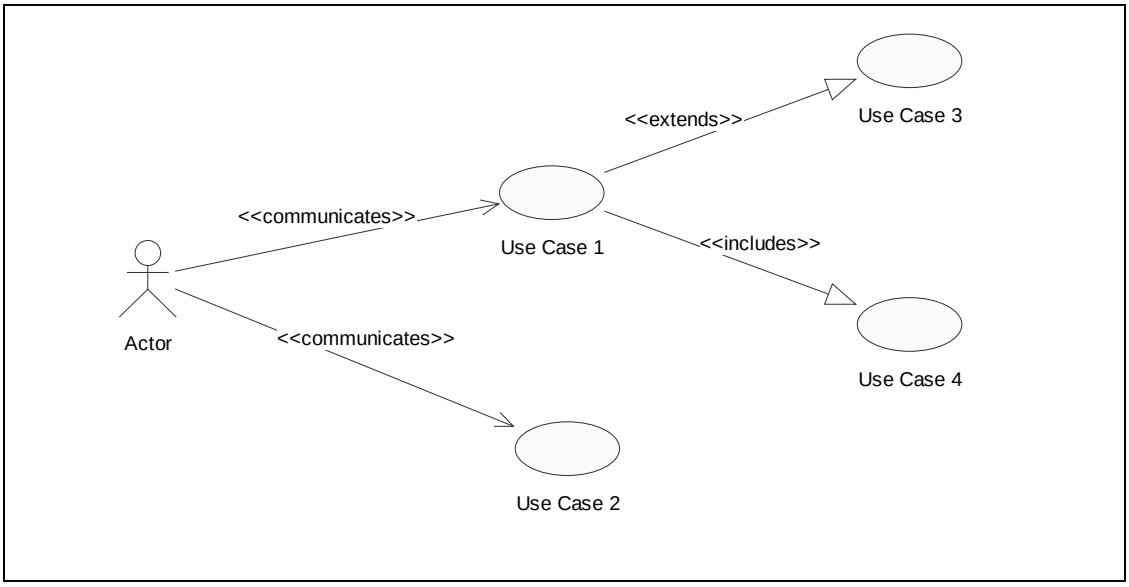


Item: **Use case modeling**

Author: Søren Langkilde Madsen
mail: soeren.langkilde@tietoenator.com
mail: slm@fagerbo.dk

Date: 24/08/00

Contents

1 Introduction

This is a very short description of the *whats* and the *hows* plus the *what-not's* and the *how-not's* of Use Case modeling.

There are some important aspects that is not described in this little introductionary description.

Abstraction level
You can have models on conceptual level describing what the system can do. You can have a concrete model describing how the system is going to be build. You can have a model on implementation level describing the system in implementation diagrams and source code. The Use Case model mainly belong to the conceptual level, but can sometimes be modeled into the concrete level. This abstraction level aspect is described in this document.

Traceability
The Use Cases do have a direct connection/relation to the requirements on one side and to the static and dynamic part of the conceptual model on the other side. The traceability aspect with concrete descriptions of how the relations is established and maintained is not described in this document.

Model organization
When you have a model with more than a few Use Cases, classes, sequences a.s.o. you can organize the model into packages. The modeling aspect is not described in this document.

Process
Use Case modeling is a part of a software development process. How the modeling fits into the overall process is not described in this document.

OK, what is described in this document then? Well the purpose of the document is to help you make as good Use Case models as possible. The descriptions should help you avoiding the most common mistakes. Description of the elements of a Use Case model, Actors and Use Cases, is found. A short description of how you model the Use Case model further is found. At last three common *major-bommers* are described.

2 What's a Use Case Model

A Use case model is a functional description of the system you're going to build. The model elements are Actors and Use Cases. The Use Case model consists of one or more Use Case diagrams and a description for each Actor and each Use Case. These descriptions can be kept in a separate document (one for each Use Case) or as a documentation attribute in the Use Case model. See ***Appendix B: Use Case description*** page 10.

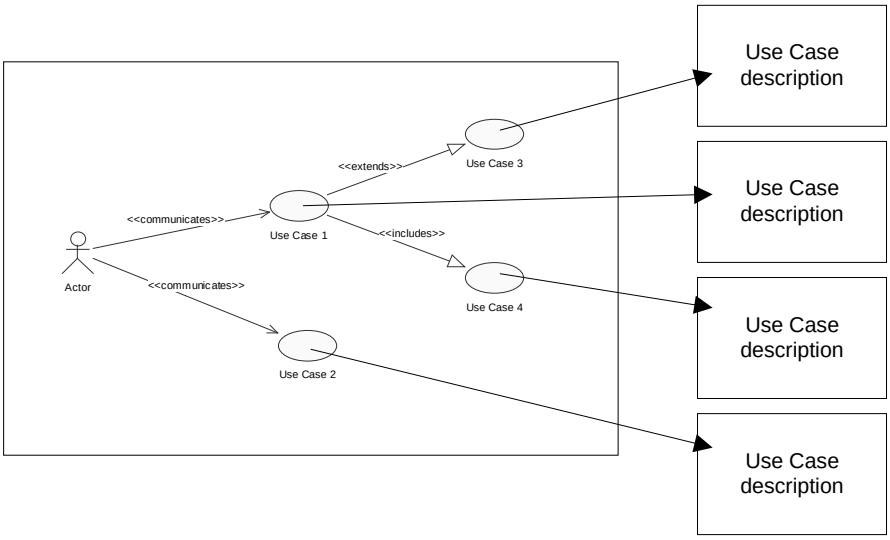


Figure 1 Use Cases and descriptions

A description of what you must describe for each Use Case is found in ***Appendix B: Use Case description*** page 10.

2.1 What's an Actor

An Actor is someone or something that interacts with the system (the application). An Actor can be e.g. a fieldtester or a mobile phone.

2.2 What's not an Actor

An Actor is not a concrete person, e.g. John Doe. An Actor is a User type.

2.3 How to find Actors

When you are looking for the Actors of the system, you can ask questions like the following:

- Who or what is using the system?
- Who or what gets support from the system?
- Who or what will maintain the system?
- Which hardware devices do the system need to handle?

Questions like these will give you candidates for Actors.

2.4 What's a Use Case

A Use case is a type of scenarios, a class of typical uses of the application. A Use Case is a complete functionality. An Actor initiates a Use Case. Specialized Use Cases (extends & includes) are initiated by other Use Cases. See ***Relations between Use Cases*** page 4.

2.5 What's not a Use Case

Parts of functionality like e.g. Messagebox popup, initializing COM port etc. Sub-functionality is not a Use Case but part of a Use Case.

2.6 How to find Use Cases

When you are looking for Use Cases you can ask questions like the following:

- Which functions/services (full functions) does Actors require from the system? What does the Actor need to do?
- Does the Actor need to read, create, destroy, modify or store some kind of information in the system?
- Does the Actor need to get notified when some events occur in the system?
- Can some Actors daily work be simplified by functionality in the system? If so what work?

2.7 Relations between Use Cases

You can have plain relations between Use Cases and between Actors and Use Cases. A relation like this means one Use Case activates another. The most common of these relations is an Actor that <<communicates>> with a Use Case.

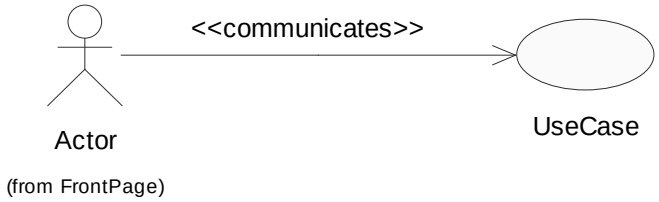


Figure 2 communicate relation

The <<communicate>>[1] stereotype indicates that the Actor communicates with the system, meaning the Actor uses some functionality described in the Use Case.

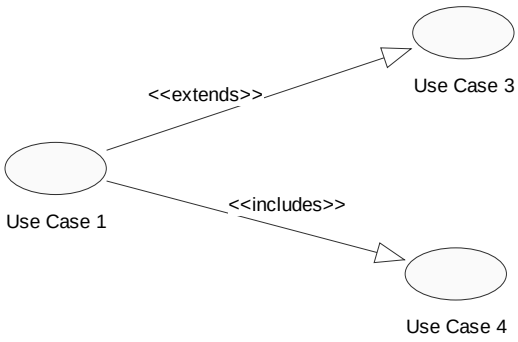


Figure 3 Extend and include specialization

When you do more detailed modeling on a Use Case model you can model common parts of Use Cases into new Use Cases (specialized Use Cases). These *sub-routines* can be of two different types;

[1] Stereotypes: In the diagrams you can mark elements as a specialized type. You mark these stereotyped elements with guillemets like: <<communicate>>

<<extends>> and <<includes>>. Observe that this relation between Use Cases is a specialization[2] (a line with a big hollow arrowhead).
When you are looking for Use Cases that filter out some common functionality and when you are going to label (stereotype) the relations with either <<extend>> or <<include>> you must understand two things.
First you must be able to extract the right functionality out of the original Use Cases. Second you must be able to know the difference between <<extend>> and <<include>>.
A Use Case is a type (class) of scenarios. Through there is one normal flow and some abnormal (alternative) flows.
If you have some part of these flows that are common for more than one Use Case, you can separate these common parts into new Use Cases that will be *reused* by the original Use Cases. When one Use Case *reuse* another Use Case it is marked with a specialization relation with the stereotype <<extend>> or <<include>>.

UML < 1.3	UML 1.3	Meaning
extends	extend	optional
uses	include	mandatory

2.8 How many Use Cases?

This is always the "1.000.000 dollar question". Many have tried to give a rule-of-thumb for how many Use Cases a good model contains. One of the better answers is that a Use Case corresponds to 6 - 12 man-months.
The amount of Use Cases depends on the application type. If you compare two application types like an intranet application and a technical application for measuring activities inside a mobile phone (a *phone-meter*) you will se that the intranet applications Use Case model will contain a larger number of Use Cases than the technical *phone-meter*.
The intranet application has many different user types (Actors) and it communicates with several other systems (Actors) like e.g. a mainframe system. There are a lot of different uses (Use Cases) of this intranet application.
The technical application, the phone-meter, only has one user type, the fieldtester, and it only communicate with the phone. The fieldtester a less different uses (Use Cases) of his/hers *phone-meter*.
If you end up with a Use Case model with way to many Use Cases you've probably been doing functional decomposition instead of Use case modeling. See ***Making functional decomposition*** page 7.

2.9 Further reading

Use Case modeling is described in the Rational Unified Process (RUP).
See Core Workflows -> Analysis & Design -> Activity Overview

[2] Specialization: This relation is normally used for showing inheritance. You must just accept that this relationtype is used for this extension/include relation. You can also send a mail to Ivar Jacobson and ask him about the rationale behind this choice.

3 Further modeling

3.1 Detailed Use Case design

The Use Case model is primarily an analysis model. It's a structured way of describing the functionality of the system you're going to build. I certain situations it can be valuable make a detailed analysis of the Use Case model. In this detailed analysis you typically look at extension points and specializations of the Use Cases.

3.2 Describe normal and alternative flows

In the model you describe each flow through the Use Case in a sequence diagram. You describe both the normal flow and the alternative (ab-normal) flows. An Actor often initiates the sequence diagrams.

3.3 Finding Domain classes

Domain classes emerge during the sequence modeling. The sequence diagrams shows how classes and Actors work together. The Actors is automatically candidates for classes.

3.4 Categorizing Domain classes

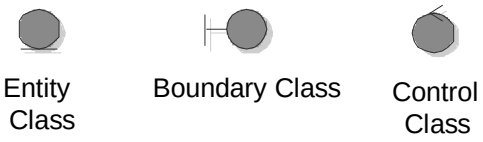


Figure 4 Stereotypes for domain classes

After having found domain classes during sequence modeling you can categorize these domain classes into three categories: Entity, Boundary and Control classes. You mark the classes with stereotypes[3].

Entity class	Hold information (datamodel) and associated behavior. You typically have several entity classes pr. Use Case. This class type corresponds to the Model classes in a Model-View-Controller pattern or the Abstraction (inner) class in the Presentation-Abstraction-Controller pattern.
Boundary class	Handles communication between classes. You typically have one boundary class pr. Use Case. This class type corresponds to the View classes in a Model-View-Controller pattern or the Presentation (inner) class in the Presentation-Abstraction-Controller pattern.
Control class	Controls behavior. You typically have one control class pr. Use Case. This class type corresponds to the Controller classes in a Model-View-Controller pattern or the Controller (inner) class in the Presentation-Abstraction-Controller pattern.

[3] stereotypes: All model items in UML can be stereotyped. This means that you added a certain meaning to the entity. Classes can be categorized.

4 Common flaws and mistakes

4.1 Making a dataflow diagram

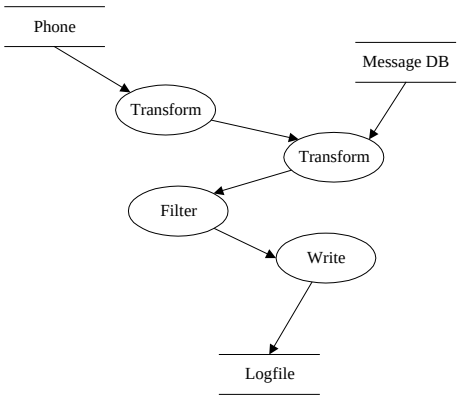


Figure 5 Dataflow diagram

In Structured Analysis[4] you did make diagrams showing how data flows through the system. The boxes are datastores, the ellipses are processes and the arrows are data flow.
It is important that you do not mistakenly do this dataflow modeling when you should do Use Case modeling.
In other words, a Use Case diagram do not show how data flows through the system. The Use Case model/diagram shows the functionality of the system.

4.2 Making functional decomposition

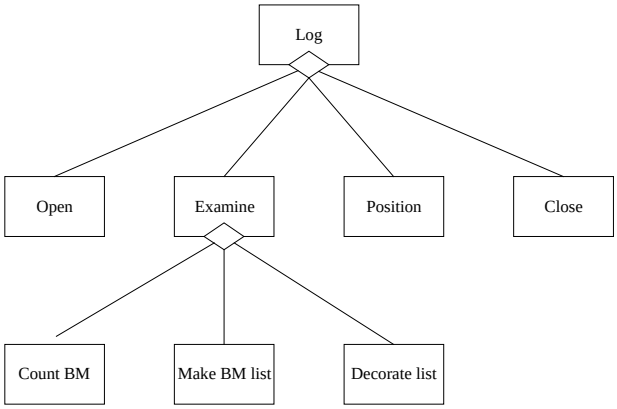


Figure 6 Functional decomposition

In Structured Analysis you handled complexity by decomposing functions into sub-functions. This was/is done in a functional decomposition diagram.
It is important that you do not mistakenly do this functional decomposition when you should do Use Case modeling.
In other words, a Use Case diagram do not show a functional decomposition of the system. The Use Case model/diagram shows the functionality of the system.

Back in the good old days of structured analysis it was a typical mistake to do data flow modeling when you should do functional decomposition and vice versa.

[4] Structured Analysis: Popular method (the religion) for software development in the late 1970'ies and the early 1980'ies.

4.3 Making too detailed diagrams

It is a very common mistake to make Use Case diagram way too detailed. There are multiple sources to this mistake.
If you are doing functional decomposition and you are consequent you will end up the whole system decomposed into tiny functions. The same thing happens if you mistakenly get focused on modeling the dataflows in the system.
Another source of errors is if you forget the rule that says that a Use Case must be a complete functionality. Every time you break this rule you end up with small incomplete functional pieces.
A nasty biproduct of this mistake is that you get a mountain of dependencies between the Use Cases making the model almost impossible to maintain.
The last source of errors is making the descriptions of the Use Cases too detailed because you describe the implementation. This means that your Use Case descriptions contains the *how's* and not the *what's*. If you find yourself writing the exact prompts, error messages, describing dialogbox layout a.s.o. Then you are on a wrong track and you must punish yourself.

5 Appendix A: Actor description

The Actors is fairly easy to describe. Actors have a lot in common with classes and can sometime end up as classes in to system or as domain classes that do not go into the design model. Domain classes normally do, but not always.

Name
The name of the Actor.

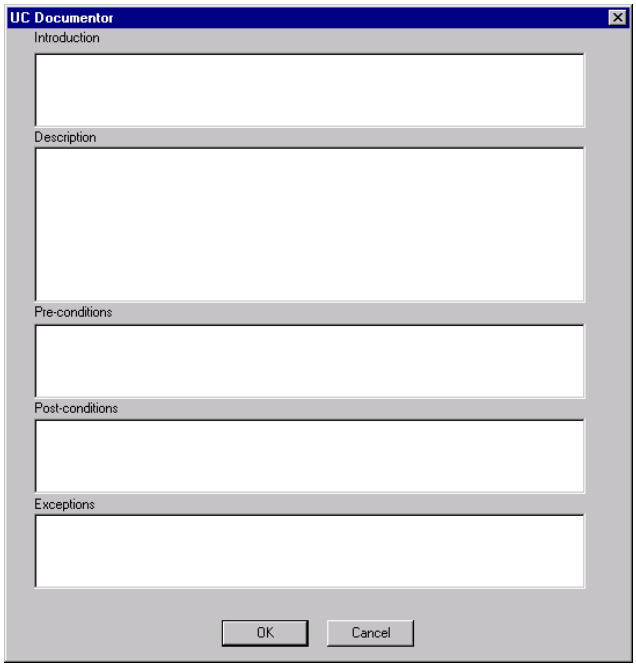
Actor type
Is the Actor a user type or a device that communicates with the system.

Role
The role the Actor plays in the system

Use Case relations
Which Use Cases do the Actor have relations to.

Actor relations
Relations to other Actors. This can be a specialization. If you have sevaral user types, and these do have some communality, the common thing can be generalized into a super type from which the specialized Actors can inherit.

6 Appendix B: Use Case description



Introduction
Short introduction to the Use Case.

Description
Describe the normal flow through the Use Case.

Pre-conditions
Which conditions must be meet before the Use Case begins.

Post-conditions
What is guarantied after the Use Case ends.

Exceptions
Describe the alternative flow through the Use Case.

Figure 7 Description of Use Cases

Figure 7 Description of Use Cases shows a dialog from an extension to Rose. You can use this to make descriptions for each Use Case. The description are placed in the documentation attribute of the Use Case model element. An alternative is to make a document containing Use Case description.

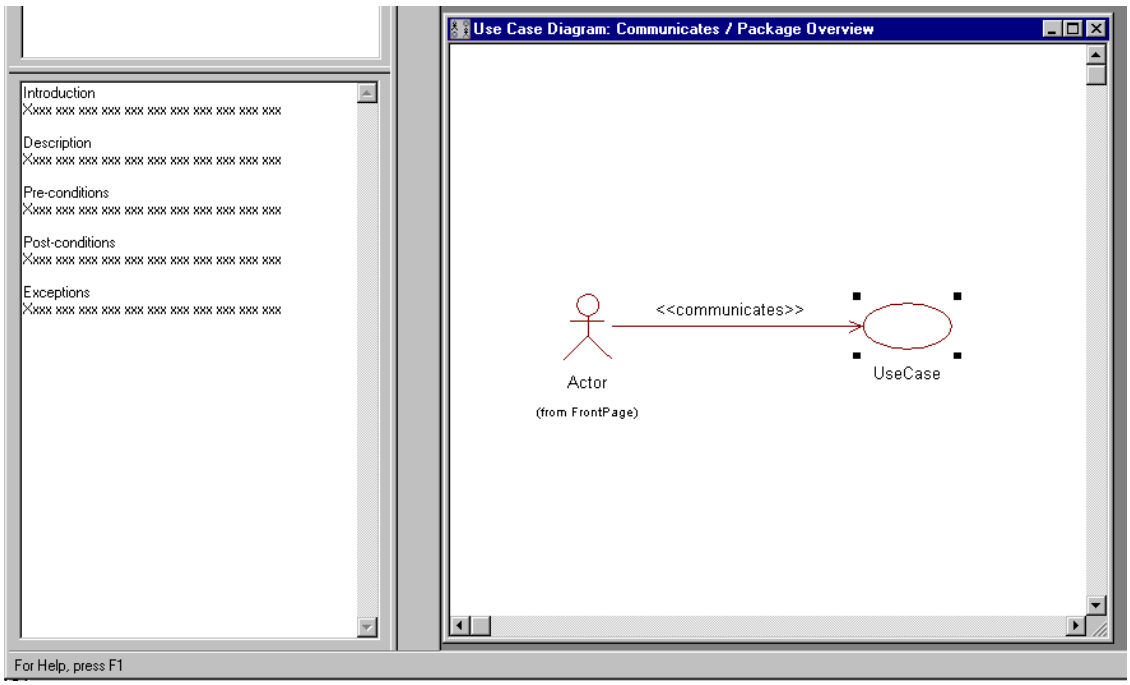


Figure 8 Documentation window in Rose

7 Appendix C: Use Case flows

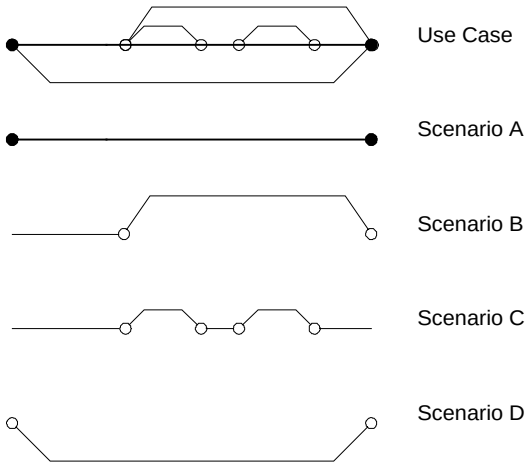


Figure 9 Use Case flows

Figure 9 Use Case flows shows an example of a normal and some abnormal (alternative) flows through a Use case.
Scenario A is the normal flow through the Use Case, everything goes normal it's a straight flow.
Scenario B to D is abnormal flows. At some point (small circles) an exception occurs and some alternative route is taken.
The informal diagramtype can be usefull when mapping out all the flows through a Use Case.