AMIDST

Use Cases in Object-Oriented Software Development

REFERENCE	:	AMIDST/WP2/N003/V02
DATE OF ISSUE	:	February 5, 1999
ACCESS RIGHTS	:	public
STATUS	:	final
EDITOR	:	Klaas van den Berg
AUTHOR[S]	:	Mehmet Aksit, Klaas van den Berg, & Pim van den Broek

SYNOPSIS

Use cases and their role in the software development process are described. The semantics of the uses relation and the extends relation between use cases are explained with control flowgraphs.

Document History

DATE	VERSION	MODIFICATION
November 26, 1998	01	Initial draft
February 5, 1999	02	Final draft

Abstract

In this document we describe use cases and its role in the software development process. There is no precise semantics of use cases. Use case descriptions can be formalized with control flowgraphs. Based on this formalization, the use of the standard technique of sequence charts can be improved, in particular for the uses relation and the extend relation between use cases.

Table of Contents

1. Introduction

Use cases, as introduced by Jacobson (Jacobson *et al.*, 1992), are frequently utilised in the requirements elicitation phase of software development. They are also part of the Unified Modeling Language UML (Rational, 1997; Booch, Rumbaugh & Jacobson, 1999). The role of use cases in software reuse is discussed in Jacobson *et al.*, 1997. There is a strong debate about the use of use cases (Berard, 1996; Cockburn & Fowler, 1998; Henderson-Sellers, Simons & Younessi, 1998, Appendix E; Simons, 1999). One of the critical points relates to the semantics of use cases.

The control-flow semantics of use case diagrams – and of the relationships between use cases - is not very well defined (Bergner *et al.*, 1998; Övergaard & Palmkvist, 1998). In this document, the control-flow semantics of use cases is described in terms of the well-established theory of control-flow graphs (Fenton & Whitty, 1986). There are other approaches to formalizing use cases (Hsia *et al.*, 1994; Regnell et al., 1996), but these do not address control flow of use case relations.

First, use case terminology is discussed and control-flow graphs are introduced briefly. Subsequently the mapping of use case diagrams and their relations onto control-flow graphs is described. Then the flow of control in sequence diagrams with branching is discussed. In the conclusion, guidelines are given for the descriptions of use cases with extends-relations and uses-relations based on the given semantics.

1.1 Use Cases

A use case (class) is a specification of actions, including variants, which a system (or other entity) can perform, interacting with an actor of the system. A use case is a specific way of using the system by performing some part of the functionality. A use case instance (also called a scenario) is a specific sequence of actions as specified in a use case carried out under certain conditions. A use case model or diagram contains a collection of related use cases (Jacobson *et al.*, 1992; Rational, 1997).

1.1.1 Use Case Relations

In Table 1 we list the terminology on uses cases and their relationships as being described for Objectory[♣] by Jacobson (Jacobson *et al.*, 1992), for SOMA by Graham (Graham, 1995), in the OPEN Modeling Language (OML) reference manual by Firesmith (Firesmith *et al.*, 1997) and the Unified Modeling Language (UML) semantics document (Rational, 1997).

[♣] Objectory: Object Factory for Software Development; SOMA: Semantic Object Modelling Approach; OPEN: Object-oriented Process, Environment and Notation

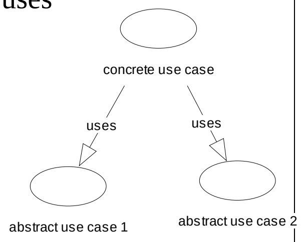
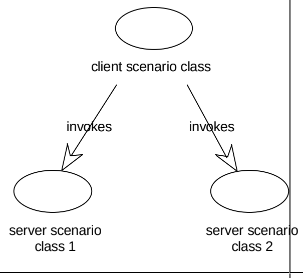
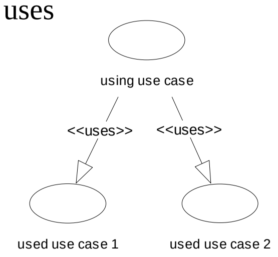
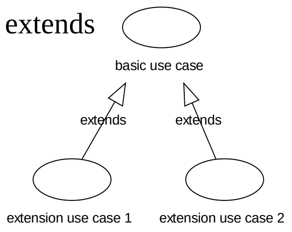
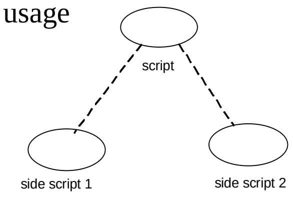
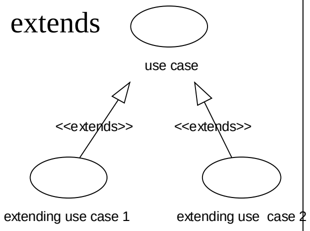
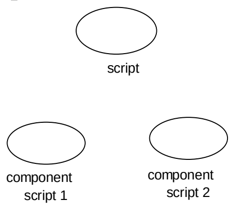
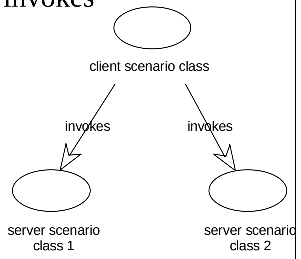
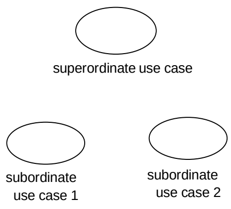
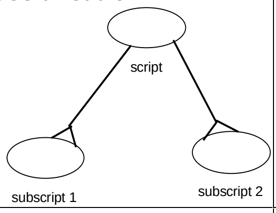
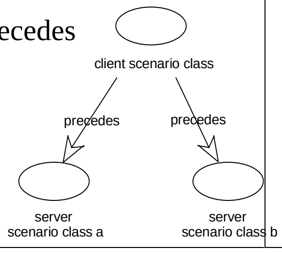
	Objectory Jacobson	**SOMA Graham**	**OML Firesmith**	**UML Rational**
1 common	uses  concrete use case uses uses abstract use case 1 abstract use case 2		invokes  client scenario class invokes invokes server scenario class 1 server scenario class 2	uses  using use case <<uses>> <<uses>> used use case 1 used use case 2
2 variant	extends  basic use case extends extends extension use case 1 extension use case 2	usage 		extends  use case <<extends>> <<extends>> extending use case 1 extending use case 2
3 component		composition 	invokes  client scenario class invokes invokes server scenario class 1 server scenario class 2	refines  superordinate use case subordinate use case 1 subordinate use case 2
4 specialised		specialisation 		
5 ordered			precedes  client scenario class precedes precedes server scenario class a server scenario class b	

Table 1. Terminology for five kinds of use cases and their relationships

In the overview in Table 1 we distinguish the following five kinds of use cases:

1. Common use cases. Common parts of use cases are factored out so that these can be (re)used by other use cases without repeating the description.

2. Variant use cases. In variant use cases, alternatives to the normal use case behaviour are captured. They are also used for exceptions.

3. Component uses cases. In component use cases, parts of use cases are further refined leading to a hierarchical decomposition of use cases.

4. Specialised use cases. Use cases may classified in more specialised versions.

5. Ordered use cases. Ordered use cases deal with situations where the completion of one use case is required before the following use case can be executed.

In OML (Firesmith *et al.*, 1997), the invokes-relationship is applied -in examples- to both common use cases and component use cases. Deviant in this table is the description of Graham (Graham, 1995) of the usage-relation between use cases (in his terminology scripts) and side-scripts. The side-scripts handle exceptions that require a redirection of the flow of control. A similar description is found in Jacobson (Jacobson *et al.*, 1992) and UML (Rational, 1997) for the extends-relation. The subscripts - which handle specialised cases – aim at a specialisation hierarchy as with inheritance.

1.1.2 Use Case Descriptions

Use cases can be described informally in natural language, semi-formal in structured natural language (SVDPI-sentences Subject-Verb-DirectObject - [Preposition - IndirectObject]) or in pseudocode, or with formal models, such as activity diagrams or sequence diagrams. Usually, one starts with an informal description, and gradually one proceeds to more formal models. In the following sections we will use these formal models for the description of use cases.

1.2 Use Cases in the Software Development Process

Use cases are used differently by current object-oriented methodologies. At first, in the Object Modeling Technique (OMT) there are no use cases (Rumbaugh et al., 1991). Later, Rumbaugh (Rumbaugh, 1996) extended OMT with the use of use cases are mainly to capture the requirements of a software system from a user-centered viewpoint. Uses cases are combined with domain analysis.

Object-Oriented Software Engineering (OOSE: Jacobson *et al.*, 1992) introduces use cases as a core concept in this methodology. In fact, use cases have their impact on all models in the software development process. In the OOSE terminology this is expressed as follows (Jacobson *et al.*, 1992: p. 132). The use case model is: expressed in terms of the domain object model, it is structured by the analysis model, it is realized by the design model, it is implemented by the implementation model, and it is tested by the testing model. This approach to use cases has been incorporated in the Objectory CASE-tool. Later, Jacobson joined the Rational company, and use cases are now a central part of the Unified Process (Jacobson, Booch & Rumbaugh, 1999) and are also one of the modeling techniques in the Unified Modeling Language (UML) (Booch, Rumbaugh & Jacobson, 1999).

2. Control-flow in use cases

In a use case instance, some path – i.e. a contiguous sequence of interactions (Firesmith *et al.*, 1997) - in the use case is taken. An actor requires some functionality of the system; this request provides the entry point of the use case. By performing a sequence of related actions this functionality is supplied by the system, either in a normal course of actions, in some variant course of actions, or by handling exceptions. After this, the exit point of the use case is reached.

The flow of control within each use case can be derived from interaction diagrams, i.e. the *sequence diagram* or the corresponding *collaboration diagram*. These diagrams can be mapped onto flowgraphs. A method m( ) sent to object Y is represented by action Y.m( ). The sequence of messages is represented by the arcs between the actions in the flowgraphs. The entry point of the use case is mapped onto the start node of the flowgraph and the exit point onto the stop node.

2.1 Control-flow graphs

A control-flow graph (Fenton & Whitty, 1986) (in short *flowgraph*) is a directed graph. The nodes in the graph represent actions (activity, method execution) and the arcs indicate the flow of control from one action to another. A flowgraph has two special nodes: the *start node* and the *stop node*. The stop node has no outgoing arcs and every node in a flowgraph lies on some path from the start node to the stop node (the one-entry one-exit property). A node with one outgoing arc is called an *action node*. A node with two or more outgoing arcs is called a *branch node*.

Elementary flowgraphs (primes) are selection with IF(c,*A*), IF(c,*A*,*B*), CASE(q,*A*,*B*,…) and iteration with WHILE(c,*A*) and REPEAT(*A*,c).

The *sequence*-operation of two flowgraphs *A* and *B*, denoted by *A*;*B*, is obtained by joining the stop node of *A* with the start node of *B*.

The *nesting*-operation of flowgraph *B* onto action node x in *A*, denoted by *A*(*B* on x), is obtained by replacing the outgoing arc of x in *A* by *B*. Often, the node x is not specified and nesting is denoted by *A*(*B*).

Flowgraphs that can be fully decomposed with sequencing and nesting into elementary flowgraphs are called *structured* flowgraphs. A large number of *metrics* has been defined to capture properties of flowgraphs, such as complexity, depth of nesting and testability (Fenton & Pfleeger, 1996).

Next, we discuss the control-flow semantics of use cases and each of the relationships between use cases in terms of control-flow graphs. From now on we use – as far as possible - the UML-notation and terminology for the description of uses cases and their relations.

2.1.1 Control-flow with common use cases

Common parts of use cases can be factored out so that these can be (re)used by other use cases without repeating the description. A use case may then depend on other (subordinate) use cases, i.e. the *uses*-relation between use cases. The resultant use case is obtained by placing the subordinate use cases at the appropriate place in the (superordinate) use case, i.e. the extension point (Rational, 1997) where the subordinate use case is called. "An extension point is a location at which the use case can be extended with additional behaviour". In the flowgraph, this is represented by *nesting* the subflowgraphs onto the (superordinate) flowgraph (see Table 2). Here, use case B uses one other use case D. The location of nesting is given by *extension points* d in B, i.e. D is called/invoked in d. As with flowgraphs, the control-flow for use cases with subordinate use cases can be obtained by nesting the sequence diagram of the used use case onto the sequence diagram of the using use case.

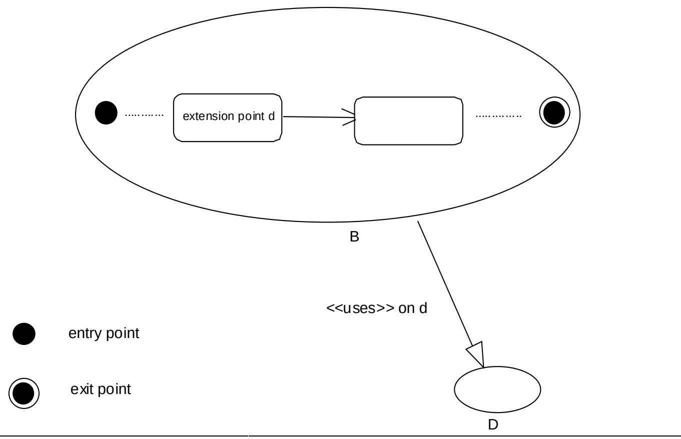
use cases relation	UML-notation	flowgraph
uses	 extension point d B <<uses>> on d entry point exit point D	B(D on d) extension point d in B

Table 2. Mapping of common use cases onto flowgraphs

2.1.2 Control-flow with variant use cases

In variant use cases, alternatives to the normal use case behaviour are captured. They are also used for special cases and exceptions. A use case may then be extended with other use cases, i.e. the *extends*-relation between use cases. The extensions are subject to conditions. The actual flow of control in the instantiated use case is determined at ’run-time’.

We follow the description by Jacobson (Jacobson *et al.*, 1992, p 165): What happens when a course is inserted in this way is as follows. The original use case runs as usual up to the point where the new case is to be inserted. At this point, the new course is inserted. After the extension has finished, the original course continues as if nothing had happened. …. The use case is not inserted only when the condition is true, but instead always takes place. Actually, the condition is always checked. If it is true, the whole course with extension is initiated; otherwise the original course continues directly.

The mapping onto flowgraphs is given in Table 3. This example is given for one extension only, i.e. use case B extends use case A at the extension point x and on the condition c. The extension point x is part of an if-then construct in A. The extension is mapped onto the flowgraph with a nesting of the flowgraph B onto A in x. The actual flow of control is determined by the value of c. If the extend condition c is fulfilled then use case B is executed. In the extended use case A, the extension point x can be just a dummy action node.

From this it can be seen that a uses-relation is semantically equivalent to an extends-relation (with if-then) for which the condition is always satisfied.

Another semantics is provided with an if-then-else construct in the extended case A. If the extend condition is not fulfilled the normal course is followed and action (or use case) D is executed, followed by the rest of the course in A. If the condition is fulfilled the extending use case B is executed instead of D, and then the rest of the course in A is taken. Now, the extending use case can be seen as an alternative to the normal course in use case D. This extends-relation can be seen as an 'extends-with-alternative'.

As with flowgraphs, the control-flow for use cases with extensions can be obtained by nesting the sequence diagram of the extension use case onto the sequence diagram of the extended use case.

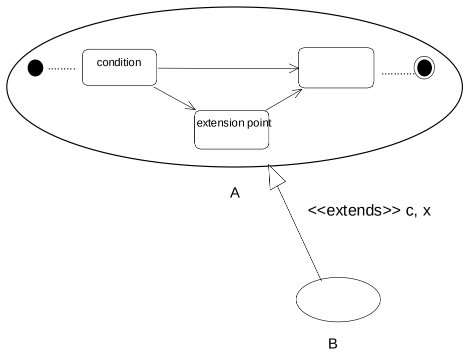
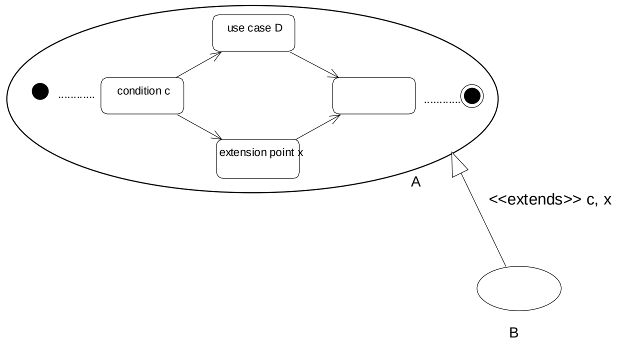
use cases relation	UML-notation	flowgraph
extends		*A*(*B* on x)) in A: if c then x extension condition c extension point x
extends	 use case D condition c extension point x A <<extends>> c, x B	*A*(*B* on x) in A: if c then x else D extension condition c extension point x

Table 3. Mapping of variant use cases onto flowgraphs

2.1.3 Control-flow with component uses cases

In component use cases, parts of use cases are further refined leading to a hierarchical decomposition of use cases. For each part it must be specified at which point in the superordinate use the subordinate use case has to be inserted. This is exactly the same situation as described for the uses-relation for common use cases. The mapping onto flowgraphs is given in the section on common use cases.

2.1.4 Control-flow with specialised use cases

Use cases can be classified in more specialised versions. The specialised use case – the sub use case - only contains the additional behaviour for the specialisation and inherits the other behaviour of the unspecialised use case – the super use case. It has to be specified on which condition the specialised use case should be taken and at which point the behaviour from the sub use case has to be inserted in the super use case. This is exactly the same situation as described for the extends-relation with variant use cases. The mapping onto flowgraphs is given in the section on variant use cases.

2.1.5 Control-flow with ordered use cases

Ordered use cases deal with situations where the completion of one use case is required before the following use case can be executed (Firesmith *et al.*, 1997). A (client) use case may then *precede* another (server) use case, i.e. the first use case must be completed first before the second use can be executed (see Table 4). We use the (not predefined) UML-stereotyped association <<precedes>> for this relation (or in tables and figures briefly <<p>>).

Precedes is a here defined as a stereotyped association between use cases. It specifies that the content of the preceded use case is added to the related use case. When an instance of the related use case has completed its sequence of actions, the sequence continues with the sequence of actions of the preceded use case. The mapping onto a control-flow graph is a *sequencing* of control-flow of the use cases.

If a selection has to be made between two component use cases, this selection should be incorporated into the superordinate use case. This maps onto an IF-THEN-ELSE flowgraph.

If iteration has to be performed on a component use case, this iteration should be incorporated into the superordinate use case. This maps onto a WHILE flowgraph.

A use case may be followed by two use cases in a precedence relation (a fork) or a use case may be preceded by two use cases in a precedence relation (a join) (see the precedence rhombus in Table 4). In this example A precedes B and A precedes C (a fork); furthermore B precedes D and C precedes D (a join). There is no precedence relation between use case B and C so that these use cases may be carried out in any order or even in parallel. However, parallel execution of flowgraphs is not covered in flowgraph theory (Fenton & Pfleeger, 1996). Possible instances with sequencing are given in the table. Any of the use cases may be empty (dummy use cases): e.g., if A is empty then this dummy use case provides the (empty) start node of the use case flowgraph; if D is empty then it provides the stop node of the flowgraph.

In the requirements elicitation phase, a fork-precedence relation between use cases may be quite natural to model parallel use cases. However, the precedence rhombus can easily be confused with a selection between alternative use cases.

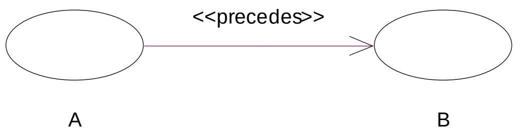
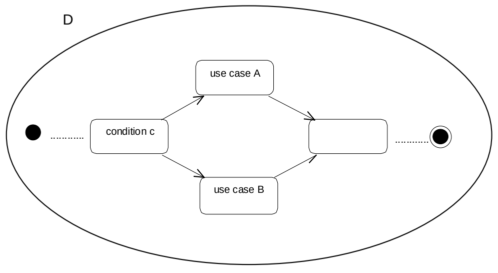
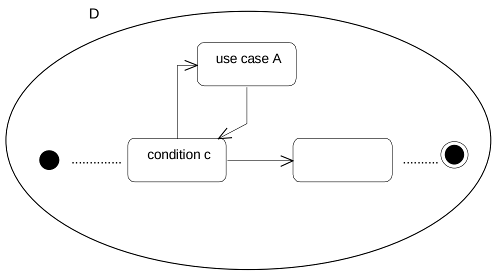
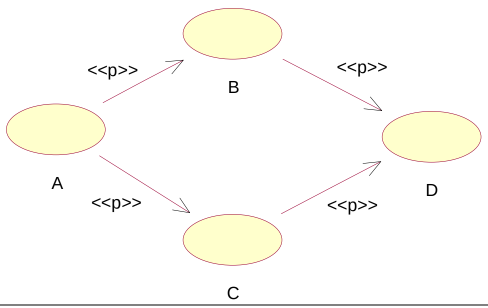
use cases relation	UML-notation	flowgraph
preceding		A; B
selection		D = …IF(c, *A*, *B*)… condition c A, B are components of superordinate D
iteration		D = …WHILE(c,*A*)… condition c A is component of superordinate D
precedence rhombus		general: A ; (B || C) ; D instances: A ; B ; C ; D A ; C ; B ; D

Table 4. Mapping of ordered use cases onto flowgraphs

2.2 Interleaving of use cases with uses-relationship

In Jacobson (Jacobson *et al.*, 1992) and UML (Rational, 1997), it is described that a use case may have several uses-relationships with other use cases. The resulting sequence in the instantiated use case will be obtained by interleaving the used sequences.

An example is given in Figure 1. Use case A has 4 subordinate use cases, each indicated with a (numeric) label. These components are A[1], A[7], A[3] and A[12]. The components lie on a path (a possible sequence) in use case A. Use case B has 3 components, and use case C has 5 components. Use case C is the using use case, and use cases A and B are the used use cases.

The uses-relation between use cases is expressed by a list of tuples, in which the first component refers to the used label and the second component to the using label. A label refers to a one-entry one-exit use case component. All labels are assumed to be unique. The use case of the used label is placed onto the use case of the using label. If there is more than one path in a use case then the uses-relation should be defined for each path separately. We assume that interleaving has the following properties:

1. The resultant use case does not depend on the order in which the use cases are being used.

2. The uses-relation between use cases preserves the order of the use cases involved, i.e. the order of components in the resulting use case corresponds to the order of the components in the using use cases and the used use cases.

There are two conditions to be satisfied to obtain this order preserving interleaving of use cases:

1. The used labels in the uses-relation must lie on a path in the used use case; in other words they are a subsequence of the labels in the used case.

2. The using labels in the uses-relation must lie on a path in the using use case; in other words they are a subsequence of the labels in the using case.

Furthermore, the using labels in the uses-relations must be unique, i.e. no using use case can use another use case more than once.

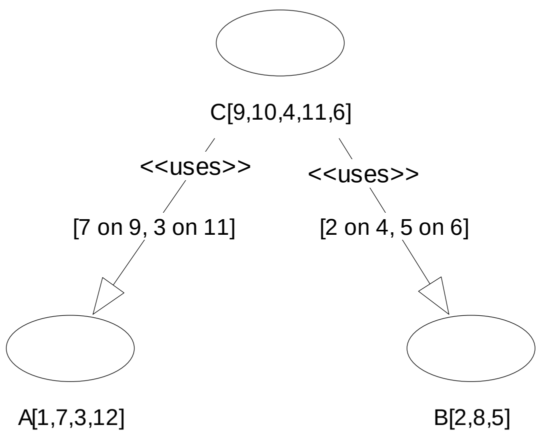


Figure 1. Multiple uses-relation between use cases

The subsequence-condition can be shown in the expanded view on the uses-relation as given in Figure 2. In the view this condition means that uses-lines between using use case and used use cases should not cross. The resulting use case consists of A[7], C[10], B[2], A[3], B[5]. The two conditions are fulfilled and the order of components of all use cases involved is preserved.

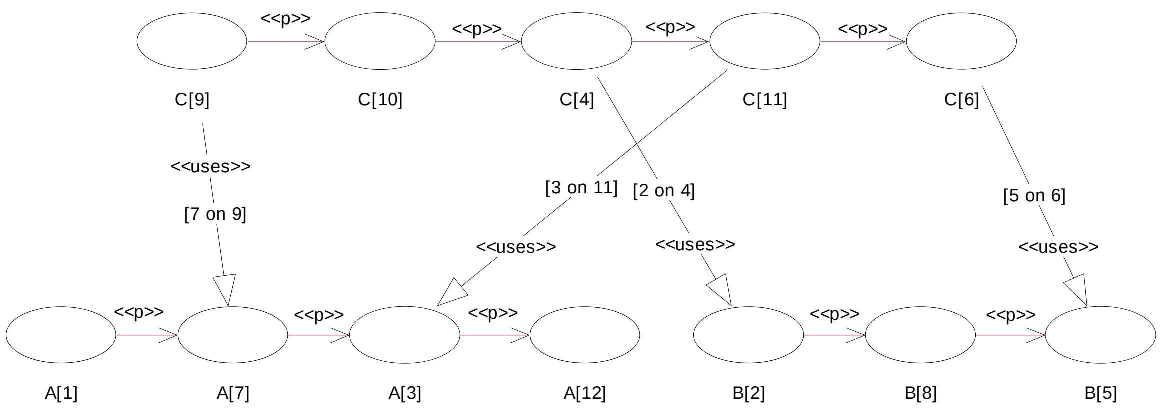


Figure 2. Expanded view on multiple uses-relation between use cases from Figure 1

2.3 Control-flow in sequence diagrams

The flow of control in use cases can be displayed in interaction diagrams, especially the sequence diagrams. However, with branching the flow of control is not always obvious. We model branching through objects with auxiliary lifelines. Once the condition is not anymore determinative, the auxiliary lifeline is joined with the main lifeline. The values of the conditions are displayed at each branching point. The flow of control can be read quite easily now from the sequence diagrams as shown in the figures given below.

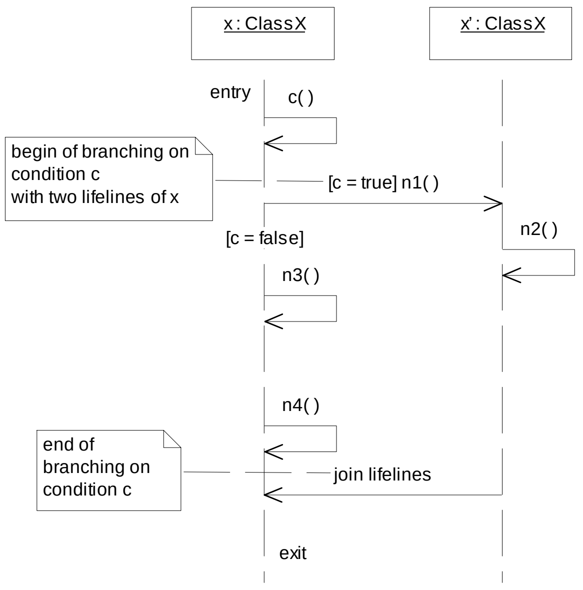


Figure 3. Branching in a sequence diagram with auxiliary lifeline

In Figure 3 the value of condition c is established. If c is true then message n1 is sent to object x followed by sending n2, otherwise message n3 is sent to x followed by n4. In order to visualise these branches, object x' is introduced. This object x' is the same as object x, however with an own auxiliary lifeline. After sending n2 the flow of control is going back to the main lifeline of the object x. At sending n3 to object x, on the lifeline of x, there is an (implicit) assumption that condition c is false. We can map this sequence diagram onto flowgraphs. The corresponding flowgraph in this case is: x.c(); IF(c,(x.n1(),x.n2()),(x.n3(),xn4))

Now, there are three types of arrows being used in sequence diagrams: with a message sent to the target object, a return value to the target object, and – as introduced above – solely the transfer of control to the target object (which is also implicit with the other arrows). Each of the arrows may have additionally a guard showing the condition on the flow of control. It is recommended to indicate the type of arrow being used in the diagrams (by adding the message name, return or join/merge/transfer respectively).

Also other objects may be involved in branching. In Figure 4, again the value of condition c is established. If c is true then message m1 is sent to object y otherwise message m2 is sent to y. In order to visualise these branches, object y' is introduced with an auxiliary lifeline. After sending m2 and m4 the flow of control is going back from the auxiliary lifeline to the main lifeline of object y. The corresponding flowgraph for this sequence diagram is: x.c(); IF(c,(y.m1(); y.m3()),(y.m2();y.m4())). In this example, the flow of control ends at object y, which provides the exit point of the (partial) sequence diagram.

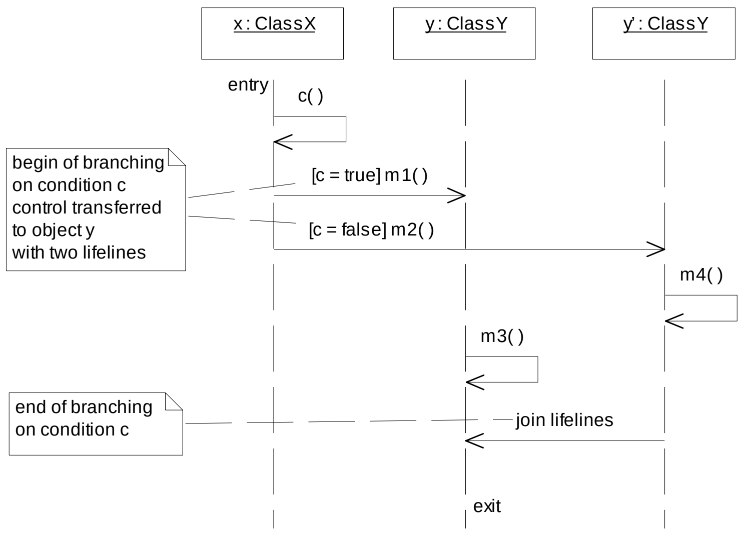


Figure 4. Branching in a sequence diagram to other object with auxiliary lifeline

2.3.1 Extension points in sequence diagrams

In the use cases presented in the previous sections there are extension points for relations with other use cases. Usually, an extension point has to be added to a use case once the need for a relation with another use case becomes apparent. An extension point z in a sequence diagram may be modelled by some message sent to a (dummy) object z. If there is a condition on the relation then this will be indicated on the branches. It must be clear which part of the use case is involved in the extension as part of the branching. An example is given in Figure 5. The original use case just contains one message m sent to object x, being the 'normal' course in the use case (part a of the figure). The extension of this use case in z is subject to condition c. The use case can be adapted for the extension with the branching IF c THEN z ELSE x.m( ) END (part b of the figure). The sequence diagram of the extending use case can be inserted on the extension point z (part c of the figure). In terms of flowgraphs, this is a nesting of the flowgraph of the extending use case onto the flowgraph of the original use case.

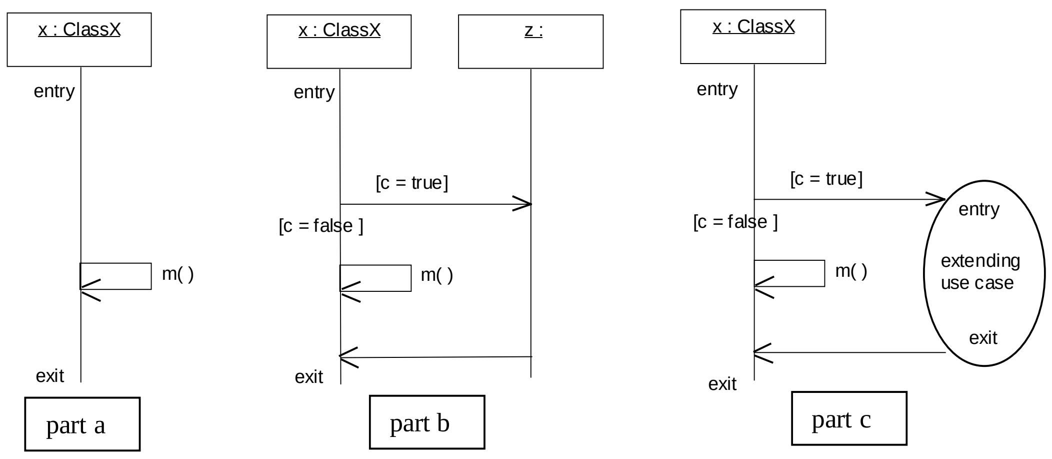


Figure 5. A sequence diagram with a conditional extension point

2.3.2 Activity diagrams and sequence diagrams

The flow of control in use cases may be described with UML-activity diagrams (Rational, 1997; Fowler & Scott, 1997). The semantics of activity diagrams can be described in terms of control-flow graphs in a similar way as shown above for sequence diagrams. The rules for nesting and sequencing activity diagrams are the same as for control-flow graphs. An example activity diagram is given in Figure 6 for the sequence diagram in Figure 4.

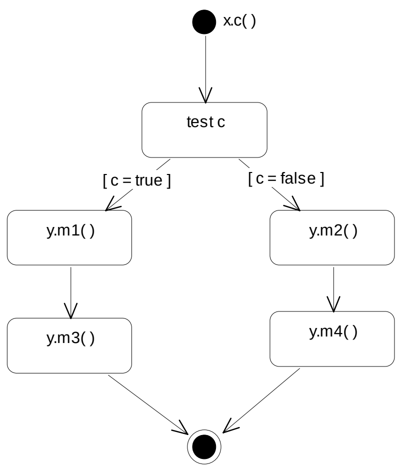


Figure 6. Activity diagram corresponding to sequence diagram in Figure 4

2.3.3 Tools and Testing

Use cases may be used for deriving tests for the resulting software. The mapping onto flowgraphs allows the use of testability metrics for a number of test strategies: all-path testing, visit-each loop path testing, simple path testing, branch testing and statement testing. For structured flowgraphs the set can be derived from the component flowgraphs and the flowgraphs onto which they are nested (Fenton & Pfleeger, 1996).

For the analysis of flowgraphs there are several tools available, such as Prometrix and Qualms (for references, see Fenton & Pfleeger, 1996). Metric values can be obtained with these tools. These static analysers need a front-end in which a flowgraph representation is derived, in this case from the sequence diagrams of use cases.

Without such analysers, we have to derive tests based on the flow of control in use cases directly from sequence diagrams, for example in the Rational Rose tool. However, conditional behaviour with branching is not (yet) supported in Rose98, nor support is provided for the UML-defined activity diagrams.

2.4 Guidelines

The control-flow semantics of use cases can be described in the well-established model of control-flow graphs. A prerequisite is that use cases have the one-entry one-exit property. If not then one may obtain unstructured use cases with an ill-defined flow of control, as the use of goto-statements in conventional programming may result in spaghetti-code.

The control-flow of the extends-relation and uses-relation between use cases has been described in terms of nesting of flowgraphs; the precedes-relation is given as a sequencing of flowgraphs. It is shown that the uses-relation is semantically equivalent with an unconditional extends-relation. Parallel execution of use cases cannot be mapped onto standard flowgraphs.

use case behaviour	**relation**	**control-flow semantics**
- common - component	generalization <<uses>>	behaviour is inserted unconditionally
- variant - specialised	generalization <<extends>>	behaviour is inserted conditionally
- ordered	dependency <<precedes>>	behaviour is appended unconditionally

Table 5. Five kinds of use cases with their control-flow semantics (from UML 1.1)

In Table 5 a summary is given of the control-flow semantics for the five kinds of use cases described in the first part of this document[1]. Both common use cases and component use cases have the control-flow semantics of the uses-relation between use cases, whereas variant uses cases and specialised use cases have the semantics of the extends-relation. Ordered use cases have the control-flow semantics of a precedes-relation in which behaviour is of one use case is sequenced (appended) to the behaviour of the preceding use case. Furthermore, we have augmented the notation for branching in sequence diagrams with auxiliary lifelines to visualise the flow of control.

With the mapping of use cases onto flowgraphs, the corresponding theory of flowgraphs can be applied to the analysis of use case diagrams, among others with metrics for structuredness, complexity and testability.

[1] In the Appendix this table is revised to adapt changes proposed in version UML 1.2 and 1.3 (Table 6)

From the analysis of use cases with flowgraphs given above, seven guidelines are derived, which - once followed - facilitate reasoning about the flow of control in use cases and related sequence diagrams:

- Define for each use case and its sequence diagram both the *entry* point and the *exit* point. These points are prerequisites for a well defined flow of control in use cases with uses-relationships and extends-relationships.

- Give for each used use case (in a uses-relation) the precise extension point in the using use case.

- Provide for each extending use case (in an extends-relation) an explicit if-then(-else) construct in the extended use case, together with the extension condition and the extension point, and - if applicable - the component in the normal use case for which the extension is an alternative.

- Do not use precedence-forks from use cases (a use case followed by more than one use cases in a *precedes*-relation), unless explicit parallelism is required. If used then the related join use case should be provided.

- Provide an if-then-else construct in the superordinate use case for selection of alternative component use cases, and a while construct for repetition of a component use case.

- Model branching in sequence diagrams with auxiliary objects with their own temporary lifeline.

- Label arrows between objects in sequence diagrams with either a message, a return or a join/merge.

3. Conclusion

In this document we described use cases and their role in the software development process. There is a strong debate about the precise semantics of use cases. We introduced a formalization of use case descriptions based on control flowgraphs. With this formalization, the use of the standard technique of sequence charts can be improved, in particular for the uses relation and the extend relation between use cases.

References

[Berard, 1996] Berard, E.V. (1996). Be Careful With “Use Cases”. See: http://www.tao.com/pub/html/use_case.html

[Bergner *et al.*, 1998] Bergner, K., Raush, A. & Sihling, M. (1998). A Critical Look upon UML 1.0. In: M. Schader & A. Korthaus (Eds.) (1998). The Unified Modeling Language. Physica-Verlag, pp. 97-92.

[Booch, Rumbaugh & Jacobson, 1999] Booch, G. Rumbaugh, J. & Jacobson, I. (1999). The Unified Modeling Language User Guide. Addison Wesley Longman

[Cockburn & Fowler, 1998] Cockburn, A. & Fowler, M. (1998). Question Time! about Use Cases. OOPSLA’98. ACM Sigplan Notices 33(10) 226-229.

[Fenton & Pfleeger, 1996] Fenton, N.E. & Pfleeger, S.L. (1996), *Software Metrics, A Rigorous & Practical Approach*. 2nd edition. Thomson, London

[Fenton & Whitty, 1986] Fenton, N.E. & Whitty, R.W. (1986). Axiomatic approach to software metrication through program decomposition, *Computer Journal*, vol. 29, no. 4, pp.329-339

[Firesmith *et al.*, 1997] Firesmith, D., Henderson-Sellers, B. & Graham, I. (1997). OPEN Modeling Language (OML) Reference Manual. Sigs, New York

[Fowler & Scott, 1997] Fowler, M. & Scott, K. (1997). UML Distilled. Applying the Standard Object Modeling Language. Addison-Wesley, Reading

[Graham, 1995] Graham, I. (1995). Migrating to Object Technology. Addison-Wesley, Wokingham

[Henderson-Sellers, Simons & Younessi, 1998] Henderson-Sellers, B., Simons, A. & Younessi, H. (1998). The OPEN Toolbox of Techniques. Addison-Wesley, Harlow

[Hsia *et al.*, 1994]Hsia, P.H., Samuel, J., Gao, J., Kung, D., Toyoshima, Y. & Chen, C. (1994). Formal Approach to Scenario Analysis. *IEEE Software* 11(2), March 1994, pp. 33-41

[Jacobson *et al.*, 1992] Jacobson, I., Christerson, M. Jonsson, P. & Ővergaard, G. (1992). Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley, Wokingham

[Jacobson *et al.*, 1997] Jacobson, I, Griss, M. & Jonsson, P. (1997). Software Reuse. Architecture, Process and Organization for Business Success. Addison Wesley Longman

[Jacobson, Booch & Rumbaugh, 1999] Jacobson, I., Booch, G. & Rumbaugh, J. (1999). The Unified Software Development Process. Addison Wesley Longman

[Övergaard & Palmkvist, 1998]. Övergaard, G. & Palmkvist, K. (1998). A Formal Approach to Use Cases and their Relationships. Workshop <<UML>> '98. http://www.it.kth.se/~gunnaro/www/index.html

[Rational, 1997] Rational (1997). UML Summary, Semantics, Notation Guide, Version 1.1, Rational Software Corporation

[Regnell et al., 1996]. Regnell, B., Andersson, M. & Bergstrand, J. (1996). A Hierarchical Use Case Model with Graphical Representation. Proceedings ECBS'96, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). Object-oriented Modeling and Design. Prentice Hall.

[Rumbaugh, 1996] Rumbaugh, J. (1996). OMT Insights. SIGS Books.

[Simons, 1999] Simons, A.J.H. (1999). Use Cases Considered Harmful. Submitted to ECOOP99.

Appendix

In the emerging version of UML 1.2 and 1.3 some major changes are expected with respect to use cases. Rational profoundly changed the description of the relations between Use Cases in UML version 1.2 (and 1.3) as compared to version 1.1. The new description can be found in Booch, Rumbaugh & Jacobson, 1999, pp. 226/8. The new versions 1.2 and 1.3 are not yet available at the Rational website http://www.rational.com.

In UML version 1.1 (as described in this document):

1. The <<extends>> relation between use cases was described as specialisation but was actually modelling variant behaviour

2. The generalisation relation was abused for both the <<uses>> and the <<extends>> relation between use cases

3. There was no (proper) specialisation relation between use cases

In UML version 1.2 / 1.3:

1. The old <<uses>> is now replaced by <<includes>>. It models common behaviour. It is denoted by a dependency relation between use cases with the arrowhead pointing to the included use case (compare the OML invokes)

2. The new <<extends>> is now used to model variant behaviour. It is denoted by a dependency relation between use cases with the arrowhead pointing to the extended use case

3. There is a (proper) specialisation relation between use cases denoted by the generalisation relation with the (open) arrowhead pointing to the general use case.

The new situation leads to the revised Table 5 in this document:

	use case behaviour	**relation**	**control-flow semantics**
UML 1.2 / 1.3	- common - component	dependency <<includes>>	behaviour is inserted unconditionally
	- variant	dependency <<extends>>	behaviour is inserted conditionally
	- specialised	generalisation	behaviour is replaced conditionally
OML / this document	- ordered	dependency <<precedes>>	behaviour is appended unconditionally

Table 6. . Five kinds of use cases with their control-flow semantics (from UML 1.2/1.3)