16   Use Cases

16.1   Overview

Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do. The key concepts associated with use cases are *actors*, *use cases,* and the *subject*. The subject is the system under consideration to which the use cases apply. The users and any other systems that may interact with the subject are represented as actors. Actors always model entities that are outside the system. The required behavior of the subject is specified by one or more use cases, which are defined according to the needs of actors.

Strictly speaking, the term "use case" refers to a use case type. An instance of a use case refers to an occurrence of the emergent behavior that conforms to the corresponding use case type. Such instances are often described by interaction specifications.

Use cases, actors, and systems are described using use case diagrams.
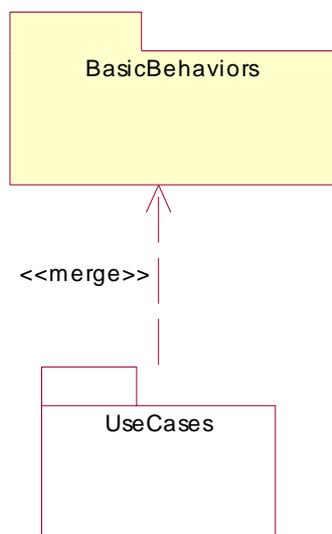
16.2   Abstract syntax

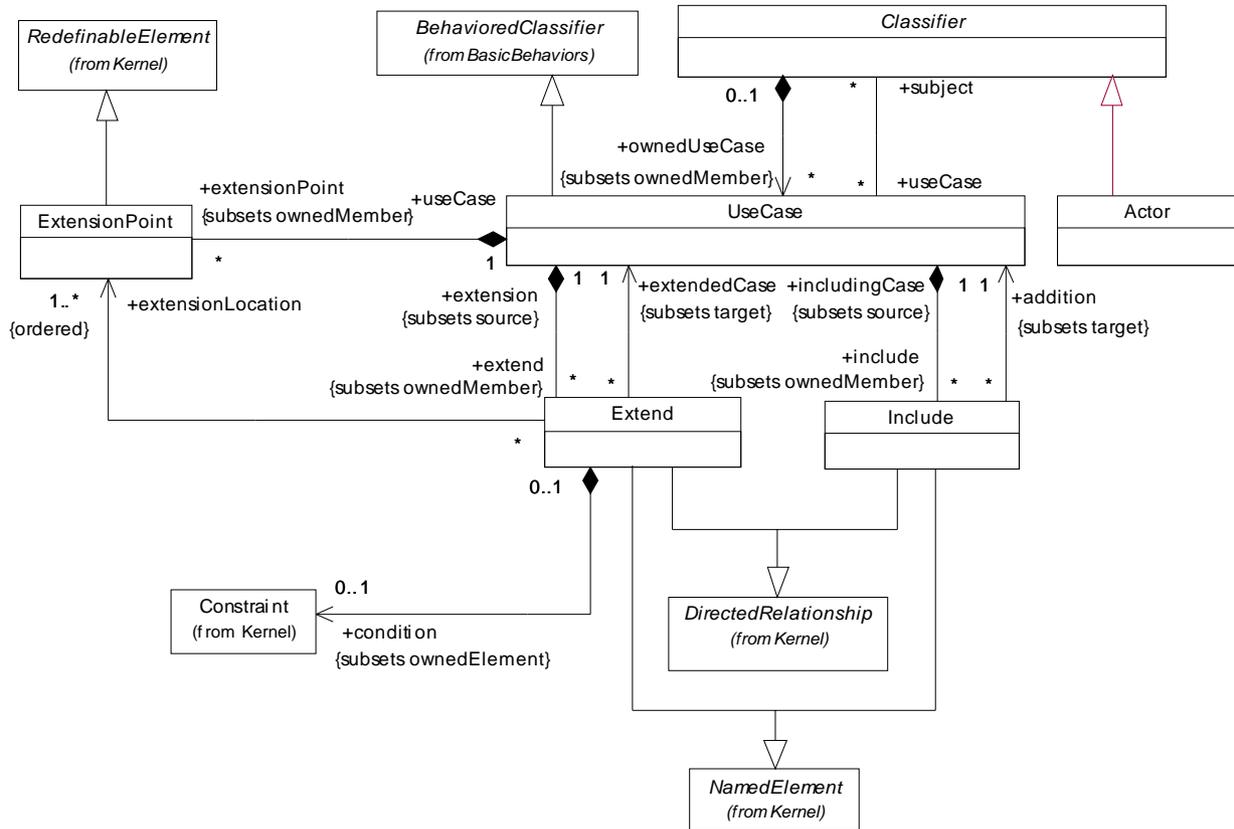**Figure 16.1 - Dependencies of the UseCases package**

Figure 16.2 - The concepts used for modeling use cases

16.3 Class Descriptions

16.3.1 Actor (from UseCases)

An actor specifies a role played by a user or any other system that interacts with the subject. (The term "role" is used informally here and does not necessarily imply the technical definition of that term found elsewhere in this specification.)

Generalizations

- "Classifier (from UseCases)" on page 572

Description

An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is *external* to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., "role") of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances.

Since an actor is external to the subject, it is typically defined in the same classifier or package that incorporates the subject classifier.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An actor can only have associations to use cases, components, and classes. Furthermore these associations must be binary.

```
self.ownedAttribute->forAll ( a |
    (a.association->notEmpty()) implies
        ((a.association.memberEnd.size() = 2) and
         (a.opposite.class.oclIsKindOf(UseCase) or
         (a.opposite.class.oclIsKindOf(Class) and not a.opposite.class.oclIsKindOf(Behavior))))
```

[2] An actor must have a name.

```
name->notEmpty()
```

Semantics

Actors model entities external to the subject. When an external entity interacts with the subject, it plays the role of a specific actor.

When an actor has an association to a use case with a multiplicity that is greater than one at the use case end, it means that a given actor can be involved in multiple use cases of that type. The specific nature of this multiple involvement depends on the case on hand and is not defined in this specification. Thus, an actor may initiate multiple use cases in parallel (concurrently) or they may be mutually exclusive in time. For example, a computer user may activate a given software application multiple times concurrently or at different points in time.
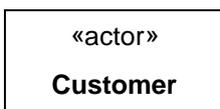
Notation

An actor is represented by "stick man" icon with the name of the actor in the vicinity (usually above or below) the icon.

Customer

Presentation Options

An actor may also be shown as a class rectangle with the keyword «actor», with the usual notation for all compartments.

«actor»

Customer

Other icons that convey the kind of actor may also be used to denote an actor, such as using a separate icon for non-human actors.

User

Style Guidelines

Actor names should follow the capitalization and punctuation guidelines used for classes in the model. The names of abstract actors should be shown in italics.

Changes from previous UML

There are no changes to the Actor concept except for the addition of a constraint that requires that all actors must have names.

16.3.2 Classifier (from UseCases)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 48 *(merge increment)*

Description

Extends a classifier with the capability to own use cases. Although the owning classifier typically represents the subject to which the owned use cases apply, this is not necessarily the case. In principle, the same use case can be applied to multiple subjects, as identified by the *subject* association role of a UseCase (see "UseCase (from UseCases)" on page 578).

Attributes

No additional attributes

Associations

- ownedUseCase: UseCase[*] References the use cases owned by this classifier. (Subsets *Namespace.ownedMember*)

- useCase : UseCase [*] The set of use cases for which this Classifier is the subject.

Constraints

No additional constraints

Semantics

See "UseCase (from UseCases)" on page 578.

Notation

The nesting (owning) of a use case by a classifier is represented using the standard notation for nested classifiers.

Rationale

This extension to the Classifier concept was added to allow classifiers in general to own use cases.

Changes from previous UML

No changes

16.3.3   Extend (from UseCases)

A relationship from an extending use case to an extended use case that specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the extended use case.

Generalizations

- "DirectedRelationship (from Kernel)" on page 59

Description

This relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions.

Note that the same extending use case can extend more than one use case. Furthermore, an extending use case may itself be extended.

It is a kind of DirectedRelationship, such that the source is the extending use case and the destination is the extended use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The extend relationship itself is owned by the extending use case.

Attributes

No additional attributes

Associations

- extendedCase : UseCase [1]   References the use case that is being extended.
                                 (Specializes *DirectedRelationship.target*)

- extension : UseCase [1] References the use case that represents the extension and owns the extend relationship. (Specializes *DirectedRelationship.source*)

- condition : Constraint [0..1] References the condition that must hold when the first extension point is reached for the extension to take place. If no constraint is associated with the extend relationship, the extension is unconditional. (Specializes *Element.ownedElement*)

- extensionLocation: ExtensionPoint [1..*] An ordered list of extension points belonging to the extended use case, specifying where the respective behavioral fragments of the extending use case are to be inserted. The first fragment in the extending use case is associated with the first extension point in the list, the second fragment with the second point, and so on. (Note that, in most practical cases, the extending use case has just a single behavior fragment, so that the list of extension points is trivial.)

Constraints

[1] The extension points referenced by the extend relationship must belong to the use case that is being extended.

extensionLocation->forAll (xp | extendedCase.extensionPoint->includes(xp))

Semantics

The concept of an "extension location" is intentionally left underspecified because use cases are typically specified in various idiosyncratic formats such as natural language, tables, trees, etc. Therefore, it is not easy to capture its structure accurately or generally by a formal model. The intuition behind the notion of extension location is best explained through the example of a textually described use case: Usually, a use case with extension points consists of a set of finer-grained behavioral fragment descriptions, which are most often executed in sequence. This segmented structuring of the use case text allows the original behavioral description to be extended by merging in supplementary behavioral fragment descriptions at the appropriate insertion points between the original fragments (extension points). Thus, an extending use case typically consists of one or more behavior fragment descriptions that are to be inserted into the appropriate spots of the extended use case. An extension location, therefore, is a specification of all the various (extension) points in a use case where supplementary behavioral increments can be merged.

If the condition of the extension is true at the time the first extension point is reached during the execution of the extended use case, then all of the appropriate behavior fragments of the extending use case will also be executed. If the condition is false, the extension does not occur. The individual fragments are executed as the corresponding extension points of the extending use case are reached. Once a given fragment is completed, execution continues with the behavior of the extended use case following the extension point. Note that even though there are multiple use cases involved, there is just a single behavior execution.

Notation

An extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend». The condition of the relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.(See Figure 16.3.)
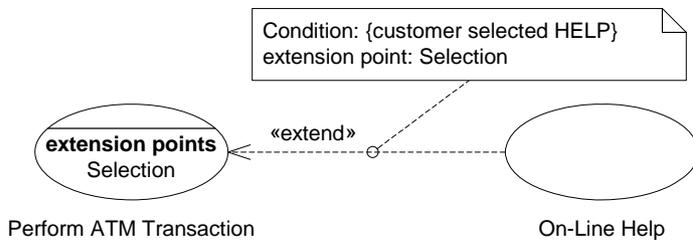
Examples

Figure 16.3 - Example of an extend relationship between use cases

In the use case diagram above, the use case "Perform ATM Transaction" has an extension point "Selection." This use case is extended via that extension point by the use case "On-Line Help" whenever execution of the "Perform ATM Transaction" use case occurrence is at the location referenced by the "Selection" extension point and the customer selects the HELP key. Note that the "Perform ATM Transaction" use case is defined independently of the "On-Line Help" use case.

Rationale

This relationship is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in another use case (which is meaningful independently of the extending use case).

Changes from previous UML

The notation for conditions has been changed such that the condition and the referenced extension points may now be included in a Note attached to the extend relationship, instead of merely being a textual comment that is located in the vicinity of the relationship.

16.3.4 ExtensionPoint (from UseCases)

An extension point identifies a point in the behavior of a use case where that behavior can be extended by the behavior of some other (extending) use case, as specified by an extend relationship.

Generalizations

- "RedefinableElement (from Kernel)" on page 125

Description

An ExtensionPoint is a feature of a use case that identifies a point where the behavior of a use case can be augmented with elements of another (extending) use case.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An ExtensionPoint must have a name.

    self.name->notEmpty ()

Semantics

An extension point is a reference to a location within a use case at which parts of the behavior of other use cases may be inserted. Each extension point has a unique name within a use case.

Semantic Variation Points

The specific manner in which the location of an extension point is defined is left as a semantic variation point.

Notation

Extension points are indicated by a text string within in the use case oval symbol or use case rectangle according to the syntax below:

        <extension point> ::= <name> [: <explanation>]

Note that *explanation,* which is optional, may be any informal text or a more precise definition of the location in the behavior of the use case where the extension point occurs.

Examples

See Figure 16.3 on page 575 and Figure 16.9 on page 582.

Rationale

ExtensionPoint supports the use case extension mechanism (see "Extend (from UseCases)" on page 573).

Changes from previous UML

In 1.x, ExtensionPoint was modeled as a kind of ModelElement, which due to a multiplicity constraint, was always associated with a specific use case. This relationship is now modeled as an owned feature of a use case. Semantically, this is equivalent and the change will not be visible to users.

ExtensionPoints in 1.x had an attribute called *location*, which was a kind of LocationReference. Since the latter had no specific semantics it was relegated to a semantic variation point. When converting to UML 2.0, models in which ExtensionPoints had a *location* attribute defined, the contents of the attribute should be included in a note attached to the ExtensionPoint.

16.3.5 Include (from UseCases)

An include relationship defines that a use case contains the behavior defined in another use case.

Generalizations

- "DirectedRelationship (from Kernel)" on page 59

Description

Include is a DirectedRelationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case.

Note that the included use case is not optional, and is always required for the including use case to execute correctly.

Attributes

No additional attributes

Associations

- addition : UseCase [1]   References the use case that is to be included. (Specializes *DirectedRelationship.target*)

- including Case : UseCase [1]   References the use case that will include the addition and owns the include relationship. (Specializes *DirectedRelationship.source*)

Constraints

No additional constraints

Semantics

An include relationship between two use cases means that the behavior defined in the including use case is included in the behavior of the base use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is then extracted to a separate use case, to be included by all the base use cases having this part in common. Since the primary use of the include relationship is for reuse of common parts, what is left in a base use case is usually not complete in itself but dependent on the included parts to be meaningful. This is reflected in the direction of the relationship, indicating that the base use case depends on the addition but not vice versa.

Execution of the included use case is analogous to a subroutine call. All of the behavior of the included use case is executed at a single location in the included use case before execution of the including use case is resumed.

Notation

An include relationship between use cases is shown by a dashed arrow with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword «include». (See Figure 16.4.)

Examples

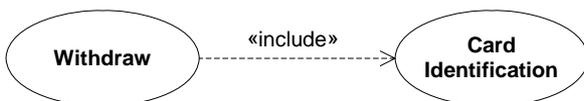A use case "Withdraw" includes an independently defined use case "Card Identification."

Figure 16.4 - Example of the Include relationship

Rationale

The Include relationship allows hierarchical composition of use cases as well as reuse of use cases.

Changes from previous UML

There are no changes to the semantics or notation of the Include relationship relative to UML 1.x.

16.3.6 UseCase (from UseCases)

A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

Generalizations

 • "BehavioredClassifier (from BasicBehaviors, Communications)" on page 419

Description

A UseCase is a kind of behaviored classifier that represents a declaration of an offered behavior. Each use case specifies some behavior, possibly including variants, that the subject can perform in collaboration with one or more actors. Use cases define the offered behavior of the subject without reference to its internal structure. These behaviors, involving interactions between the actor and the subject, may result in changes to the state of the subject and communications with its environment. A use case can include possible variations of its basic behavior, including exceptional behavior and error handling.

The subject of a use case could be a physical system or any other element that may have behavior, such as a component, subsystem, or class. Each use case specifies a unit of useful functionality that the subject provides to its users (i.e., a specific way of interacting with the subject). This functionality, which is initiated by an actor, must always be completed for the use case to complete. It is deemed complete if, after its execution, the subject will be in a state in which no further inputs or actions are expected and the use case can be initiated again or in an error state.

Use cases can be used both for specification of the (external) requirements on a subject and for the specification of the functionality offered by a subject. Moreover, the use cases also state the requirements the specified subject poses on its environment by defining how they should interact with the subject so that it will be able to perform its services.

The behavior of a use case can be described by a specification that is some kind of Behavior (through its ownedBehavior relationship), such as interactions, activities, and state machines, or by pre-conditions and post-conditions as well as by natural language text where appropriate. It may also be described indirectly through a Collaboration that uses the use case and its actors as the classifiers that type its parts. Which of these techniques to use depends on the nature of the use case behavior as well as on the intended reader. These descriptions can be combined. An example of a use case with an associated state machine description is shown in Figure 16.6.

Attributes

No additional attributes

Associations

 •   subject : Classifier[*]   References the subjects to which this use case applies. The subject or its parts realize all the use cases that apply to this subject. Use cases need not be attached to any specific subject, however. The subject may, but need not, own the use cases that apply to it.

- include : Include[*]   References the Include relationships owned by this use case.
(Specializes *Classifier.feature* and *Namespace.ownedMember*)

- extend : Extend[*]   References the Extend relationships owned by this use case.
(Specializes *Classifier.feature* and *Namespace.ownedMember*)

- extensionPoint: ExtensionPoint[*]   References the ExtensionPoints owned by the use case.
(Specializes *Namespace.ownedMember*)

Constraints

[1] A UseCase must have a name.

self.name -> notEmpty ()

[2] UseCases can only be involved in binary Associations.

[3] UseCases cannot have Associations to UseCases specifying the same subject.

[4] A use case cannot include use cases that directly or indirectly include it.

not self.allIncludedUseCases()->includes(self)

Additional Operations

[1] The query allIncludedUseCases() returns the transitive closure of all use cases (directly or indirectly) included by this use case.

UseCase::allIncludedUseCases() : Set(UseCase)
allIncludedUseCases = self.include->union(self.include->collect(in | in.allIncludedUseCases()))

Semantics

An execution of a use case is an occurrence of emergent behavior.

Every instance of a classifier realizing a use case must behave in the manner described by the use case.

Use cases may have associated actors, which describes how an instance of the classifier realizing the use case and a user playing one of the roles of the actor interact. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the subject. It is not possible to state anything about the internal behavior of the actor apart from its communications with the subject.

When a use case has an association to an actor with a multiplicity that is greater than one at the actor end, it means that more than one actor instance is involved in initiating the use case. The manner in which multiple actors participate in the use case depends on the specific situation on hand and is not defined in this specification. For instance, a particular use case might require simultaneous (concurrent) action by two separate actors (e.g., in launching a nuclear missile) or it might require complementary and successive actions by the actors (e.g., one actor starting something and the other one stopping it).

Notation

A use case is shown as an ellipse, either containing the name of the use case or with the name of the use case placed below the ellipse. An optional stereotype keyword may be placed above the name and a list of properties included below the name. If a subject (or system boundary) is displayed, the use case ellipse is visually located inside the system boundary rectangle. Note that this does not necessarily mean that the subject classifier owns the contained use cases, but merely that the use case applies to that classifier. For example, the use cases shown in Figure 16.5 on page 580 apply to the "ATMsystem" classifier but are owned by various packages as shown in Figure 16.7.
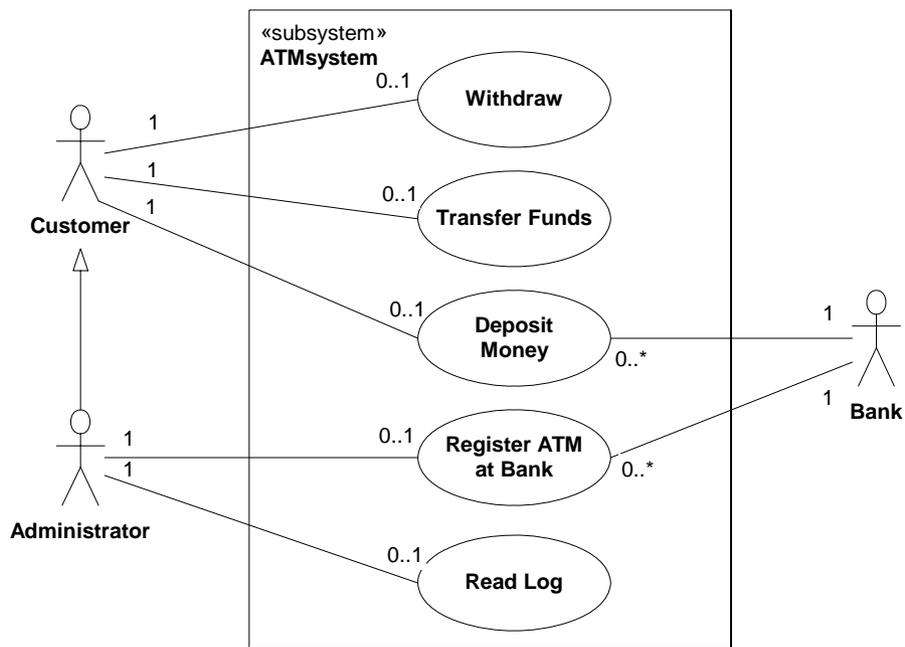
Figure 16.5 - Example of the use cases and actors for an ATM system
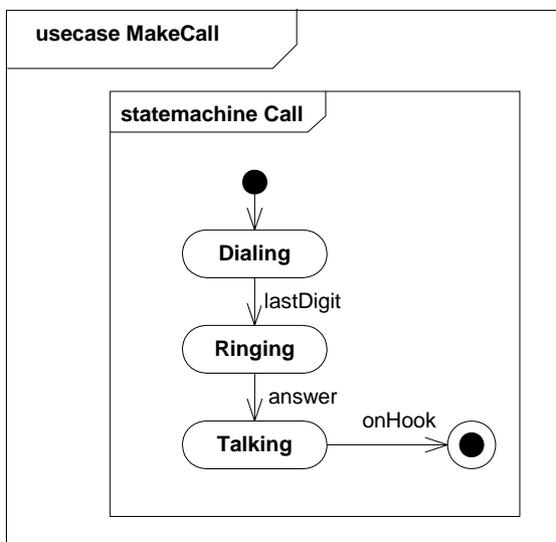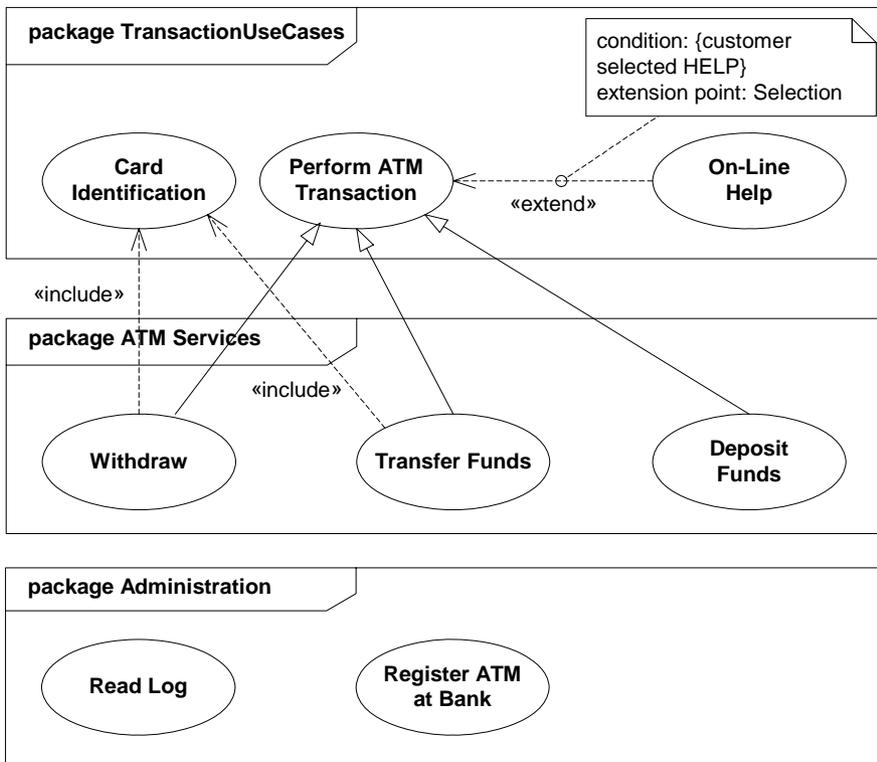

**Figure 16.6 - Example of a use case with an associated state machine behavior**

Figure 16.7 - Example of use cases owned by various packages

Extension points may be listed in a compartment of the use case with the heading **extension points**. The description of the locations of the extension point is given in a suitable form, usually as ordinary text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.

Use cases may have other associations and dependencies to other classifiers (e.g., to denote input/output, events, and behaviors).

The detailed behavior defined by a use case is notated according to the chosen description technique, in a separate diagram or textual document. Operations and attributes are shown in a compartment within the use case.

Use cases and actors may represent roles in collaborations as indicated in Figure 16.8.
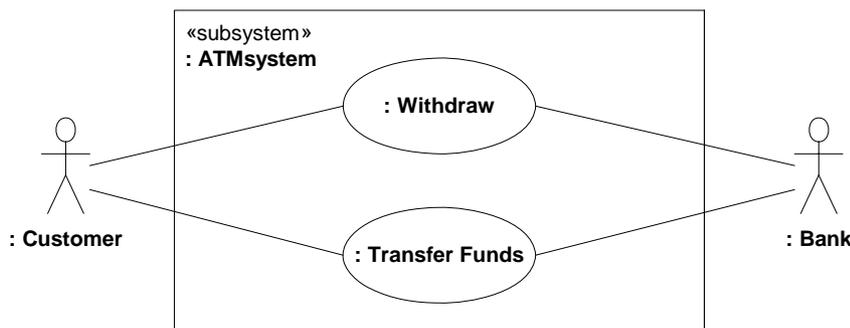
Figure 16.8 - Example of a use case for withdrawal and transfer of funds

Presentation Options

A use case can also be shown using the standard rectangle notation for classifiers with an ellipse icon in the upper-right-hand corner of the rectangle with optional separate list compartments for its features. This rendering is more suitable when there are a large number of extension points.

```
┌─────────────────────────────┐
│                      ◯       │
│   OrderStationery : PlaceOrder │
├─────────────────────────────┤
│  extension points            │
│                              │
│  order created : in Created state │
│  order processed : in Processed state │
│  order cancelled : in Cancelled state │
│  order rejected : in Rejected state │
│  order completed : in Completed state │
│  order destroyed : in Destroyed state │
│  order delivered : in Delivered state │
│  order obfuscated : in Obfuscated state │
└─────────────────────────────┘
```
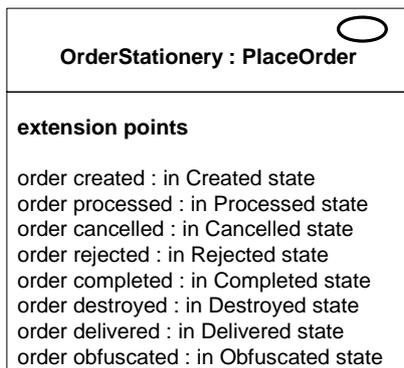
Figure 16.9 - Example of the classifier based notation for a use case

Examples

See Figure 16.3 through Figure 16.9.

Rationale

The purpose of use cases is to identify the required functionality of a system.

Changes from previous UML

The relationship between a use case and its subject has been made explicit. Also, it is now possible for use cases to be owned by classifiers in general and not just packages.

16.4   Diagrams

Description

Use Case Diagrams are a specialization of Class Diagrams such that the classifiers shown are restricted to being either Actors or Use Cases.
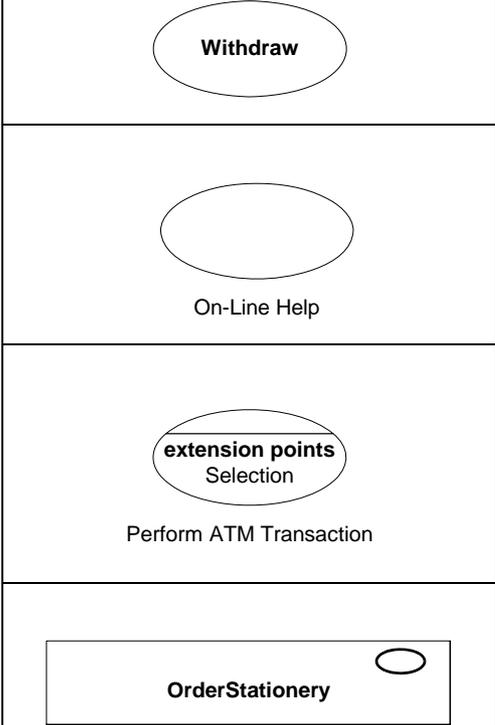
Graphic Nodes

The graphic nodes that can be included in structural diagrams are shown in Table 16.1.

Table 16.1 - Graphic nodes included in use case diagrams

Node Type	Notation	Reference
Actor (default)	 **Customer**	See "Actor (from UseCases)" on page 570.
Actor (optional user-defined icon - example)		
Extend		See "Extend (from UseCases)" on page 573.
Extend (with Condition)		
ExtensionPoint		See "ExtensionPoint (from UseCases)" on page 575.
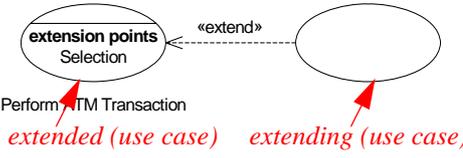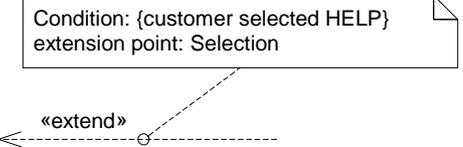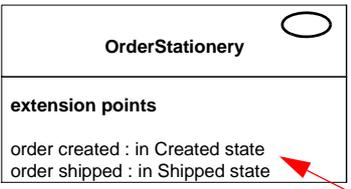
Table 16.1 - Graphic nodes included in use case diagrams

Node Type	Notation	Reference
Include	*including use case* *included use case*	See "Include (from UseCases)" on page 576.
UseCase		See "UseCase (from UseCases)" on page 578.
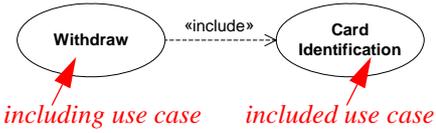
Examples
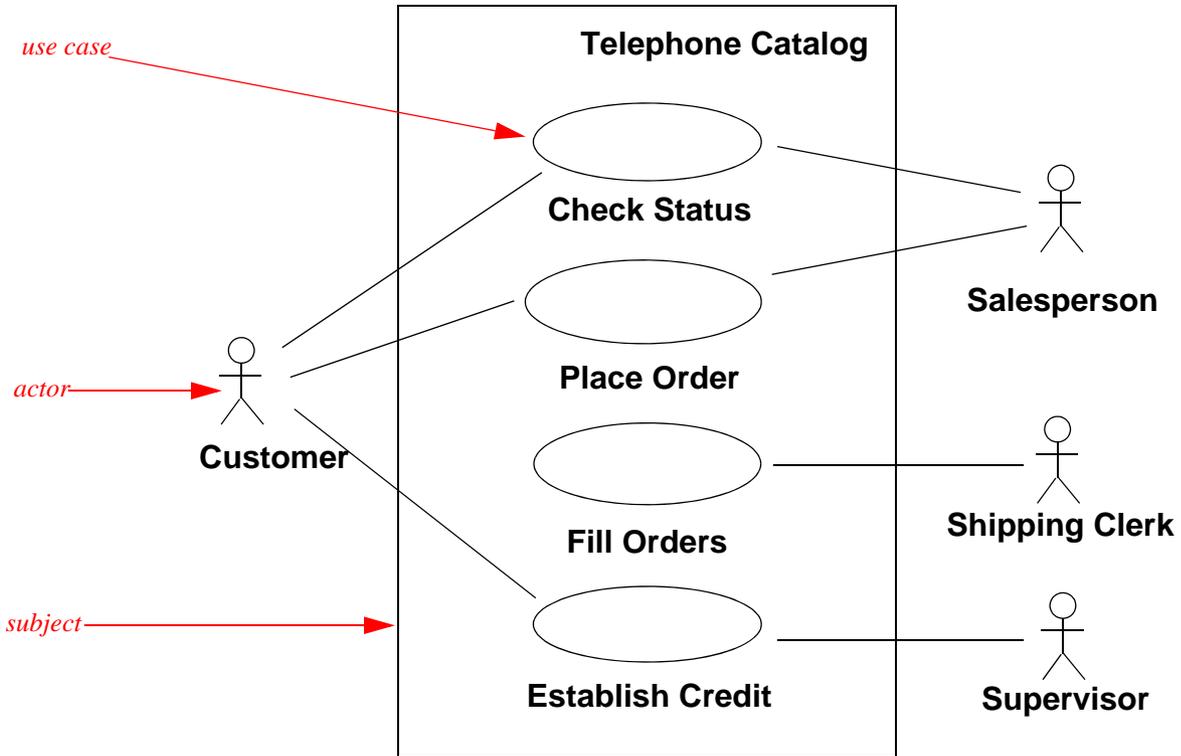
Figure 16.10 - UseCase diagram with a rectangle representing the boundary of the subject

The use case diagram in Figure 16.10 shows a set of use cases used by four actors of a physical system that is the subject of those use cases. The subject can be optionally represented by a rectangle as shown in this example.

Figure 16.11 illustrates a package that owns a set of use cases.

Note – A use case may be owned either by a package or by a classifier (typically the classifier specifying the subject).
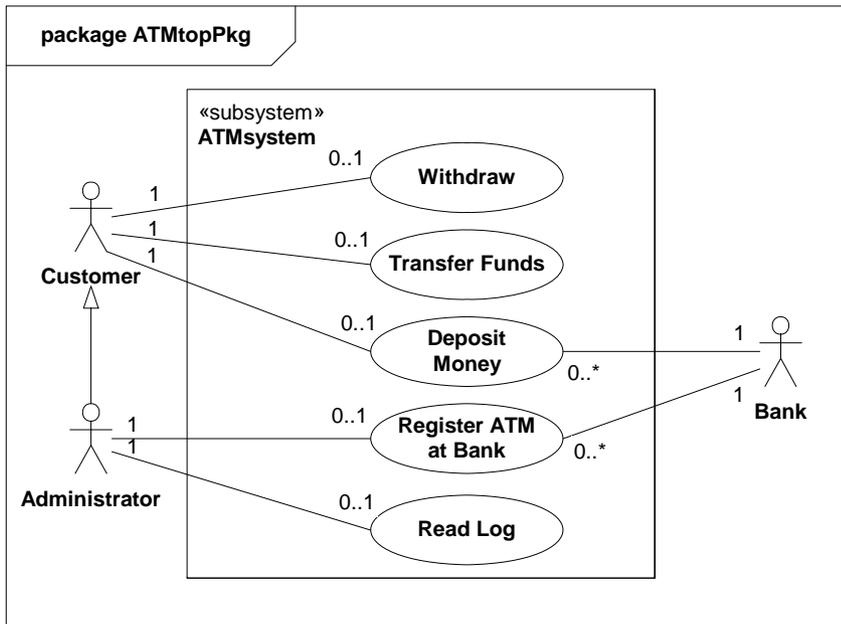
Figure 16.11 - Use cases owned by a package

Changes from previous UML

There are no changes from UML 1.x, although some aspects of notation to model element mapping have been clarified.