Chapter

72 *System Analysis*

This chapter gives a thorough description of the different models in the system analysis activity as well as some guidelines on how to create these models. A recommendation on consistency rules that are relevant for the models in this activity and for the consistency between the models from the previous activity and this activity, is also included.

System Analysis Overview

While the purpose of the requirements analysis is to understand the problem to be solved and the requirements this puts on the system, the purpose of the system analysis is to understand the architecture of the system itself. Essentially the issue of the system analysis is to find out what objects are needed to implement the requirements on the system. This means that the system analysis to a large extent is an analysis of the information that is needed to be represented in the system and the structure of the system itself. Information is here used in a broad sense which includes not only the data to be manipulated in the system but also the containers for algorithms and interfaces.

The system analysis in SOMT is very similar to corresponding activities in other object-oriented analysis methods and the major input and outputs of this activity are illustrated in Figure 635.

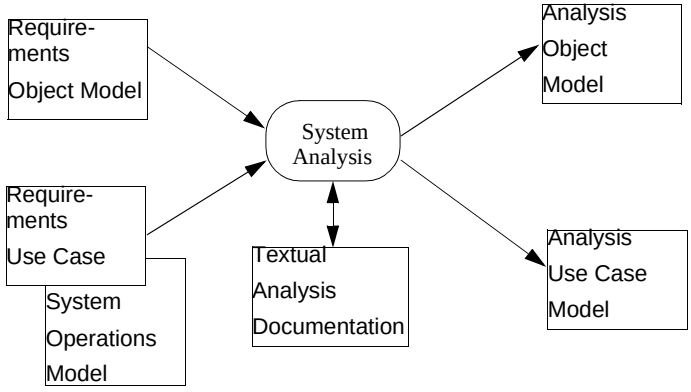

Figure 635: The major inputs and outputs of the system analysis activity

The main input to the system analysis is the requirements object model and use cases developed during the requirements analysis and the main output is another object model, the analysis object model that describes the logical architecture of the system. In addition to this model a use case model is also created in the system analysis to describe the dynamic aspects of the architecture and textual analysis documentation is used to document analysis results not suitable to be express as use cases or object models. The different models will be discussed in detail in the following sections, the analysis object model in "Analysis Object Mod-

el" on page 3723 and the use cases in "Analysis Use Case Model" on page 3735.

The tasks to perform in the system analysis activity are thus essentially the following:

1. Start defining an initial version of the analysis object model, in particular concentrating on creating an overall architecture of the application.
2. Then start refining some of the most important requirements use cases to check that the architecture defined in the analysis object model will work.
3. Continue by iterating between modifying/refining the analysis object model and creating more analysis use cases, either by refining requirements use cases or by describing particular mechanisms in the application.

In parallel with the tasks above it may also be necessary to study various aspects of the chosen architecture, e.g. with respect to non-functional requirements. These results can be documented in the textual analysis documentation.

Analysis Object Model

The intention with the analysis object model is that it is a means to describe the architecture, i.e. the main objects that need to be implemented in the completed system. The notations used are the same as for the requirements object model in the requirements analysis. An overview of the notations is given in "Object Model Notation" on page 3670 and in "State Chart Notation" on page 3674.

At a first glance the requirements object model in the requirements analysis and the analysis object model in the system analysis seem similar but there are several reasons to distinguish between them. The major motivation is that the purpose of the models are different: The purpose of the requirements object model is to investigate and describe the problem that the system is to solve and the environment that the system is to operate in, while the purpose of the system analysis model is to analyze and define the architecture of the system itself. Another more pragmatic difference is that the requirements object model consists of objects visible on the border of the system and outside the system, e.g. users of the

system, while the system analysis object model is focused on the internal object structure of the system.

In the same way as the requirements object model can be structured into a number of different diagrams, the analysis object model can naturally also be decomposed into more than one diagram. This is even more important than it was for the requirements object model since the analysis object model tends to be much larger than the requirements object model.

The Logical Architecture of the System

The major purpose of the analysis object model is to describe an object structure that defines the logical architecture of the application. It describes how the application at a certain level of abstraction can be considered to be divided into a number of subsystems or objects that together fulfil the requirements posed on the application.

For each class that is identified in the logical architecture the most important issue is to note the responsibilities of this class. Why is the class included into the architecture? What is it supposed to do? The responsibilities of an object of a specific class are described by answers to a set of questions:

- What information or knowledge is the object responsible for maintaining?
  - This is described by the attributes of the class.
- What should the object be able to do?
  - Described by the operations of the class.
- What other objects does the object need to know to fulfill its responsibilities?
  - Defines associations and aggregations to other classes.
- Are the responsibilities of objects of this class similar to that of other classes? Is it “the same as class ... except that...”?
  - Defines inheritance relations.

The result when identifying different objects and answering the questions above is one (or more) class diagrams that describe the architecture of the system. The responsibilities are described by the operations

and attributes. The associations between the classes in the model represent needs for objects to be aware of other objects to be able to fulfil its task. As an example, consider Figure 636 that describes the architecture of the access control system.

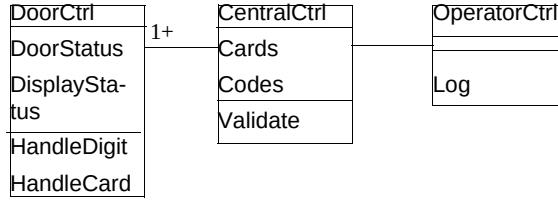

Figure 636: Object model diagram describing the system architecture

Note that object analysis model can be viewed as a refinement of the context diagrams in the requirements object model.

Finding the Objects

The objects in the analysis object model can come from several different sources. Some useful examples are:

- objects from the requirements object model
- objects from interfaces
- objects from use cases

These different sources are described in the following sections.

Requirements Object Model as Source of Objects

The requirements object model is of course one of the major sources of objects for the analysis object model, in particular for the information modeling part. Since the requirements object model should contain most of the objects in the problem domain a lot of them will probably have to be represented in the system and should thus be part of the analysis object model. Note however, that in this case it is not the same object that is found in the requirements object model and the analysis object model. The object in the analysis object model is in this case a container of information about the “real” object that is modeled in the requirements object model. In many cases of course entire inheritance and/or aggregation hierarchies can be reused in the analysis object model as illustrated in Figure 637. Notice that the *Guard* object is not need-

ed to be represented in the system and thus is not introduced in the analysis Object Model.

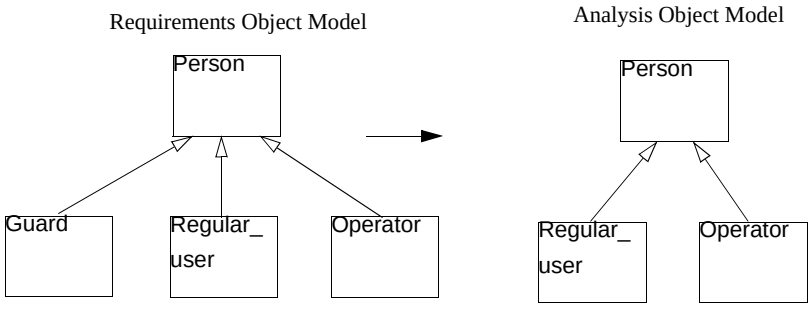

Figure 637: Reuse of requirements object model in the analysis object model

The objects found in the requirements object model usually have one thing in common, they represent entities in the real world that the system needs to store information about.

An algorithm to find the information objects needed by the system based on the requirements object model can thus be phrased as follows.

- For all objects in the requirements object model:
  - Decide if the system needs information about this object to fulfil its task.
  - If the answer is “yes” then add it to the analysis object model using *Paste as* to get the implinks and thus provide traceability back to the requirements object model.

Objects from Interfaces

Another useful way to find objects is to consider the interfaces that the system will have to the environment. It is often very useful to introduce a special kind of object that hides the specific features about how to access the interface from the rest of the system. In [18] these kind of objects are called *interface objects*. Where do we get the interface objects from? There are several different sources to search:

- The application area itself, which in some cases make it obvious what interfaces must exist in the system.

- The use cases from the requirements analysis can be searched for interface objects:
  - They may explicitly identify some interface, e.g. “The user enters a card into the card reader”.
  - Since they define actors that communicate with the system, each actor must use some kind of interface when interacting with the system, even if it is not stated explicitly which interface that is used. So by starting with the actor we may analyze what interfaces he/she will need. If nothing else the actor itself may introduce an interface object.

One important motivation for introducing interface objects is to make modification of the system easier. If the hardware of an interface is changed, e.g. the card readers of an access control system, then the logic of how to handle them is encapsulated in one object. This makes it likely that this object is the only thing that needs to be changed in the software.

Performance requirements are another motivation to introduce interface objects. Very often the interfaces can be a bottleneck with respect to performance. By encapsulating the interface in one object providing high-level operations to the rest of the system it is possible to make an optimized implementation of this object, e.g. making special purpose hardware or enhance the performance.

As an example consider the access control system and specifically the *Enter building* use case described in text in Example 603 on page 3705. From this text we can directly find a number of interfaces: a card reader, a display, a door lock, etc. All of these are likely candidates to result in interface objects in the analysis object model. When describing the interface objects it is usually fruitful to use a communication style class diagram to show how they interact with the external actor and the relevant objects in the analysis object model. If the interaction is non-trivial it might also be a good idea to show the interaction pattern using one or more MSC use cases that describe the different ways this particular object interacts with its environment.

Objects from Use Cases

The use cases can be used as a tool to find the objects that are needed. The strategy is to take a use case and investigate how the functionality that is implied by the use case is distributed among the objects in the analysis object model. One way to do this is to produce an MSC that describes the interaction as described in "Analysis Use Case Model" on page 3735. Check which interface objects are involved, which internal objects are modified or accessed and consider the question of introducing a special controller object to encapsulate the sequencing of the use case. Is there already a control object that might take on the responsibility, or is there a need to create a new control object to handle the logic of the use case?

Consider for example the *Enter building* use case described textually in Example 603 on page 3705 and using an MSC in Figure 628 on page 3707. Since this use case contains a sequence of steps the system needs to represent that essentially has to do with the state of one of the doors and the associated lock and other devices, a special control object *DoorControl* seems natural to introduce. This could give an object model diagram as in Figure 638.

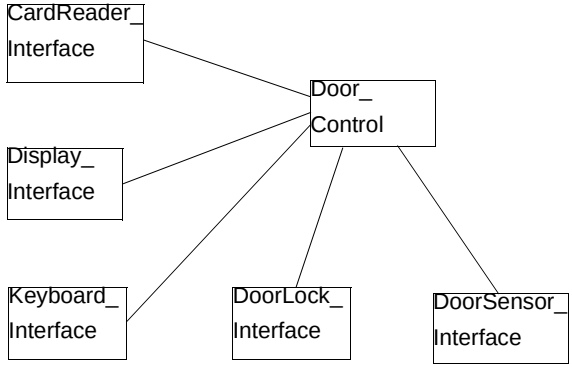

Figure 638: An object model diagram that introduces the DoorControl object

Notice that the objects in this object model correspond to the interface objects that more or less directly could be extracted from the use case text together with the new control object. It may in this context be useful to name the objects according to their function. The interface objects are

in this example called “XXInterface” and the control objects “XXControl” or something similar.

Notice that this model also introduces associations between the interface objects and the control object. These associations represent the communication paths that are needed among the objects. In one way or another information will flow following these associations.

Another type of object that might be found when analyzing the use cases are DataServer objects. These objects define the access possibilities to (complex) data structures. There are several possible sources to search for these type of objects:

- The object structures that come from the requirements object model represent information that has to be stored in the system. In many cases there is a need for a DataServer object that “owns” this information and that provides an access to it. So, investigating the objects from the requirements object model and how they are to be used is a way to identify DataServer objects.
- A second source of information is given by the use cases, since they often express the need for some kind of computation that involves several, related objects. When there is a need for a more complex algorithm that operates on an object structure a DataServer object might be useful.

By once more analyzing the *Enter building* use case described in Example 603 on page 3705 we can see that there is a need for a checking mechanism that determines if the code a user enters is the correct code associated with the card he previously has entered. This indicates the need for a *CardAndCodeDataServer* object that is responsible for maintaining the information about cards and code. Furthermore we can see that a likely operation on this object is a *Validate* operation that tells if one particular combination of card and code is correct. This may give an object model as in Figure 639.

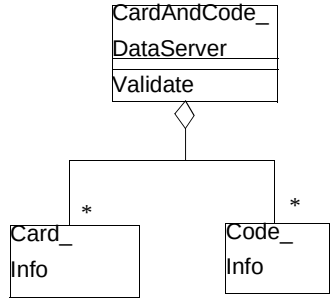

Figure 639: An object model diagram introducing the CardAndCodeDataServer object

Finding Attributes and Operations

As discussed above the attributes and operations are used as a means to describe the responsibilities of objects. They describe the purpose of introducing an object into the model by answering the questions:

- What should the object be able to do?
- What information or knowledge is the object responsible for maintaining?

In practise, the attributes can be found for example:

- In the use cases
- In the requirements object model (by keeping already described attributes)
- In the textual requirements

Some useful sources of operations are:

- The requirements object model (keeping the existing operations)
- The analysis use case model (the messages in the MSC diagrams)

The MSC messages in the analysis use case model can often be considered for operations in the analysis object model. Consider the behavior patterns in the use case model as well, since they often describe the functionality on a more detailed level.

Finding Associations

As discussed above the associations are used to show how object of one class need to know other objects. Usually the associations are found when analyzing the responsibilities of the classes since the motivation for introducing an association is that it is needed by a particular object.

However, some other sources where it is useful to look for the associations are:

- The requirements object model (preserving or modifying existing relations)
- The textual requirements
- The analysis use case model

In particular the last source, the use cases are important. The activity of finding associations in the analysis object model and the activity of constructing the analysis use case model are closely related.

Describing Object Behavior

Sometimes it is useful to describe the behavior of the objects presented in the Logical Architecture. The Analysis Use Case Model describes how objects of the Logical Architecture interact. State Charts describe how these objects reacts internally as a result of the interaction presented in the Analysis Use Case Model.

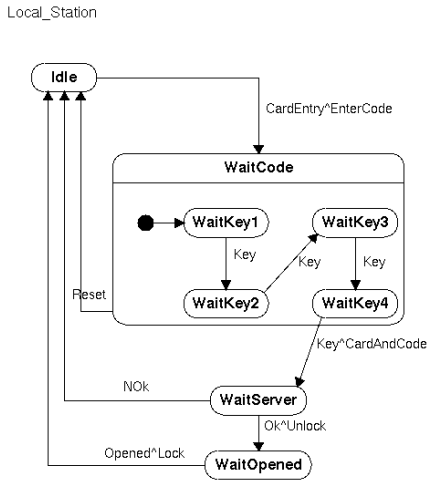

Figure 640: A state chart describing the behavior of an object

It is not necessary to describe the behavior of every object in the Logical Architecture. The focus should be on objects with complex behavior and where complex data structures are dependent of the state of their object. Modeling the behavior of an object will lead to a better knowledge of how the object will function internally. “Which are the actions of the object?”, and “what data structures are needed?” are some questions that may be answered. If an object is too complicated it might be necessary to divide its class into several smaller classes which should be reflected in the corresponding class diagram. In other words: the relationship between the Logical Architecture diagram and the Object Behavior diagram is bidirectional.

When creating a state chart, the Logical structure is the obvious source of information. It is necessary to consider the object’s class specification when specifying its behavior. An operation on a class will lead to transitions in the behavior model. A transition may lead to an internal action or a sending of an event to an external object. A transition may also lead to value changes of the object’s attributes. If these attribute changes are important they should be reflected in the state chart. The operations on the class may also be found in the Analysis Use Case Model together with the possible sending of events.

Architecture of Large Systems

For large applications it is also often necessary to divide the analysis model into more than one module, e.g. to facilitate an analysis by more than one team. One possible strategy for doing this is a recursive approach where we first make an architecture on a high level, where each class represents a subsystem. Then each subsystem is refined by a separate team. It is important to be vary careful about the responsibilities of the classes in the top-level architecture, as described by their attributes and operations, since they will form the input to the different analysis teams.

In this context it is useful to use aggregations to describe a “subsystem” or “is-composed-of” relation, i.e. to describe how the system is decomposed into parts that recursively are decomposed into smaller parts. As an example, consider Figure 641 that describes the structure of the access control system. In this example the system is composed of one or more DoorCtrls, one CentralCtrl and one OperatorCtrl. The OperatorCtrl is itself composed of a component handling remote communication (RemoteCom) and a UserInterface.

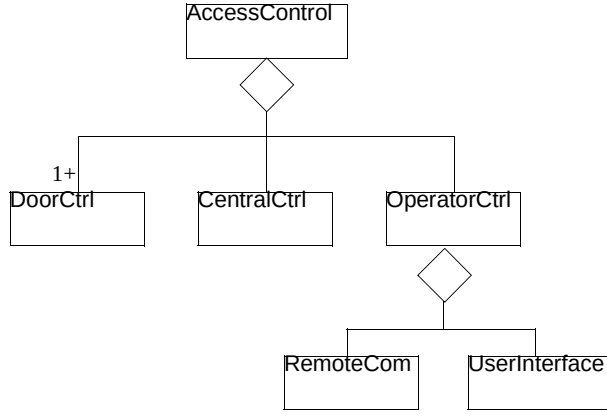

Figure 641: Using aggregations to describe the access control system

Note that the usage of aggregation as a means to describe subsystem relations has an impact on the object structure. Since the assembly classes are used to represent aggregation of classes and not single objects, they should in general not have any “intelligence” by themselves. This implies that the operations provided by the aggregate classes in practise

should be implemented by one or more of the parts classes that the aggregate class contain.

It is a good practise to make the decomposition into modules aligned with the top-level architecture. One top level module architecture of the upper level of the application, identifying the responsibilities of the subsystems. The leaf classes of this top level architecture are then defined in separate modules possible by different analysis teams. The top level module then forms an interface or contract between the analysis teams.

It is impossible to give any rules for how to find the best way to divide a system into subsystems, but most authors agree on some measures to tell if a certain decomposition is a good one. The following two rules are from [5] describing OMT guidelines for decomposition interpreted in the SOMT context:

- The structure must be designed so that most interactions are within the subsystems and not across the boundaries.
- A system should not be divided into too many subsystems, 20 is probably too many. It is better to use a hierarchy of subsystems instead.

The subsystems may be chosen based on several different approaches, but the major idea is to group together objects, that together provide a certain function to the rest of the system, into a subsystem which then can be used as an abstraction of the entire group of objects.

Analysis Use Case Model

The purpose of the analysis use case model is to show the dynamic view of how the functionality of the system is decomposed in the same way as the analysis object model describes the static view of the decomposition. This implies that the internal communication between the various parts of the system is the major concern for this activity. Two different types of use cases can be distinguished in the analysis use case model:

- Refined requirements
- Behavior pattern use cases

Refined Requirements

The analysis use cases that are refined requirements are simply the use cases from the requirements analysis refined to the analysis object model level. Each requirements use case is distributed among the objects from the analysis object model. The purpose of these use cases is mainly to document how the logical architecture as described in the analysis object model is capable of implementing the requirements that are expressed by the use cases. In practice this is done by taking each of the use cases defined in the requirements analysis and reformulate it in terms of the objects that are defined in the analysis object model. It is possible to use both the textual use case notation and the MSC notation to represent the use cases, but since the purpose of the analysis use case model is to show how the functionality is distributed among the objects the MSC notation is especially useful. As in the requirements analysis, HMSCs can be used to simplify the use case model.

Consider again the access control system with a logical architecture according to Figure 636 on page 3725 where the system is divided into a DoorCtrl, a CentralCtrl and an OperatorCtrl components. If we take the *Enter building* use case as defined by an MSC in Figure 628 on page 3707 and replace the system with the three components we get an MSC as shown in Figure 642.

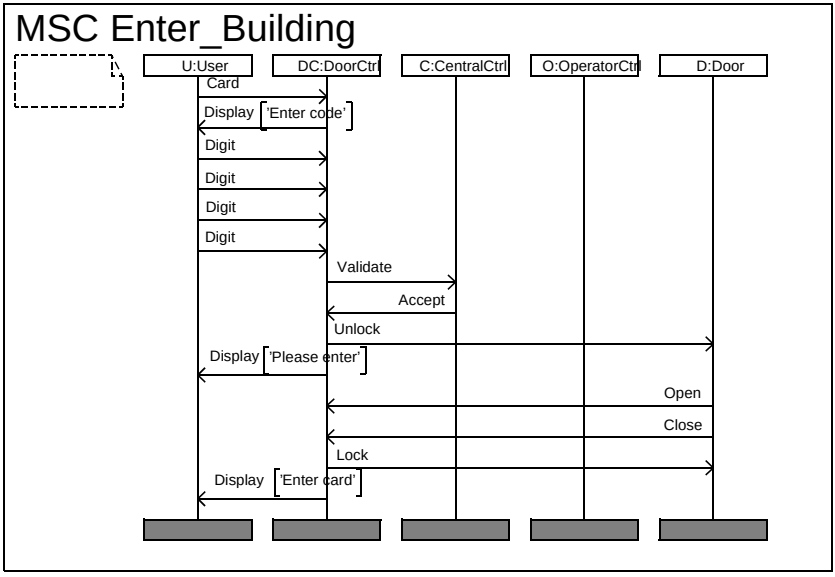

Figure 642: The Enter Building use case distributed over a logical architecture

In this MSC we can see that the original system instance is replaced by instances of DoorCtrl, CentralCtrl and OperatorCtrl. This use case deals mainly with the handling of user interaction at the door and since this is the responsibility of the DoorCtrl most of the action is performed by this component. Only the validation of the card and code is performed centrally.

It is easy to see that the strategy outlined above is extendable to allow a decomposition of a system into not only one level of components, but into a hierarchy of components. For each new level of decomposition all use cases that involve the decomposed components are taken as input to the validation of the new decomposition. The component that was decomposed is replaced by the new components and new versions of the use cases are created.

Behavior Patterns

When designing the object structure for a system there is often a need to document behavior patterns that involve one or more objects that participate to fulfill a common objective. Sometimes this can be described in the refined requirements use cases, but there are two advantages of creating special use cases for specific mechanisms and behavior patterns:

- By describing detailed communication patterns in special use cases, the refined requirements use cases can be on a higher level of abstraction and are not made unnecessary complex.
- By focusing special use cases on specific parts of the system, it is easier to understand and maintain the requirements of the involved object than if these where distributed among all the refined requirements use cases.

As an example consider the keyboard interface of the access control application. The purpose of the keyboard is to allow the user to enter a code consisting of four digits. In a refined requirements use case showing a user entering an office by pushing the digits on a keyboard this can be shown using an MSC reference symbol, as in Figure 643.

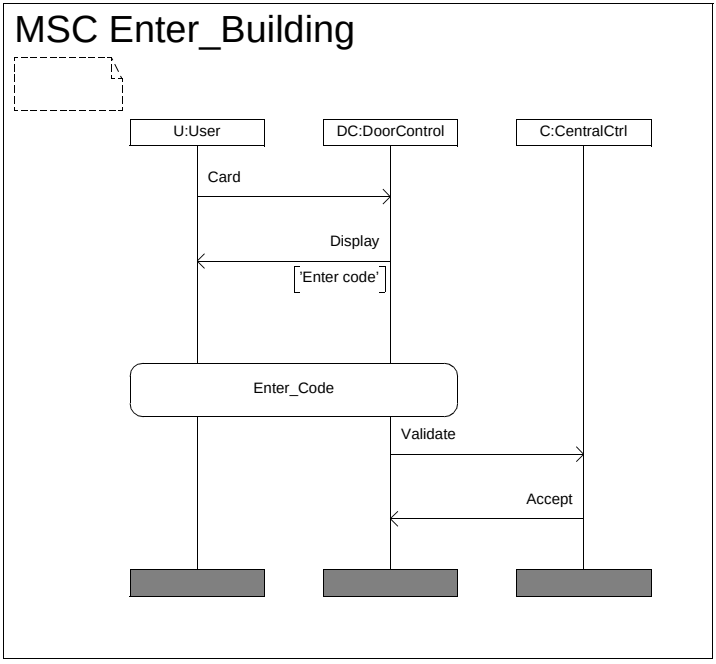

Figure 643: Part of a refined requirements use case

What happens when the user enters a code is thus described in the MSC Enter_Code. We see that it is the user and the DoorControl object that are involved in this interaction.

However, there is of course a specific protocol for Keyboard objects that defines how they interact with the user and the DoorCtrl that is not shown on the abstraction level of the refined requirements use case. This may for example look like in Figure 644, which shows the behavior pattern use case Enter_Code.

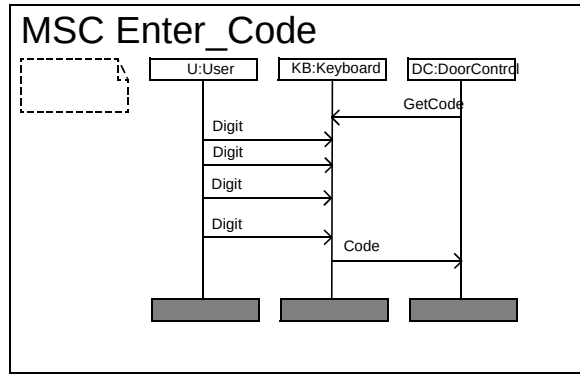

Figure 644: A behavior pattern use case showing the protocol for Keyboard interface objects

Other examples of behavior patterns that may need special use cases are internal communication between different parts of a system using proxies (i.e. local dummy objects that are used instead of a remote object and hides the communication aspect from the user of the proxy), initialization protocols that describe how various objects are created and exchange information, and in general all other tactical solutions to problems that need a special explanation in the system analysis.

Textual Analysis Documentation

In many cases there is a need to express aspects of the architecture or architecture related requirements in a textual format as a complement to the more structured object models and use cases. This may for example be to record experiments performed to check performance aspects of the architecture or other non-functional requirements. Other examples may include documentation of risk assessments performed in the system analysis phase.

To satisfy this need SOMT allows different textual documents to be included among the analysis documentation.

Requirements Traceability

One important aspect in the system analysis is the relation between the models that are created in this activity and the requirements, both external requirements specifications and the models from the requirements analysis. Important questions are:

- Have all requirements been implemented?
- Which are the system analysis objects that implement one particular requirement?
- Which are the requirements that are implemented by one specific object in the analysis object model?

To be able to answer this type of questions it is important to create and maintain descriptions of the dependencies between concepts among the requirements and system analysis concepts. As discussed in "Implinks and the Paste As Concept" on page 3666 the means to do this in SOMT is given by what is called implementation links (or implinks for short). An implink is an association between two concepts where one of the concepts implements the other. One example is the implinks that exist between objects in the requirements object model and the corresponding object in the analysis object model. Consider for example Figure 637 on page 3726 that illustrates how domain objects are modeled in the analysis object model. In this case there should for example be an implink between the *Operator* object in the requirements object model and the *Operator* object in the analysis object model.

Another example is the links between use cases on different levels. Links between the requirements use cases and the analysis use cases show that the requirements are handled on the system analysis level.

In particular when doing "what if..." analysis of the consequences a modification or extension of the system has, these type of links are invaluable. For example consider the case where we would like to specialize the concept of an *Operator* into *RegularOperator* and *ChiefOperator* where the chief operator has some special privileges that a regular operator does not have. We then look at the requirements object model and try to understand what consequences this modification will have. With an implink we immediately see that the analysis object OperatorInfo will have to be changed and with further links from this object we can get a good idea of the consequences caused by the modification.

Consistency Checks

This section provides a list of consistency checks that can be made on the models produced by the system analysis.

- Check that all use cases from the requirements analysis have been refined to analysis use cases.
- Check that all entities in the requirements object model are either represented in the analysis object model or not really needed by the application.
- Check that the object model diagrams and MSCs conform to the static rules for each notation.
- Check that the instances in the MSCs correspond to classes in the object model or to actors that interact with the system.
- Check that the messages received by the instances in the MSCs correspond to operations on the corresponding classes. Note that for remote procedures there may be two messages in an MSC for one operation, one message for the request and one for the reply. In this case the reply message should not have any corresponding operation.

Summary

The system analysis is an activity that is focused on understanding the system to be built. The major tools to facilitate and document the understanding of the system are the analysis object model and the analysis use case model. The analysis object model is intended to capture the objects that are needed in order to describe a solution of the problem.

The analysis use case model describes how the objects in the analysis object model cooperate to fulfill the requirements posed on the system by the use cases from the requirements analysis.

Chapter 72 System Analysis