18 UseCases

18.1 Use Cases

18.1.1 Summary

UseCases are a means to capture the requirements of systems, i.e., what systems are supposed to do. The key concepts specified in this clause are *Actors*, *UseCases*, and subjects. Each UseCase's subject represents a system under consideration to which the UseCase applies. Users and any other systems that may interact with a subject are represented as Actors.

A UseCase is a specification of behavior. An instance of a UseCase refers to an occurrence of the emergent behavior that conforms to the corresponding UseCase. Such instances are often described by Interactions.

18.1.2 Abstract Syntax

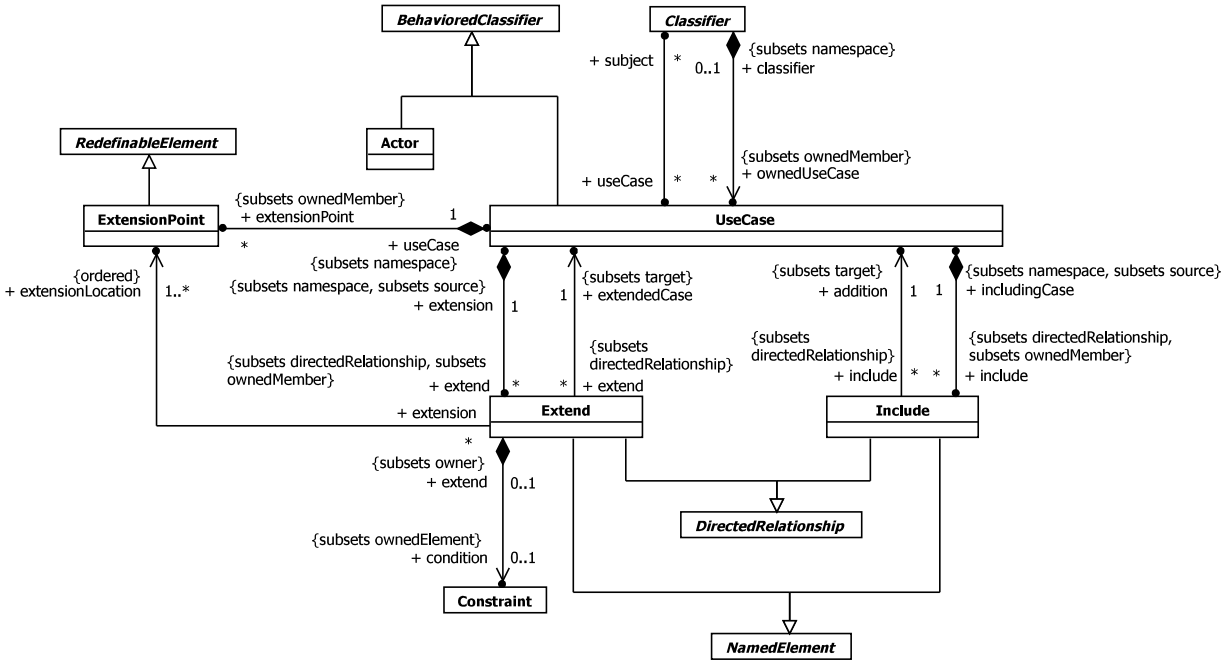

Figure 18.1 UseCases

18.1.3 Semantics

18.1.3.1 Use Cases and Actors

A UseCase may apply to any number of subjects. When a UseCase applies to a subject, it specifies a set of behaviors performed by that subject, which yields an observable result that is of value for Actors or other stakeholders of the subject.

A UseCase is a kind of BehavioredClassifier that represents a declaration of a set of offered Behaviors. Each UseCase specifies some behavior that a subject can perform in collaboration with one or more Actors. UseCases define the offered Behaviors of the subject without reference to its internal structure. These Behaviors, involving interactions between the Actors and the subject, may result in changes to the state of the subject and communications with its environment. A UseCase can include possible variations of its basic behavior, including exceptional behavior and error handling.

A subject of a UseCase could be a system or any other element that may have behavior, such as a Component or Class. Each UseCase 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 must always be completed for the UseCase 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 UseCase can be initiated again, or in an error state.

UseCases 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 UseCases may also state the requirements the specified subject poses on its environment by defining how the Actors should interact with the subject so that it will be able to perform its services.

The behaviors of a UseCase can be described by a set of Behaviors (through its ownedBehavior relationship), such as Interactions, Activities, and StateMachines, as well as by pre-conditions, post-conditions and natural language text where appropriate. It may also be described indirectly through a Collaboration that uses the UseCase and its Actors as the Classifiers that type its parts. Which of these techniques to use depends on the nature of the UseCase behavior as well as on the intended reader. These descriptions can be combined. An example of a UseCase with an associated StateMachine is shown in Figure 18.12.

UseCases may have associated Actors, which describe how an instance of the Classifier realizing the UseCase and a user playing one of the roles of the Actor interact. Two UseCases specifying the same subject cannot be associated as each of them individually describes a complete usage of the subject.

When a UseCase 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 the UseCase. The manner in which multiple Actors participate in the UseCase depends on the specific situation on hand and is not defined in this specification. For instance, a particular UseCase 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).

A UseCase may be owned either by a Package or by a Classifier. Although the owning Classifier typically represents a subject to which the owned UseCases apply, this is not necessarily the case, as illustrated by the example in Figure 18.10 and Figure 18.11.

An Actor models a type of role played by an entity that interacts with the subjects of its associated UseCases (e.g., by exchanging signals and data). Actors may represent roles played by human users, external hardware, or other systems.

NOTE. An Actor does not necessarily represent a specific physical entity but instead a particular role of some entity that is relevant to the specification of its associated UseCases. 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.

NOTE. The term “role” is used informally here and does not imply any technical definition of that term found elsewhere in this specification.

When an Actor has an association to a UseCase with a multiplicity that is greater than one at the UseCase end, it means that a given Actor can be involved in multiple UseCases 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 UseCases in parallel (concurrently) or at different points in time.

18.1.3.2 Extends

An Extend is a relationship from an extending UseCase (the extension) to an extended UseCase (the extendedCase) that specifies how and when the behavior defined in the extending UseCase can be inserted into the behavior defined in the extended UseCase. The extension takes place at one or more specific extension points defined in the extended UseCase.

Extend is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in one or more UseCases.

The extended UseCase is defined independently of the extending UseCase and is meaningful independently of the extending UseCase. On the other hand, the extending UseCase typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending UseCase defines a set of modular behavior increments that augment an execution of the extended UseCase under specific conditions.

NOTE. The same extending UseCase can extend more than one UseCase. Furthermore, an extending UseCase may itself be extended.

Extend is a kind of DirectedRelationship, such that the source is the extending UseCase and the target is the extended UseCase. It is also a kind of NamedElement so that it can have a name in the context of its owning UseCase. The Extend relationship itself is owned by the extension.

An ExtensionPoint identifies a point in the behavior of a UseCase where that behavior can be extended by an Extend relationship. Each ExtensionPoint has a unique name within a UseCase.

The specific manner in which the location of an ExtensionPoint is defined is intentionally unspecified. This is because UseCases may be specified in various formats such as natural language, tables, trees, etc. The intuition behind the notion of extensionLocation is best explained through the example of a textually described UseCase: Usually, a UseCase with ExtensionPoints consists of a set of finer-grained behavioral fragment descriptions, which are most often executed in sequence. This segmented structuring of the UseCase 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 UseCase typically consists of one or more behavior fragment descriptions that are to be inserted into the appropriate spots of the extended UseCase. An extensionLocation, therefore, is a specification of all the various ExtensionPoints in a UseCase where supplementary behavioral increments can be merged.

If the condition of the Extend is missing or evaluates to true at the time the first ExtensionPoint is reached during the execution of the extended UseCase, then all of the appropriate behavior fragments of the extending UseCase will also be executed. If the condition is false, this does not happen. The individual fragments are executed as the corresponding ExtensionPoints of the extended UseCase are reached. Once a given fragment is completed, execution continues with the behavior of the extended UseCase following the ExtensionPoint. Note that even though there are multiple UseCases involved, there is just a single behavior execution.

18.1.3.3 Includes

Include is a DirectedRelationship between two UseCases, indicating that the behavior of the included UseCase (the addition) is inserted into the behavior of the including UseCase (the includingCase). It is also a kind of NamedElement so that it can have a name in the context of its owning UseCase (the includingCase). The including UseCase may depend on the changes produced by executing the included UseCase. The included UseCase must be available for the behavior of the including UseCase to be completely described.

The Include relationship is intended to be used when there are common parts of the behavior of two or more UseCases. This common part is then extracted to a separate UseCase, to be included by all the base UseCases having this part in common. As the primary use of the Include relationship is for reuse of common parts, what is left in a base UseCase 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 UseCase depends on the addition but not vice versa.

All of the behavior of the included UseCase is executed at a single location in the included UseCase before execution of the including UseCase is resumed.

The Include relationship allows hierarchical composition of UseCases as well as reuse of UseCases.

18.1.4 Notation

A UseCase is shown as an ellipse, either containing the name of the UseCase or with the name of the UseCase placed below the ellipse. An optional stereotype keyword may be placed above the name.

A subject for a set of UseCases (sometimes called a *system boundary*) may be shown as a rectangle with its name in the top-left corner, with the UseCase ellipses visually located inside this rectangle. The same modeled UseCase may be visually depicted as separate ellipses within multiple subject rectangles. Where a subject is a Classifier with a standard stereotype, the keyword for the stereotype shall be shown in guillemets above the name of the subject. In cases where the metaclass of a subject is ambiguous, the keyword . corresponding to the notation for the metaclass of Classifier (see 9.2.4) shall be shown in guillemets above the name. Where multiple keywords and/or stereotype names apply, the

notational options defined by 9.2.4 shall apply. The subject notation is illustrated by the example in Figure 18.2 which shows a Component with the standard stereotype «Subsystem».

Note that this notation for the subject classifier differs from the normal Classifier notation – it has no header or compartments.

Note also that the subject rectangle does not imply that the subject classifier owns the contained UseCases, but merely that the UseCases apply to that classifier. In particular, there is scope for confusion between a UseCase appearing visually contained in a boundary rectangle representing a Classifier that is its subject, and appearing visually contained in a compartment of a Classifier that is its owner (see Figure 18.9).

Attributes and operations may be shown in compartments within the UseCase oval, with the same content as though they were in a normal Classifier rectangle.

ExtensionPoints may be listed in a compartment of the UseCase with the heading **extension points**. Each ExtensionPoint is denoted by a text string within the UseCase oval symbol 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 UseCase where the extension point occurs, such as the name of a State in a StateMachine, an Activity in an activity diagram, a precondition, or a postcondition.

UseCases may have other Associations and Dependencies to other Classifiers (e.g., to denote input/output, events, and behaviors).

The detailed behaviors defined by a UseCase are notated according to the chosen description technique, in a separate diagram or textual document.

A UseCase may also be shown using the standard rectangle notation for Classifiers with an ellipse icon in the upper-right-hand corner of the rectangle, as illustrated by the example in Figure 18.5. In this case, “extension points” is an optional compartment. This rendering is more suitable when there are a large number of extension points or features.

An Actor is represented by a “stick man” icon with the name of the Actor in the vicinity (usually above or below) the icon, as illustrated by the example in Figure 18.6.

An Actor may also be shown as a Classifier rectangle with the keyword «actor», with the usual notation for all compartments, as illustrated by the example in Figure 18.7.

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, as illustrated by the example in Figure 18.8.

The nesting (owning) of a UseCase by a Classifier may optionally be represented by nesting the UseCase ellipse inside the Classifier rectangle in a separate compartment, as illustrated by the example in Figure 18.9. This is a case of the optional compartment for ownedMembers described in 9.2.4.

An Extend relationship between UseCases is shown by a dashed arrow with an open arrowhead pointing from the extending UseCase towards the extended UseCase. The arrow is labeled with the keyword «extend». The condition of the Extend as well as references to the ExtensionPoints are optionally shown in a note symbol (see 7.2.4) attached to the corresponding arrow, as illustrated by the example in Figure 18.3.

An Include relationship between UseCases is shown by a dashed arrow with an open arrowhead pointing from the base UseCase to the included UseCase. The arrow is labeled with the keyword «include», as illustrated by the example in Figure 18.4.

18.1.5 Examples

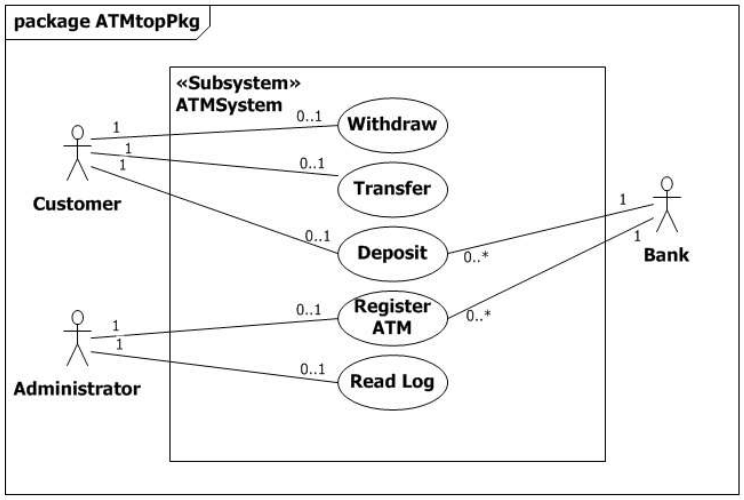

Figure 18.2 Class diagram of a Package owning a set of UseCases, Actors, and a Subsystem

Figure 18.2 illustrates a class diagram corresponding to the Package ATMtopPkg which owns a set of UseCases, Actors, and a Subsystem that is the subject of the UseCases. In this example the subject is a Component with the Subsystem standard stereotype. The metaclass need not be shown because the Subsystem stereotype necessarily implies that the subject is a Component - see Clause 22.

The example shows that a Customer or Administrator may or may not participate in any of their associated UseCases (hence the 0..1 multiplicity). From the UseCase perspective, every UseCase in the example must have an Actor to initiate it (hence the 1 multiplicity). The Deposit and Register ATM UseCases require participation by the Bank, while the bank can participate with many Deposit and Register ATM UseCases at the same time.

In the UseCase diagram in Figure 18.3 below, the UseCase "Perform ATM Transaction" has an ExtensionPoint "Selection." This UseCase is extended via that ExtensionPoint by the UseCase "On-Line Help" whenever execution of the "Perform ATM Transaction" UseCase occurrence is at the location referenced by the "Selection" extension point and the customer selects the HELP key.

NOTE. The "Perform ATM Transaction" UseCase is defined independently of the "On-Line Help" UseCase.

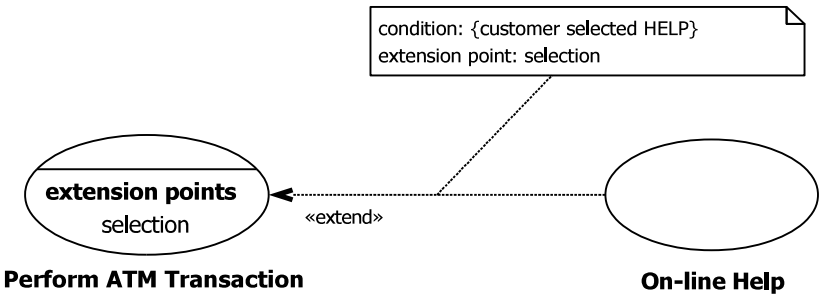

Figure 18.3 Example Extend

In Figure 18.4 below a UseCase "Withdraw" includes an independently defined UseCase "Card Identification."

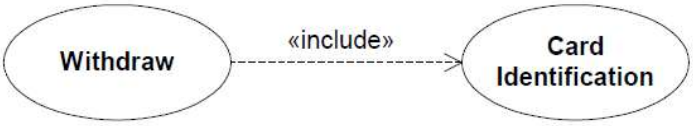

Figure 18.4 Example Include

Figure 18.5 shows a UseCase using the standard rectangle notation for Classifiers with an ellipse icon, and the optional "extension points" compartment.

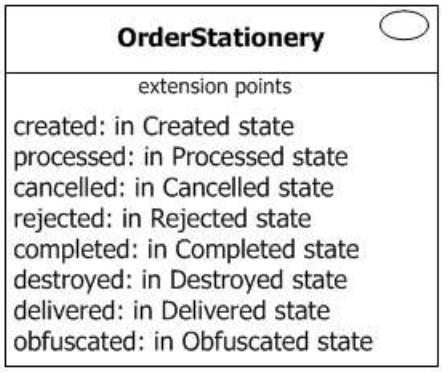

Figure 18.5 UseCase using Classifier rectangle notation

Figure 18.6, Figure 18.7 and Figure 18.8 exemplify the three different notations for Actors.

Figure 18.6 Actor notation using stick-man

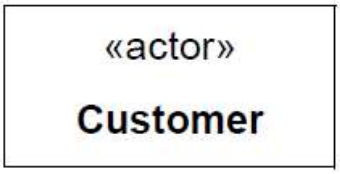

Figure 18.7 Actor notation using Class rectangle

Figure 18.8 Actor notation using icon

Figure 18.9 illustrates an ownedUseCase of a Class using an optional ownedMember compartment. The compartment name "owned use cases" is derived from the property name ownedUseCase according to the rules specified in 9.2.4.

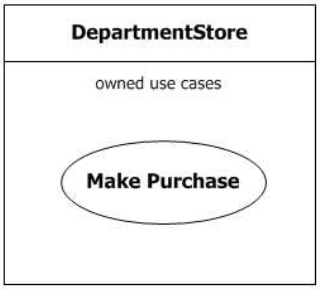

Figure 18.9 Notation for UseCase owned by Classifier

UseCases need not be owned by their subject. For example, the UseCases shown in Figure 18.10 below (which are functionally the same as those shown in Figure 18.2) apply to the "ATMsystem" subsystem but are owned by various packages as shown in Figure 18.11.

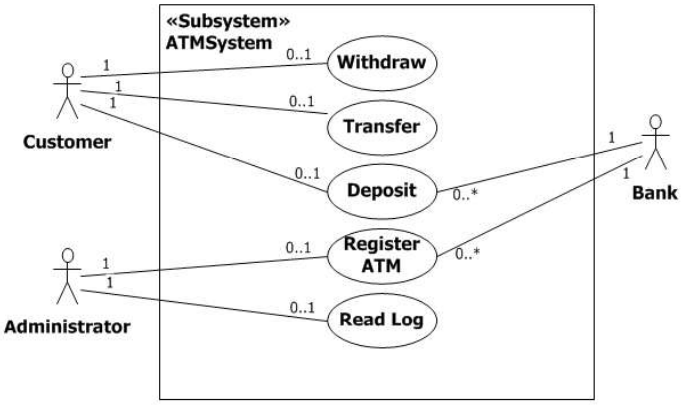

Figure 18.10 Example ATM system with UseCases and Actors

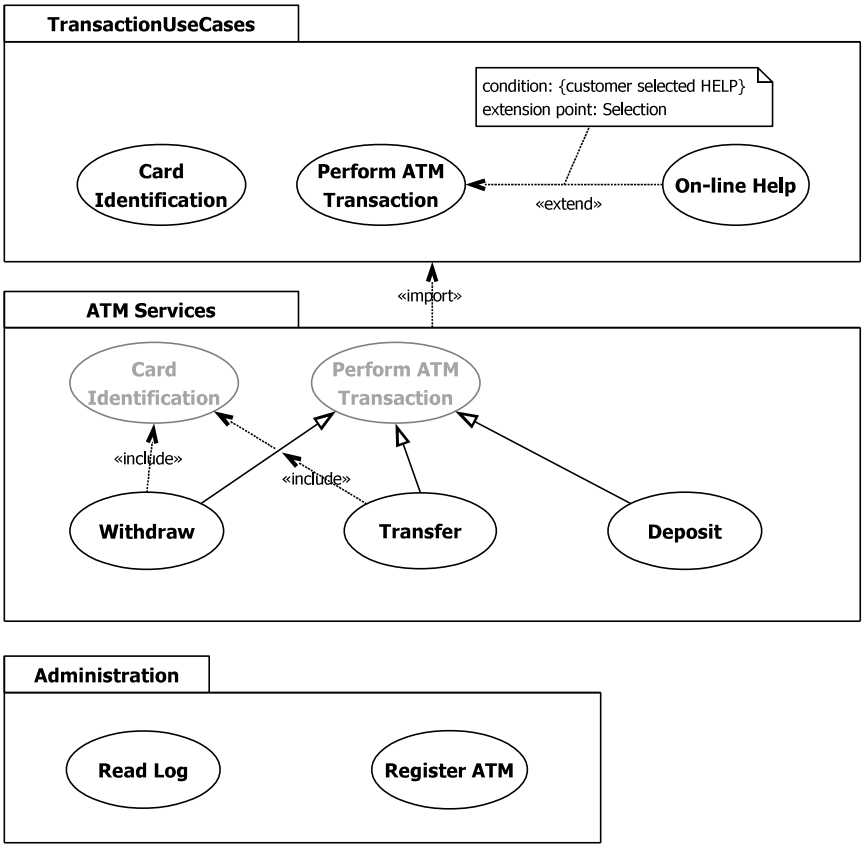

Figure 18.11 Example UseCases owned by Packages

Figure 18.12 shows a UseCase which has a StateMachine as one of its ownedBehaviors. The Classifier symbol for the StateMachine may be shown in the optional "owned behaviors" compartment, and the internal details of the StateMachine are shown in the state machine diagram on the right.

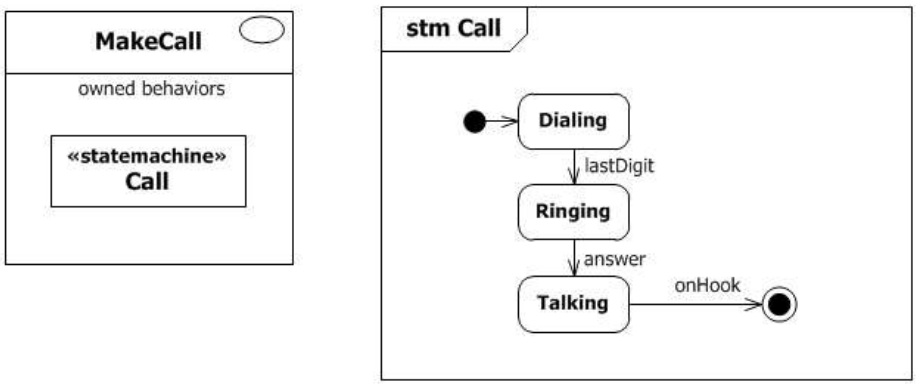

Figure 18.12 Example UseCase with associated StateMachine

18.2 Classifier Descriptions

18.2.1 Actor [Class]

18.2.1.1 Description

An Actor specifies a role played by a user or any other system that interacts with the subject.

18.2.1.2 Diagrams

Use Cases

18.2.1.3 Generalizations

BehavioredClassifier

18.2.1.4 Constraints

- associations
  An Actor can only have Associations to UseCases, Components, and Classes. Furthermore these Associations must be binary.

```
inv: Association.allInstances()->forAll( a |
  a.memberEnd->collect(type)->includes(self) implies
  (
    a.memberEnd->size() = 2 and
    let actorEnd : Property = a.memberEnd->any(type = self) in
      actorEnd.opposite.class.oclIsKindOf(UseCase) or
      ( actorEnd.opposite.class.oclIsKindOf(Class) and not
         actorEnd.opposite.class.oclIsKindOf(Behavior))
      )
  )
```

- must_have_name
  An Actor must have a name.

```
inv: name->notEmpty()
```

18.2.2 Extend [Class]

18.2.2.1 Description

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

18.2.2.2 Diagrams

Use Cases

18.2.2.3 Generalizations

NamedElement, DirectedRelationship

18.2.2.4 Association Ends

- ♦ condition : Constraint [0..1]{subsets Element::ownedElement} (opposite A_condition_extend::extend)
  References the condition that must hold when the first ExtensionPoint is reached for the extension to take place. If no constraint is associated with the Extend relationship, the extension is unconditional.

- extendedCase : UseCase [1..1]{subsets DirectedRelationship::target} (opposite A_extendedCase_extend::extend)
  The UseCase that is being extended.

- extension : UseCase [1..1]{subsets NamedElement::namespace, subsets DirectedRelationship::source} (opposite UseCase::extend)
  The UseCase that represents the extension and owns the Extend relationship.

- extensionLocation : ExtensionPoint [1..*]{ordered} (opposite A_extensionLocation_extension::extension)
  An ordered list of ExtensionPoints belonging to the extended UseCase, specifying where the respective behavioral fragments of the extending UseCase are to be inserted. The first fragment in the extending UseCase 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 UseCase has just a single behavior fragment, so that the list of ExtensionPoints is trivial.

18.2.2.5 Constraints

- extension_points
  The ExtensionPoints referenced by the Extend relationship must belong to the UseCase that is being extended.

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

18.2.3 ExtensionPoint [Class]

18.2.3.1 Description

An ExtensionPoint identifies a point in the behavior of a UseCase where that behavior can be extended by the behavior of some other (extending) UseCase, as specified by an Extend relationship.

18.2.3.2 Diagrams

Use Cases

18.2.3.3 Generalizations

RedefinableElement

18.2.3.4 Association Ends

- useCase : UseCase [1..1]{subsets NamedElement::namespace} (opposite UseCase::extensionPoint)
  The UseCase that owns this ExtensionPoint.

18.2.3.5 Constraints

- must_have_name
  An ExtensionPoint must have a name.

```
inv: name->notEmpty ()
```

18.2.4 Include [Class]

18.2.4.1 Description

An Include relationship specifies that a UseCase contains the behavior defined in another UseCase.

18.2.4.2 Diagrams

Use Cases

18.2.4.3 Generalizations

DirectedRelationship, NamedElement

18.2.4.4 Association Ends

- addition : UseCase [1..1]{subsets DirectedRelationship::target} (opposite A_addition_include::include)
  The UseCase that is to be included.

- includingCase : UseCase [1..1]{subsets NamedElement::namespace, subsets DirectedRelationship::source} (opposite UseCase::include)
  The UseCase which includes the addition and owns the Include relationship.

18.2.5 UseCase [Class]

18.2.5.1 Description

A UseCase specifies a set of actions performed by its subjects, which yields an observable result that is of value for one or more Actors or other stakeholders of each subject.

18.2.5.2 Diagrams

Use Cases, Classifiers

18.2.5.3 Generalizations

BehavioredClassifier

18.2.5.4 Association Ends

- ♦ extend : Extend [0..*]{subsets A_source_directedRelationship::directedRelationship, subsets Namespace::ownedMember} (opposite Extend::extension)
  The Extend relationships owned by this UseCase.

- ♦ extensionPoint : ExtensionPoint [0..*]{subsets Namespace::ownedMember} (opposite ExtensionPoint::useCase)
  The ExtensionPoints owned by this UseCase.

- ♦ include : Include [0..*]{subsets A_source_directedRelationship::directedRelationship, subsets Namespace::ownedMember} (opposite Include::includingCase)
  The Include relationships owned by this UseCase.

- subject : Classifier [0..*] (opposite Classifier::useCase)
  The subjects to which this UseCase applies. Each subject or its parts realize all the UseCases that apply to it.

18.2.5.5 Operations

- allIncludedUseCases() : UseCase [0..*]
  The query allIncludedUseCases() returns the transitive closure of all UseCases (directly or indirectly) included by this UseCase.

```
body: self.include.addition->union(self.include.addition->collect(uc |
uc.allIncludedUseCases()))->asSet()
```

18.2.5.6 Constraints

- binary_associations
  UseCases can only be involved in binary Associations.

```
inv: Association.allInstances()->forAll(a | a.memberEnd.type->includes(self) implies
a.memberEnd->size() = 2)
```

- no_association_to_use_case
  UseCases cannot have Associations to UseCases specifying the same subject.

```
inv: Association.allInstances()->forAll(a | a.memberEnd.type->includes(self) implies
   (
   let usecases: Set(UseCase) = a.memberEnd.type->select(oclIsKindOf(UseCase))-
>collect(oclAsType(UseCase))->asSet() in
   usecases->size() > 1 implies usecases->collect(subject)->size() > 1
   )
)
```

- cannot_include_self
  A UseCase cannot include UseCases that directly or indirectly include it.

```
inv: not allIncludedUseCases()->includes(self)
```

- must_have_name
  A UseCase must have a name.

```
inv: name -> notEmpty ()
```

18.3 Association Descriptions

18.3.1 A_addition_include [Association]

18.3.1.1 Diagrams

Use Cases

18.3.1.2 Owned Ends

- include : Include [0..*]{subsets A_target_directedRelationship::directedRelationship} (opposite Include::addition)

18.3.2 A_condition_extend [Association]

18.3.2.1 Diagrams

Use Cases

18.3.2.2 Owned Ends

- extend : Extend [0..1]{subsets Element::owner} (opposite Extend::condition)

18.3.3 A_extend_extension [Association]

18.3.3.1 Diagrams

Use Cases

18.3.3.2 Member Ends

- UseCase::extend
- Extend::extension

18.3.4 A_extendedCase_extend [Association]

18.3.4.1 Diagrams

Use Cases

18.3.4.2 Owned Ends

- extend : Extend [0..*]{subsets A_target_directedRelationship::directedRelationship} (opposite Extend::extendedCase)

18.3.5 A_extensionLocation_extension [Association]

18.3.5.1 Diagrams

Use Cases

18.3.5.2 Owned Ends

- extension : Extend [0..*] (opposite Extend::extensionLocation)

18.3.6 A_extensionPoint_useCase [Association]

18.3.6.1 Diagrams

Use Cases

18.3.6.2 Member Ends

- UseCase::extensionPoint
- ExtensionPoint::useCase

18.3.7 A_include_includingCase [Association]

18.3.7.1 Diagrams

Use Cases

18.3.7.2 Member Ends

- UseCase::include

- Include::includingCase

18.3.8 A_subject_useCase [Association]

18.3.8.1 Diagrams

Use Cases, Classifiers

18.3.8.2 Member Ends

- UseCase::subject
- Classifier::useCase