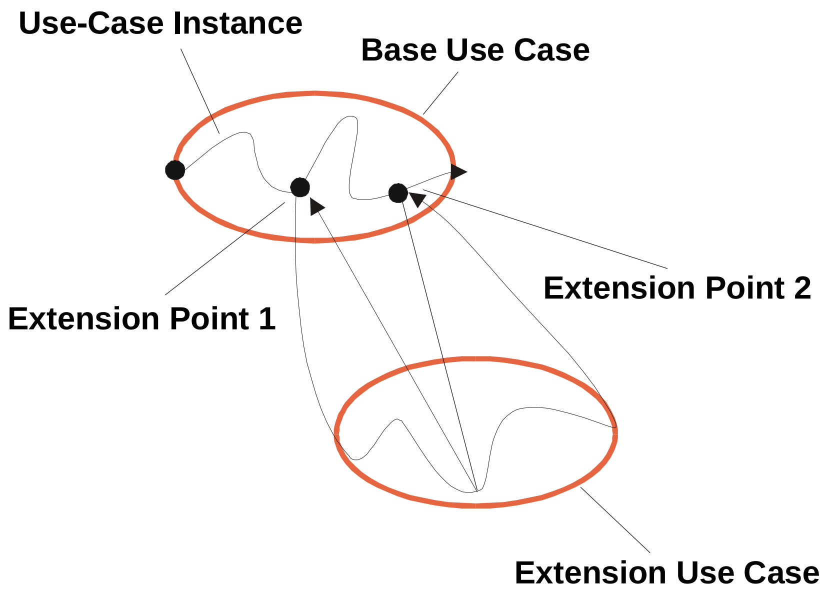
If both the base use case and the "base plus extension" use case must be explicitly instantiable, or if you want the addition to modify behavior in the base use case, you should use use-case-generalization instead.

When a use-case instance performing the base use case reaches a location in the base use case that has an extension point defined for it, the condition on the corresponding extend-relationship is evaluated. If the condition is true or if it is absent, the use-case instance will follow the extension. If the condition of the extend-relationship is false, the extension is not executed. Once the use-case instance has performed the extension, the use-case instance resumes executing the base use case at the point where it left off.



A use-case instance following a base use case and its extension.

An extension use case can have more than one insertion segment, each related to its own extension point in the base use case. If this is the case, the use-case instance will resume the base use case and continue to the next extension point specified in the extend-relationship. At that point it will execute the next insertion segment of the extension use case. This is repeated until the last insertion segment has been executed. Note that the condition for the extend-relationship is checked at the first extension point only, once started all insertion segments must be performed.



A use-case instance following a base use case and an extension use case, the latter with two insertion segments.

Use-Case-Generalization

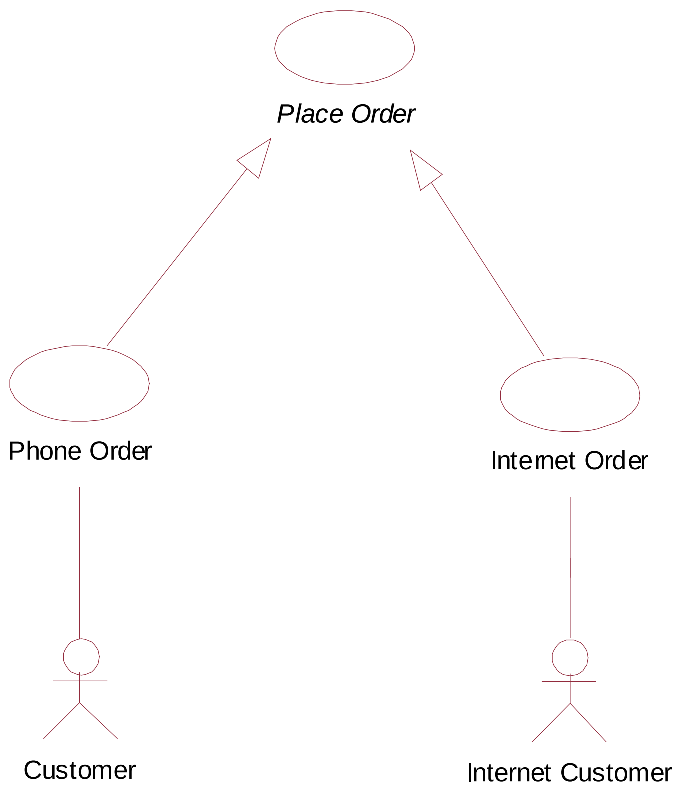
A use-case-generalization is a relationship from a child use case to a parent use case, specifying how a child can specialize all behavior and characteristics described for the parent.

Notation: a generalization arrow.

Neither parent nor child is necessarily abstract, although the parent in most cases is abstract. A child inherits all structure, behavior, and relationships of the parent. This is generalization as applicable to use cases.

Generalization is used when you find two or more use cases that have commonalities in behavior, structure, and purpose. In such case you can describe the shared parts in a new use case, that then is specialized by child use cases.

Example:



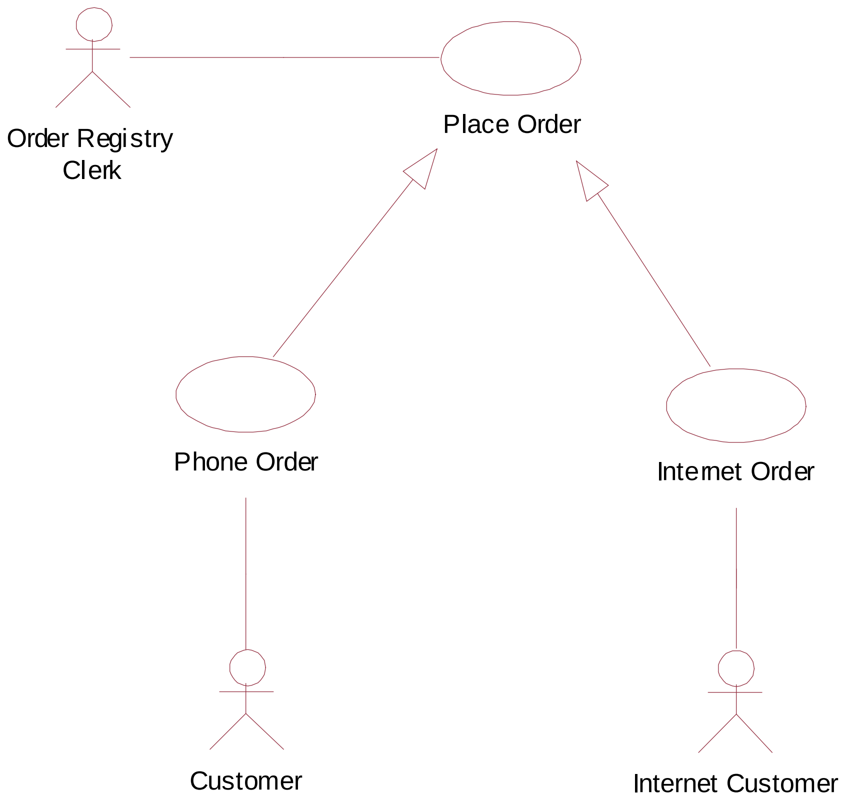
The use cases Phone Order and Internet Order are specializations of the abstract use case Place Order.

In an Order Management system, the use cases Phone Order and Internet Order share a lot in structure and behavior. A general use case Place Order is defined where that structure and common behavior is defined. The abstract use case Place Order need not be complete in itself, but provides a general behavioral framework which the child use cases can make complete.

The parent use case is not always abstract.

Example:

Consider the Order Management system in the previous example. Say that we want to add an Order Registry Clerk actor, who can enter orders into the system on behalf of a customer. This actor would initiate the general Place Order use case, which now must have a complete flow of events described for it. The child use cases can add behavior to the structure the parent use case provides, and also modify behavior in the parent.

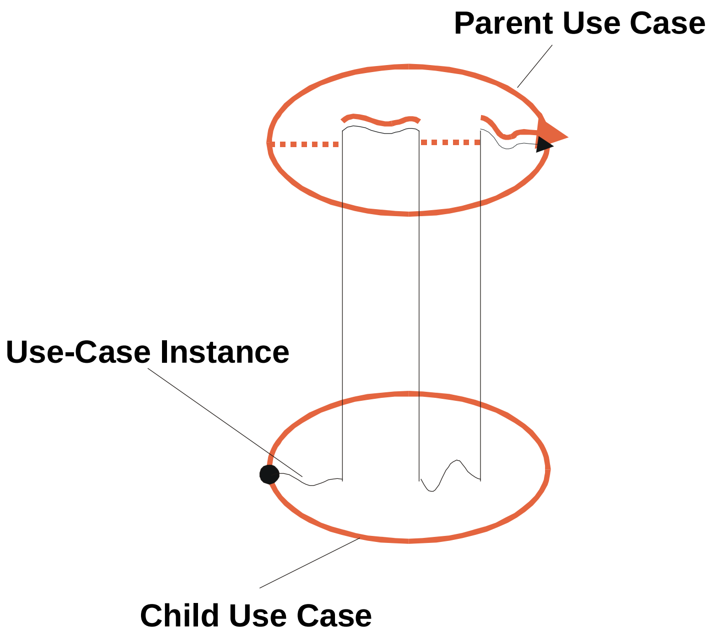


The actor Order Registry Clerk can instantiate the general use case Place Order. Place Order can also be specialized by the use cases Phone Order or Internet Order.

If two child use cases are specializing the same parent (or base), the specializations are independent of one another, meaning they are executed in separate use-case instances. This is unlike the extend- or include-relationships, where several additions implicitly or explicitly modify one use-case instance executing the same base use case.

Both use-case-generalization and include can be used to reuse behavior among use cases in the model. The difference is that with use-case generalization, the execution of the children are dependent on the structure and behavior of the parent (the reused part), while in an include-relationship the execution of the base use case only depends on the result of the function the inclusion use case (the reused part) performs. Another difference is that in a generalization the children share similarities in purpose and structure, while in the include-relationship the base use cases reusing the same inclusion can have completely different purposes, they just need the same function to be performed.

A use-case instance executing a child use case will follow the flow of events described for the parent use case, inserting additional behavior and modifying behavior as defined in the flow of events of the child use case.



The use-case instance follows the parent use case, with behavior inserted or modified as described in the child use case.

[This page intentionally left blank]