Chapter

78 *SOMT Tutorial*

This tutorial is intended to present how to combine object-oriented analysis and SDL design in practise in a development process. This is a method developed by Telelogic, known as the SOMT method, *SDL-oriented Object Modeling Technique*.

We will demonstrate, using an Access Control system as example, the various activities and models in SOMT together with the provided tool support for SOMT in Telelogic Tau.

Through the tutorial you will practise on various exercises that will get you familiar with the SDL suite tools as well as the SOMT method.

Introduction

Purpose of This Tutorial

This tutorial presents how to use the SOMT method and Telelogic Tau in practise in a design process.

The working example is an Access Control system. The system shall control the entrances to an office. Each employee working in the office has a card and a personal code. To enter the office, the employee enters a card into a card reader and types a personal code on a keypad. To exit the office the employee presses an exit button.

You will perform the development process for the Access Control system applying the SOMT method. The tutorial will guide you through the development process step by step presenting a number of hands on exercises for you to perform. The tutorial is expected to be read sequentially.

After reading the tutorial, you should have gained knowledge about how to apply the SOMT method on a development process.

Note: Platform differences

This tutorial, and the others that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running Telelogic Tau on your platform. Only if a screen shot differ in an important aspect between the platforms, two separate screen shots will be shown.

Required Skills

It is assumed that you have a basic knowledge about UML and SDL. We also recommend newcomers to acquaint themselves with the basic features of the SDL suite tools. You can do this by practising on the exercises in the tutorials provided for the different tools. Please see the previous chapters in this volume.

It is recommended that you have read the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual.

Preparations

1. Make a new empty directory of your own for the purpose of this tutorial, e.g. `~/somttutorial` **(on UNIX)** or `C:\Telelogic\SDL_TTCN_Suite4.5\work\somttutorial` **(in Windows)**.

2. Copy the SOMT tutorial directory and its subdirectories in `$telelogic/sdt/examples/somttutorial` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\somttutorial` **(in Windows)**, into this new directory (contact if necessary your system manager).

> **Note: Installation directory**
>
> **On UNIX, the Telelogic Tau installation directory is pointed out by the environment variable** `$telelogic`. If this variable is not set in your UNIX environment, you should ask your system manager or the person responsible for the Telelogic Tau environment at your site for instructions on how to set this variable correctly.
>
> **In Windows, the Telelogic Tau installation directory is assumed to be** `C:\Telelogic\SDL_TTCN_Suite4.5` **throughout this tutorial.** If you cannot find this directory on your PC, you should ask your system manager or the person responsible for the Telelogic Tau environment at your site for the correct path to the installation directory.

3. **On UNIX**, `cd` to your own subdirectory `somttutorial`

4. Start Telelogic Tau.

5. Specify the source directory for the system by double clicking on the *Source directory* symbol located second uppermost in the Organizer window. The source directory specifies where new documents that you have created are saved by default, and from where to read when opening and converting documents. Since there are multiple versions of the Access Control system, each version with diagrams stored on files with identical names (but in different directories), omitting to specify the source directory may cause the wrong version of a file to be opened.

6. In the Set Directories dialog that is opened, select the third radio button associated with *Source directory*. In the text field, enter the complete path and name of your own `somttutorial` directory, if it is not there already. Press *OK* to close the dialog. (You do not have to change any of the other options in this dialog.)

Preparing the Documentation Structure

What You Will Learn

- To prepare a SOMT project by making preparations in the Organizer

Introduction to the Exercise

Your task is to modify the *basic view* of the Organizer to get the desired document structure.

The result of the exercise will be an Organizer structure containing a number of *chapters* and *modules*, see Figure 713 on page 3842. The chapters will correspond to the different activities in SOMT and the modules will correspond to the models in each activity.

Deleting Unwanted Chapters

When you start a new project with Telelogic Tau you will get the default *basic Organizer view*, see Figure 711. (This view could be different if you do not have the default preferences set). The Organizer contains a few black lines with attached names, the *chapters*. The purpose of these chapters is to group together collections of documents.

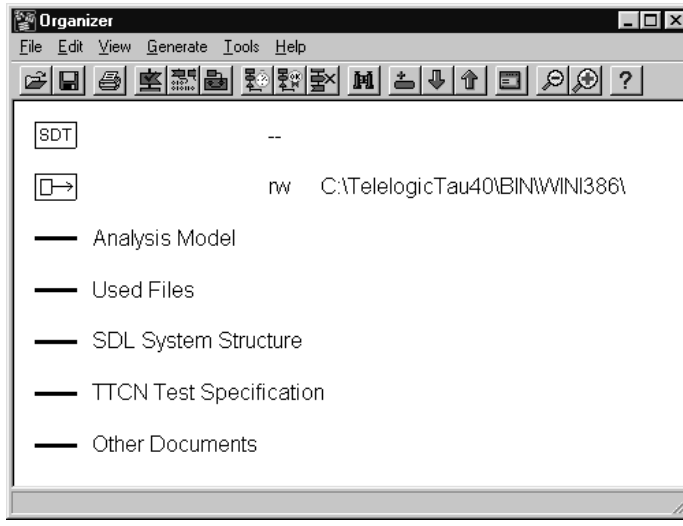

Figure 711: The basic Organizer view

We want each chapter in the Organizer view to represent an activity in SOMT. The current chapters in the Organizer will not fit into our future documentation structure so they should be removed.

Delete the unwanted chapters by following the steps below:

1. Make sure you have the basic view in the Organizer.
2. Select the chapter named `Analysis Model`.
3. Select the *Remove* command in the *Edit* menu or press the `<Delete>` button. You also find the *Remove* command in the pop up menu. The Remove dialog is issued asking you to *Remove* or to *Cancel* the action.
4. Press the *Remove* button. The dialog disappears and the chapter is deleted.
5. Repeat the steps above and remove all of the remaining chapters.

Adding New Chapters

You should now organize the Organizer view into chapters corresponding to the different activities in the SOMT method, i.e. each chapter should contain documents and diagrams from one particular activity.

You will have to add four chapters and they will be named `Requirements Documents`, `System Analysis Documents`, `System Design Documents` and `Object Design Documents`, respectively.

First, add the `Requirements Documents` chapter:

1. Select the *Add New* command in the *Edit* menu. The Add New dialog arises with the *Organizer* radio button set.
2. Select the *Chapter* option in the option menu connected to the Organizer radio button.
3. Change the document name `Untitled` to **`Requirements Documents`**.
4. Press the *OK* button or <Return>. A chapter named `Requirements Documents` will appear as the uppermost chapter object.
5. Now repeat the steps above and add the three remaining chapters and name them **`System Analysis Documents`**, **`System Design Documents`** and **`Object Design Documents`**, respectively.

If the chapters show up in another order than the one you want in the Organizer window, you may move a selected chapter by using the arrow quick buttons in the tool bar.

6. If needed, move the chapters in the Organizer to get a structure corresponding to the one in Figure 712.

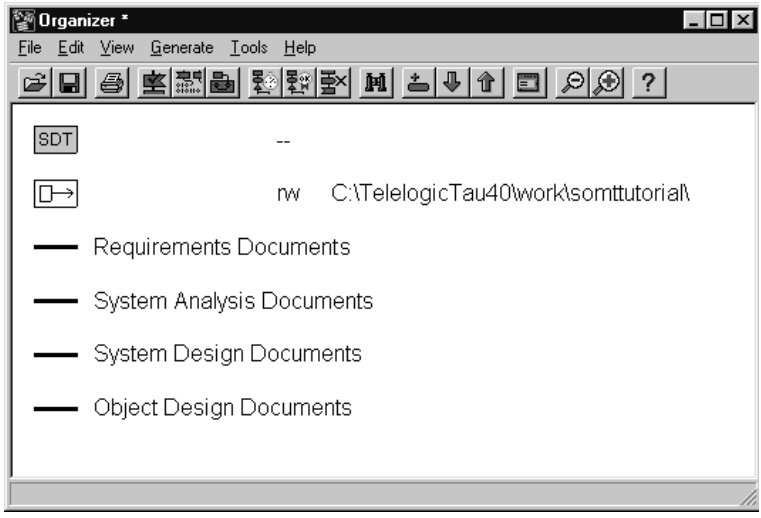

Figure 712: The Chapter structure

Adding the Organizer Modules

The next step to take when preparing the document structure is to add the Organizer *modules*. A module in the Organizer forms a naming scope around the documents it contains. It may contain any kind of documents.

As each activity in SOMT consists of a number of models, it seems natural to let a model correspond to a module in the corresponding chapter. You should now add the modules to the chapters in the Organizer structure.

1. Select the chapter named `Requirements Documents`.
2. Select the *Add New* command in the *Edit* menu.
3. In the Add New dialog, make sure that the Organizer radio button is set. Select the *Module* option in the Organizer option menu.
4. Change the name `Untitled` to **`RequirementsUseCaseModel`**

> **Note:**
>
> You are not allowed to have any space characters in the name of a module.

5. Press the *OK* button. A module named `RequirementsUseCaseModel` appears in the `Requirements Documents` chapter.

6. Now add the other modules to their respective chapter in the Organizer view. Let each model in a SOMT activity have its own module. The document structure in the Organizer should look like Figure 713 when you are finished.

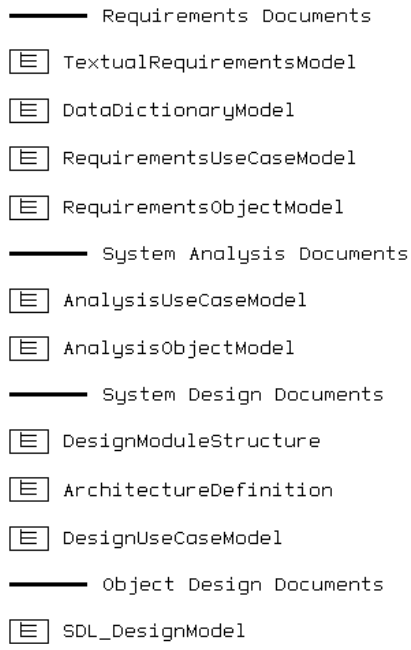

Figure 713: The complete document structure

This structure will form the framework to organize the forthcoming documents around.

7. *Save* the Organizer structure and name the file **`accesscontrol.sdt`**.

Now you have finished the preparations and you can start to develop the Access Control system using the SOMT method.

Identifying the Requirements

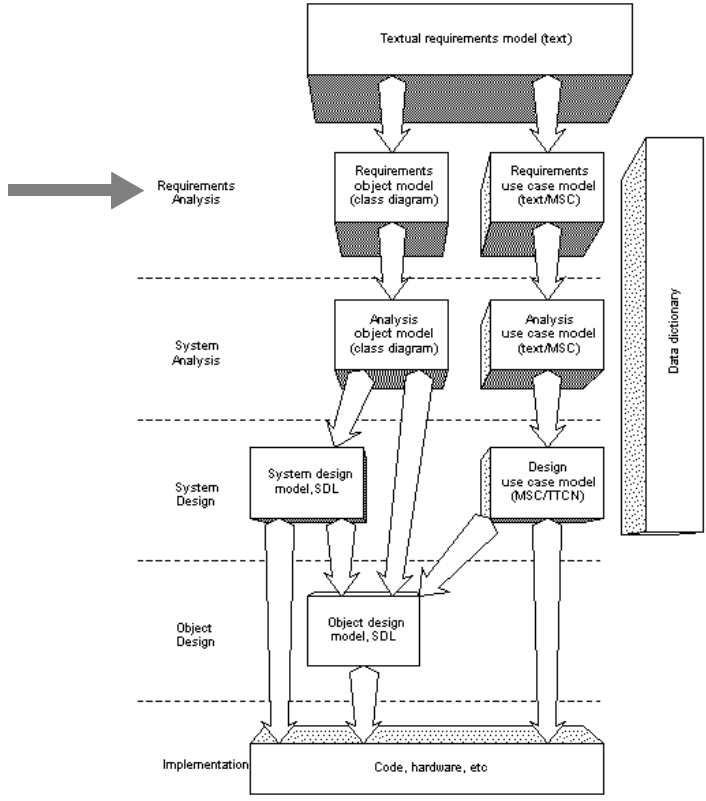

Figure 714: Overview of the SOMT process

What You Will Learn

- To bring in external (requirements) documents into the Organizer
- To identify important concepts
- To use a data dictionary
- To identify actors and use cases and to compile the information gained into textual documents

- To create a textual use case
- To create an MSC use case out of a textual use case
- To make a requirements object model
- To connect important concepts in the different documents with *implinks*
- To perform consistency checks

Introduction to the Exercise

In this exercise you will perform the tasks associated with the requirements analysis activity. The purpose of the requirements analysis is to:

- Gain understanding of the problem domain - the Access Control system and the environment in which it is going to exist.
- Find and understand all requirements imposed on the Access Control system.

Producing a complete requirements analysis would take too much time in this tutorial. Therefore, you will only perform parts of every required step of the process.

The result will not be a complete requirements structure, but you will have acquired knowledge of how to use the SOMT method in the process of identifying requirements.

Preparing the Exercise

You can use your own document structure from the previous exercise (just move your `accesscontrol.sdt` file to the `ReqA` directory), or use a provided solution.

1. Open the system file `somttutorial/ReqA/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\reqa\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/ReqA/` **(on UNIX)**, or `somttutorial\reqa\` **(in Windows)**, in the same way as you did in the preparation to this tutorial (see "Preparations" on page 3837).

Studying the Textual Requirements

Including External Textual Requirements

A textual document with requirements is the input to the Access Control system development project and it will form the base from which the Access Control system is developed. You will later on create implementation links (so called *implinks*) between the textual requirements document and other models. This is done to make it possible to follow a requirement through a number of models all the way down to code.

The textual requirements document of the Access Control system is contained in a text file. This file should now be included in the Organizer work area.

1. Select the module named `TextualRequirementsModel` in the `Requirements Documents` chapter.
2. Select the *Add Existing* command in the *Edit* menu.
3. In the Add Existing dialog, change the filter to **`*.txt`** and press the *Filter* button. Select the file `TextualRequirements.txt` and press *OK* to add it.
4. The `TextualRequirements` document is now added to the module `TextualRequirementsModel` in the Organizer and the *Text Editor* showing the document is opened. The document looks like Example 607.

Example 607: The textual requirements ––––––––––––––––––––––––

The task is to design the software to support a computerized Access Control system. The purpose of the system is to control the accesses to an office.

An entrance leading to an office can have four different security levels:

1. Always unlocked
2. Requires a card to unlock
3. Requires a card as well as a code to unlock
4. Always locked

The security levels of an office entrance can be altered during the day.

Each employee working in the office has a card with a personal code consisting of four digits. To open a door with security level three, the

employee enters her card into a card reader and types her personal code on a keypad. The time between consecutive keystrokes when typing the code is not allowed to exceed three seconds. To enter through a door with security level two, the employee just enters her card into a card reader.

Each entrance leading into the office consists of a door with an electric lock as well as a card reader, a keypad and a display on the outside, and an exit button on the inside. The employee needs a card and a code to *enter* the office. To exit, the employee just presses the exit button and the door is unlocked for ten seconds.

All entrances communicate directly with a central controller which makes sure that a validation of the correctness of cards and codes is performed. The controller has access to a database consisting of all card numbers and their corresponding personal codes. If the card is valid and, in case of security level three, the corresponding code correct, the door is unlocked for ten seconds and the employee may enter. In case of an invalid or unregistered card, access to the office is not allowed. In case of an incorrect code, the employee is informed of this and must try again by entering the card into the card reader and retyping the personal code.

The Access Control system must read its data, consisting of card numbers with their corresponding personal code, from a database. The database is managed by using a separate management system that is not developed within the project. The system operator, who is running the management system, is authorized to register new employees, cards and codes, to change a code if the employee wishes so, to delete employees from the database and to change the security level of an entrance. The system operator is also responsible for initializing the Access Control system. All the actions mentioned above are done using the management system.

The system must be able to recover from computer and connection failures. If a connection between an entrance and the central controller is lost, the door is locked from the outside not permitting anyone to enter (i.e. security level four is set). It is, however, possible to open the door from the inside by means of the exit button.

The system must be extensible to include new functions and be easily maintained.

Creating Textual Endpoints

Now you should study the textual requirements document and mark all concepts (nouns) that you find essential for the problem domain as *link endpoints*. These marks will be very useful in later stages of the project. In this tutorial most of the endpoints in the textual requirements document have already been created. You task is to add the two missing ones:

1. In the second sentence of the textual requirements document locate the word “office” and mark it with the mouse.

   When you mark an endpoint see to it that you only mark the word itself and not any additional characters, like a space or a dot after the word.

2. In the *Link* submenu in the *Tools* menu, choose *Create Endpoint*.

3. The text will be underlined indicating that the text fragment now is a link endpoint.

4. Now, locate the word “entrance” in the third sentence and create an endpoint out of it by repeating the procedure above.

5. If you go through the rest of the document you can see that the rest of the important concepts already have been marked as endpoints.

6. *Save* the document.

7. Open the *Link Manager*. This is done by choosing *Link Manager* in the *Link* submenu in the *Tools* menu. You can do this either in the Organizer window or in the Text Editor window; the result will be the same.

   The Link Manager window will pop up showing all the endpoints of the textual requirements document. The endpoint background color is used to show the endpoint status. As the endpoints are newly created, and the link file has not been saved yet, the background of the endpoints is painted gray.

8. *Save* the link file from the *File* menu, giving it the name `Links.sli`.

9. Close the Link Manager window.

Creating the Data Dictionary

A data dictionary is a textual document which should define all important concepts found during the whole development process. It forms a common vocabulary for the members of the project. It is a good idea to:

- Provide each item included in the data dictionary with a name and a brief explanation.
- Categorize the concepts in nouns, verb phrases and relation phrases.
- Sort the concepts alphabetically.
- Have a section in the data dictionary for each activity. This might be a good idea because a certain concept often has different meanings in different activities. For example, a concept can be described by a class in one activity and in the next activity it might be described by a block with a corresponding process.

All the important objects, relations and verbs that you find in the textual requirements should be included in the data dictionary. This has already been done in an existing `DataDictionary` file, so you do not have to do anything. Just add the existing file:

1. Add the existing `DataDictionary.txt` file to the `DataDictionaryModel` module in the Organizer. The Text Editor will show the `DataDictionary`.
2. Read through the document to get yourself acquainted with the problem domain vocabulary.

All nouns, relation phrases and verb phrases in the data dictionary are marked as link endpoints. This has been done to make it possible to do *entity matches* between any model and the data dictionary. An entity match checks that all entities in one model have matching entities in another model. That is, we can check that all entities in a model really are described in the data dictionary. This will be performed in "Entity Match" on page 3865.

The example below shows a part of the requirements analysis data dictionary.

Example 608: A data dictionary

Nouns/Objects

Access control system - A system to control the access rights to an office so that no unauthorized persons can enter without permission.

Card - Each employee working in the office gets a card and a corresponding personal code. By means of this card and code, the employee can get access to the office.

Cardnumber - The number that uniquely defines a card.

...

Relation Phrases

Card with code - Each employee in the office has a card with a personal code.

Connection between central controller and entrance - There is a connection between every entrance and the central controller.

...

Verb Phrases

Change code - An operation done by the system operator to change the code of a card.

Change Security Level - An operation done by the system operator to alter the security level of an entrance.

Connection is lost - The connection between an entrance and the central controller can sometimes fail. In case of broken connection nobody can enter the office. It is, however, possible to leave the office.

...

Creating the Use Case Model

The purpose of a *use case model* is to capture the requirements and present them from the users point of view, thus, making it easier for the intended users to validate the correctness of the requirements analysis.

The use case model consists of:

- A list of actors
- A list of use cases
- A number of MSCs (message sequence charts) and/or textual use cases

The use case model is also a useful source of information when developing the *requirements object model*, see "Creating the Requirements Object Model" on page 3861.

A use case is a sequence of actions showing a possible usage of a system. Use cases developed during the requirements analysis activity should mainly concern the interaction between the system and the users of the system. No message exchanges within the system should be shown.

Users of a system may be people, other systems or objects outside the system border which interact with the system.

An *actor* is a user taking part in a use case. An actor is not supposed to be an individual user, but rather represents one of the different roles a user can play when interacting with the system.

There are different ways to describe a use case:

- A textual description of the use case
- A description of the use case using an MSC
- A combination of both a textual description and an MSC

Describing use cases using textual descriptions will make it easier to model exceptions and alternative paths of action sequences. Describing use cases using MSCs will make the use cases more formal and easier to verify. Also, as MSCs will be used in the coming activities, it might be a good idea to start using them already in the requirements analysis. The tutorial will use both textual descriptions and MSCs in the requirements analysis activity, and only MSCs in the later activities.

Creating a List of Actors

Now it is time to create a list of actors. The list of actors should list the actors by name, together with their respective responsibility.

1. In the Organizer, select the `RequirementsUseCaseModel` module and choose *Add New*. In the Add New dialog, set the *Text* radio button and choose *Plain* in the corresponding option menu. Name the new document **`ActorsList`** and set the toggle button *Show in Editor*. This will give you a new text document in the Organizer window and an empty Text Editor window will pop up.
2. Try to find the actors of the Access Control system by studying the textual requirements in Example 607 on page 3845. For information on how to find actors, see "Finding Actors" on page 3851 below.
3. List the actors by name in the newly created textual document together with a brief description of the actor's role when interacting with the system.
4. Mark the actors in the `ActorsList` as endpoints in the same way as you did in "Creating Textual Endpoints" on page 3847.
5. *Save* the document giving it the name **`ActorsList.txt`**.

Finding Actors

You will find actors by studying the textual requirements. Useful questions to ask are:

- Which users will need services from the system to perform their tasks?
- Which users are needed by the system to perform its tasks?
- Are there any external systems that use or are being used by our system?

In practise, the activity of defining actors should be performed iteratively. Try to find as many of the actors as possible now. If you do not believe you found them all, start creating some MSC use cases (as having done some MSCs often makes it easier to determine the actors). Then go back and complete the list of actors.

In our case with the Access Control system we find that an employee, who daily interacts with the system, is an obvious candidate for the list

of actors. Also, considering the third question above, it is obvious that the management system is being used by our system and therefore should be added to the list. The third actor, the door, may not be so easy to find at a first glance, but when you have created the MSCs it will become more evident that the door is an actor as well. The door's interaction with the system consists of notifying the system every time it is opened or closed.

The example below shows a part of the list of actors.

Example 609: Part of a list of Actors

Employee - Someone who needs to enter and exit the office. To enter the office, an employee must have a registered card and (depending on the current security level) a corresponding personal code. To exit, the employee must press an exit button to unlock the door.

ManagementSystem - The management system starts and maintains the Access Control system. All changes to the database are handled by the management system. The management system is run by a system operator.

...

Creating a List of Use Cases

When you have defined the set of actors it is time to describe the way they interact with the system, which is done in use cases. The first step is to create a list of all use cases. The list of use cases should list the use cases by name together with a short description.

1. *Add* a *new* plain text document in the `RequirementsUseCaseModel` module. Name it **`UseCaseList`** and set the toggle button *Show in Editor*. Press *OK*.

2. Try to find the normal use cases and list them in the newly created textual document. For information on how to find use cases, see "Finding Use Cases" on page 3853.

3. To each use case, add a general one-sentence description of its functionality.

4. For each normal use case, examine which exceptions that can occur and state these exceptions as well in the list.
5. Mark the use case names in the `UseCaseList` as endpoints.
6. Save the document giving it the name **`UseCaseList.txt`**.

Finding Use Cases

It is often quite easy to identify use cases by looking at the purpose of the system. To verify that you have identified most of the important use cases you should:

- look at the list of actors, and, for each actor,
- identify the tasks that the actor should be able to perform and the tasks which the system needs the actor to perform. Each such task is a candidate for a new use case. It is often very useful to check the textual requirements document for verb phrases (or you could look directly in the DataDictionary to see which verb phrases that have been stated as important); these are possible candidates for use cases.

Start with the employee actor and try to determine which actions he or she needs to perform. There are different ways to enter an office, either using a card or using both a card and a code. Both ways are obvious candidates for the use case list. Also, the employee must be able to exit the office, this will be yet another use case.

The Management system must inform the Access Control system when there has been a change in security level. This will be our fourth use case.

As for the door actor, the task of notifying the system when a door is opened and closed can be included in the enter/exit office use cases. You should always try to make the use cases as complete as possible, that is, make **one** complete use case instead of several minor ones.

When you have found the normal use cases, refine them by examining the exceptions that are possible for each use case. Look in the textual requirements document and try to find the exceptions that can occur.

In the case where an employee enters the office, the first thing that can go wrong is that there is a connection failure between the entrance and the central controller. Other possible things that can fail are that the card

is invalid, the code is wrong, the time between consequent keystrokes when typing the code is too long, and, finally, the door is never opened even though it was unlocked. All these exceptional cases can be found by studying the textual requirements thoroughly.

The example below shows a part of the use case list.

Example 610: Part of a Use Case List ––––––––––––––––––––––––––

Normal Cases:

Enter_Office_With_Card - Describes the interaction between an employee and the Access Control system when the employee wants to enter the office through a door with security level two.

Enter_Office_With_Card_And_Code - Describes the interaction between an employee and the Access Control system when the employee wants to enter the office through a door with security level three.

Exit_Office - Describes the interaction between an employee and the Access Control system when the employee wants to exit the office.

...

Exceptional Cases:

Exc_No_Connection

Exc_Invalid_Card

...

––

Creating a Textual Use Case

Now that we have a list of the actors to the system as well as a list of use cases, we can start to create a more detailed description of the use cases. A textual use case consists essentially of natural text structured into a number of text fields, see "Describing a Textual Use Case" on page 3855. In this exercise we will only create one textual use case, as creating them all takes too much time. The use case we will focus on

throughout the rest of the tutorial is the one where an employee enters an office with both a card and a code.

1. Add a new textual document in the `RequirementsUseCaseModel` module and name it **`Enter_Office_With_Card_And_Code`**.
2. Try to create the textual use case consisting of the fields described in "Describing a Textual Use Case" on page 3855.
3. Create endpoints of the textual use case name (for consistency use exactly the same name as you used in the list of use cases) and of the actors involved in the use case.
4. Save the document giving it the name **`Enter_Office_With_Card_And_Code.txt`**.

Describing a Textual Use Case

A textual use case should consists of the following fields:

- **Name**: The name of the use case.
- **Actors**: A list of the actors involved in the use case.
- **Preconditions**: A list of properties that must be true for this use case to take place.
- **Postconditions**: A list of properties that are true when the use case is finished.
- **Description**: A textual description of the normal sequence of events that describe the interaction between the actors and the system.
- **Exceptions**: A list of exceptional interactions that complement the normal flow of events described in the `Description` field. If an exception leads to different postcondition properties compared to the normal sequence this should be noted.

The description field should thus describe what happens when everything is going as expected. No exceptions should be considered here. They are not described until the exceptions field.

The example below shows a textual description of the use case "Enter office with card and code."

Example 611: A Textual Use Case

Use case name: Enter_Office_With_Card_And_Code

Actor: Employee, door

Preconditions: System is initialized, security level three is set, and the door is closed and locked. The display displays “Enter card”.

Postconditions: The door is closed and locked again.

Description: An employee enters a card into the card reader. The display displays “Enter code”. The employee enters a code consisting of four digits using the keypad. The door is unlocked and “Please enter” is displayed. The employee opens the door, enters the office and closes the door again. The door is locked and “Enter card” is displayed.

Exceptions:

- If the employee enters an invalid or unregistered card, “Invalid card” is displayed for three seconds and then “Enter card” is displayed.

- If the time between consequent keystrokes when typing the code exceeds three seconds, everything is interrupted and “Enter card” is displayed.

- If the employee types the wrong code, “Wrong code” is displayed for three seconds and then “Enter card” is displayed.

- If the employee does not open the door within ten seconds after it has been unlocked the door is locked again and “Enter card” is displayed.

- If there is no connection between the entrance and the central controller and a card is entered, then the text “Connection failure” is displayed for three seconds and then “Enter card” is displayed again.

Creating an MSC Use Case

The second notation for use cases used in SOMT is MSCs. Creating MSCs for all the use cases and their exceptions takes too much time in this tutorial. Therefore you will concentrate on the use case corresponding to the textual description you just created,

`Enter_Office_With_Card_And_Code`, and one of its exceptions, when an employee enters an invalid card.

1. Select the module `RequirementsUseCaseModel` and choose *Add New*. In the Add New dialog, set the *MSC* radio button. Name the document **`Enter_Office_With_Card_And_Code`** and set the *Show in Editor* toggle button.

2. In the MSC Editor, try to create the MSC. Look at the textual description of the use case and describe it by means of the notations defined for MSCs. Also, make references to exceptions at the points where these can occur. Figure 715 shows an example of the complete MSC.

   – Each actor should be represented by a separate instance. The Access Control system itself should also be represented by a separate instance.

   – Actions, displayed messages, etc. should be drawn as MSC messages between the instances.

   – An exception is drawn by adding an *MSC reference* symbol, located last in the MSC Editor's symbol menu. An MSC reference symbol is a reference to another MSC, described in a separate MSC diagram. The symbol is added to one of the instance axes. By convention, MSC exceptions are named "exc" followed by the name of the exception. To connect the symbol to all three axes, select *Connect* from the *Edit* menu. Press the *Global* button to connect the reference symbol to all axes.

3. Save the MSC diagram giving it the name **`Enter_Office_With_Card_And_Code.msc`**.

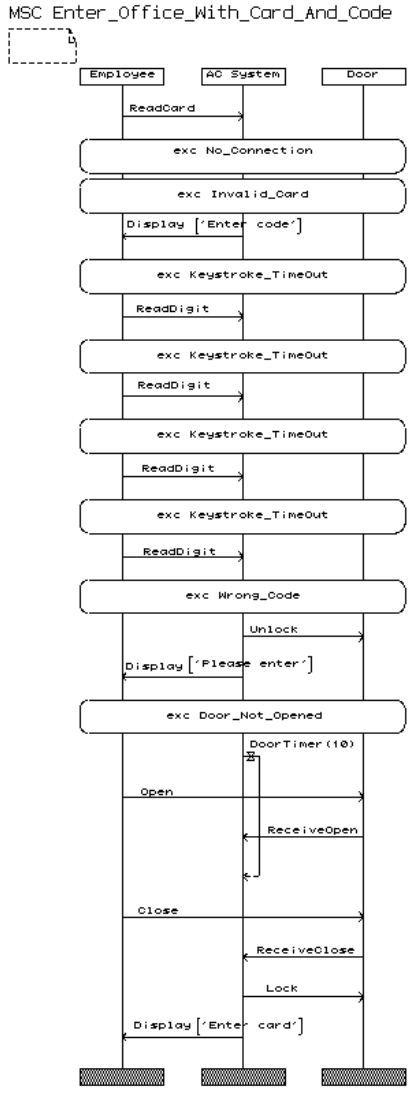

Figure 715: An MSC example

4. Create a new module in the `Requirements Documents` chapter in the Organizer. Name the module **`MSC_Exceptions_ReqA`**.
5. Add a new MSC document to the newly created module in the Organizer and name the document **`Exc_Invalid_Card`**.
6. In the MSC Editor, try to create the exception, i.e. describe what happens when an employee has entered an invalid card.
7. Save the diagram giving it the name **`Exc_Invalid_Card.msc`**.

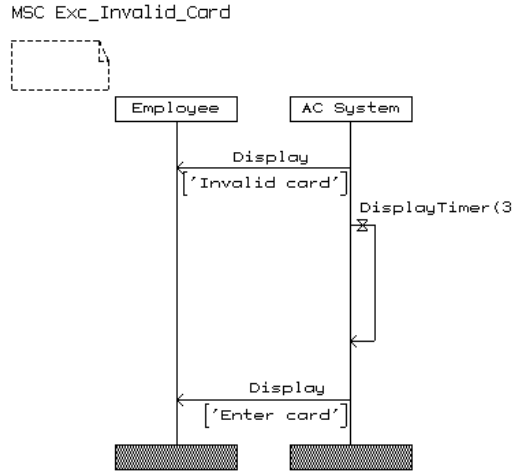

Figure 716: An MSC exception example

8. In the Organizer view, select the MSC `Exc_Invalid_Card` and then choose *Associate* in the *Edit* menu. The Associate dialog appears.
9. Choose to associate the `Exc_Invalid_Card` MSC with the `Enter_Office_With_Card_And_Code` MSC as it is an exception to this use case.

The `Requirements Documents` chapter should now look like in Figure 717.

In reality you repeat the steps above for all the use cases found and associate each one of them with its exceptions. In this tutorial, however,

we will not create the entire use case model as that would take too much time.

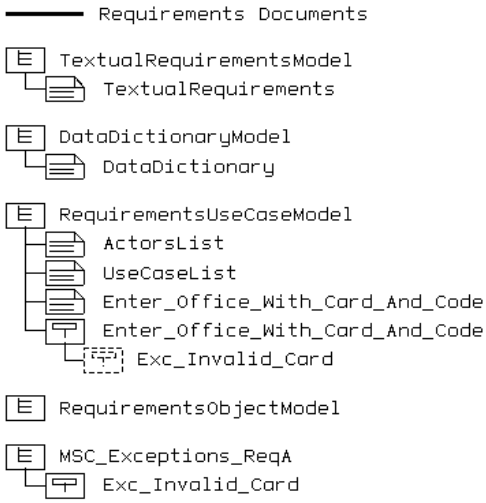

Figure 717: The Requirements Documents chapter

Now, when we have our use cases and a data dictionary we will continue the activity with producing a requirements object model. In practise, you should work with all the models in parallel. The activities in SOMT are **not** supposed to be performed in a sequential order, rather, producing the models is a highly iterative process.

Creating the Requirements Object Model

The requirements object model is intended to capture the objects, the relations between these objects and other concepts of the real world that are of importance for the application we intend to build. There are different types of concepts that can be described in this model. The two major diagram types show the logical structure of the data and information and the context of the system.

Relations between objects in the model will be expressed through associations, aggregations and inheritance.

Creating a Requirements Object Model

Now you should create the requirements object model.

1. In the Organizer, select the `RequirementsObjectModel` module and choose *Add New*. In the Add New dialog, set the *UML* radio button and make sure the *Object Model* option in the UML option menu is set. Name the new document **`LogicalStructure`** and set the toggle button *Show in Editor*. This will pop up an empty OM Editor window.
2. Try to find the objects, see "Identifying the Objects" on page 3863.
3. Enter the classes found into the object model diagram in the OM Editor and give them a suitable name. As you can see, every class is automatically marked as an endpoint.
4. Relate the classes by means of associations, aggregations and inheritance, see "Identifying the Relations" on page 3864.
5. Consider if multiplicity is needed on any of the associations and if so, add it. (Double-click a line to bring up the Line Details dialog.)
6. To increase the readability of the model, name the associations or attach role names to the classes. The diagram should look something like in Figure 718 when you are finished.

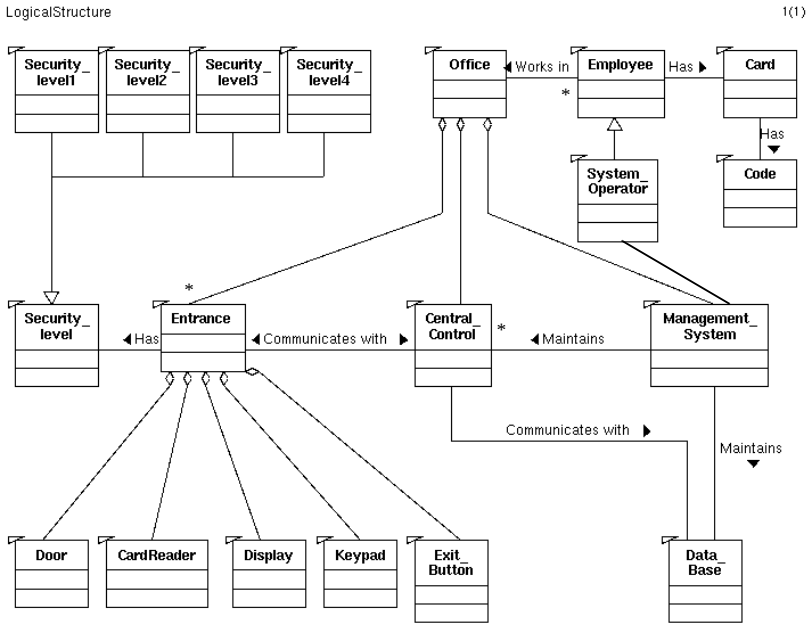

Figure 718: The logical structure

7. Save the diagram giving it the name **`logicalstructure.som`**.

8. Add yet another object model to the `RequirementsObjectModel` module in the Organizer. Name it **`ContextDiagram`**.

9. In the diagram, show the system and the external actors interacting with it. Use collapsed class symbols (select *Collapse* from the *Edit* menu). The classes are automatically marked as endpoints.

10. *Clear* the endpoint on the `Access_Control_System` class as we will not need this. (Select *Clear Endpoint* from the *Link* submenu in the *Tools* menu.)

11. *Save* the diagram giving it the name **`contextdiagram.som`**.

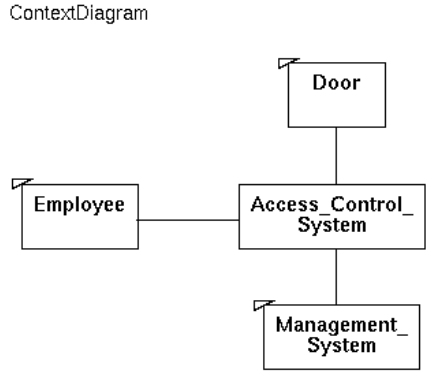

Figure 719: The Context diagram

Identifying the Objects

The main input sources to the requirements object model are the textual requirements, the use case model and the data dictionary. Other sources of information are domain experts, textbooks etc.

A classical way to find the objects is to study the textual requirements and note all nouns (or look directly in the nouns section in the data dictionary). If a particular noun appears in many places, the concept is probably important for the problem domain and should be modeled in the requirements object model.

The use cases are also helpful for finding the objects. They define the actors that interact with the system and these are obvious object candidates. Other likely object candidates are the entities that are transported in to or out of the system. The use cases are helpful in identifying these concepts as well.

The requirements object model should at least describe all concepts that are visible on the outside of the system. This includes all physical entities that a user can see as well as the knowledge a user must have to use the system. It is, however, not only concepts **outside** the system that should be modeled in the requirements object model. Concepts **inside** the system that are so obvious that we know of them already at this stage should be dealt with as well. In our Access Control system, for instance, it is quite easy to see that the system itself is built up of a central control and a number of entrances, each having its own local control. Therefore,

in the requirements object model, we do not model the heart of the system as one class, but as two communicating classes.

Identifying the Relations

The information sources when identifying relations between objects are the same as when identifying the objects. Look for relation phrases in the textual requirements (or use the data dictionary as an information source). You may also take a look at each object and ask the questions:

1. What services does the object provide?
2. Does the object need services from other objects to complete its services?

If the object needs services from other objects, identify these objects and model the relations in the object model.

There are three different types of relations, described below.

The Association Relation

The association relation describes how different classes relate to each other by means of information exchange.

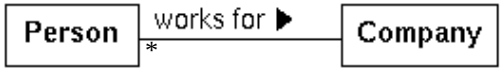

Figure 720: An Association relation

The Aggregation Relation

The aggregation relation is a special case of the association relation and it describes a “consists of” relation. For example: a document **consists of** paragraphs.

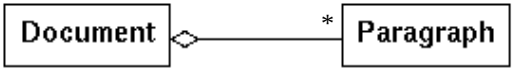

Figure 721: An Aggregation relation

The Inheritance Relation

The inheritance relation describes an “is a” relation. For example: a car **is a** vehicle.

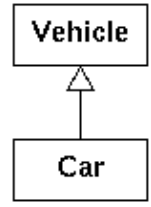

Figure 722: An Inheritance relation

Entity Match

Now it is time to do some consistency checks between the created models. When you do an entity match you check that all entities in one model have matching entities in another model.

1. *Open* the Link Manager. The window popping up shows all endpoints from the models you have created during the requirements analysis activity.

2. To be able to perform an entity match you must be in entity mode (i.e. **not** in endpoint mode). Press the *Show endpoints or entities* button in the Tool bar to change to entity mode. (The view in the Link Manager window will look just the same, since one entity corresponds to exactly one endpoint in all our models.)

The first thing we will check is that all important concepts in the textual requirements model are described in the data dictionary.

3. Choose *Consistency Check* in the *Tools* menu. The Consistency Check dialog appears and you are asked to choose between a link check and an entity match. Set the *entity match* radio button and press *Continue*.

4. A new dialog appears and you are asked to select the documents representing the **from** group. Select the `TextualRequirementsModel` module and press *Continue*. (As you can see, the text document in the module is also selected when you select the module.)

5. Yet another dialog appears asking you to select the documents representing the **to** group. Select the `DataDictionaryModel` module and press *Check*.

6. The Link Manager window will show the result of the entity match in a new *consistency view*. Entities from the **from** group are shown as normal endpoints and entities from the **to** group are shown as dashed endpoints. The links shown are temporary links created by the Link Manager to indicate matching entities, see Figure 723.

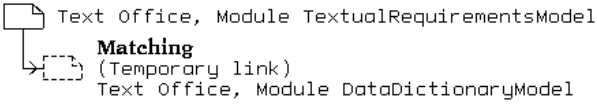

Figure 723: Matching entities

7. As you can see, if you scroll through the Link Manager window, all the concepts from the textual requirements have a matching entity in the data dictionary. (An endpoint from the **to** group without a matching link from it would have indicated that no corresponding entity could be found in the **to** group.)

There are a few more consistency checks which you can perform at this point:

- Check that all entities in the requirements object model are described in the data dictionary.
  - Let the `RequirementsObjectModel` module form the **from** group and the `DataDictionary` module the **to** group.
  - As you can see from the result all concepts but the four different security levels have been described in the data dictionary.
- Check that all important concepts in the textual use cases are described in the data dictionary and in the use case list. Important concepts in a textual use case are the actors and the use case name. The actors should be described in the data dictionary and in the actors list. The use case name should be described in the use case list.
  - In this case the **from** group will be the textual use case, `Enter_Office_With_Card_And_Code`, and the **to** group will be the `DataDictionary`, the `ActorsList` and the

`UseCaseList`. (You can select any number of individual documents in the list, not only modules.)

– The result shows that the use case name was found in the `UseCaseList` and that the actors were described both in the data dictionary and in the list of actors.

- Check that all actors in the use cases are modeled in the context diagram and vice versa.

  – First, the `Actorslist` will form the **from** group and the `Context Diagram` will form the **to** group, then we will do another entity match with the groups vice versa.

Creating Implinks

Now that we know that all our models are consistent, it is time to add the *implinks*. Implinks are used to enable traceability between the models.

We will start with creating implinks from the concepts in the textual requirements to the requirements object model, in particular the logical structure diagram.

1. Open the Link Manager by selecting it in the submenu *Link* in the *Tools* menu if it is not already open.

2. Make sure the window shows endpoint view, not entity view or consistency view. If necessary, press the *Show endpoint or entities* quick button.

   In the Link Manager window you see all the endpoints from the different models in the requirements analysis activity. If you scroll to the very end of the window you can see how many endpoints and links you have in your system. There should be no links at this point.

3. To check that all endpoints are present in the Link Manager window, choose *Check Endpoints* in the *Tools* menu.

4. The Check Endpoints window will pop up showing if any previously unknown endpoints were found. If so, select these and press *Add*. Then press *Continue*.

5. Another version of the Check Endpoints window pops up showing if any invalid endpoints were found. In such case you can choose to

delete these by pressing *Delete*. Press *OK* when you are done and want to close the dialog.

6. To be able to create the implinks between the textual requirements and the classes in the logical structure you only need to see the endpoints from these diagrams in the Link Manager window. You do not have to see all the other endpoints. Therefore, choose *Filter* in the *View* menu.

   The Filter dialog pops up and you can choose to set filter settings for links, endpoint or documents, by selecting from the option menus.

7. Choose to set the filter for *documents* and select that the only documents to be *shown* should be the textual requirements and the logical structure documents.

8. Press *Apply* and then *Done*.

9. In the Link Manager window, highlight the endpoint `Text Office` by first selecting the endpoint and then clicking the *Highlight quick* button in the tool bar. The endpoint is highlighted with a frame around it.

10. Create an implink to the `Class Office` by first selecting the class and then pressing the *Create Link* quick button in the tool bar. The Create Link dialog will open.

11. Name the link **`Implementation Link`** and press the *Create* button. A link from the text “office” to the class Office is created.

The rest of the links between the textual requirements and the object model diagram are created in a similar way. You can do this if you want, or go to the next exercise, where this has been done, and check out the result.

Links from the `UseCaseList` to the different MSCs and their exceptions should also be created. You cannot do this here however, as you have not created all the use cases.

What we aim at here is to create links from the textual requirements, through the object models of the different activities, to the SDL design. Simultaneously we want to create links from the list of use cases, through the use case models in the different activities, to the SDL design. The result of this will be that we can trace a design decision backwards to requirements through either object models or use case models.

Summary

After having completed an entire requirements analysis, the `Requirements Document` chapter the Organizer view should look like in Figure 724.

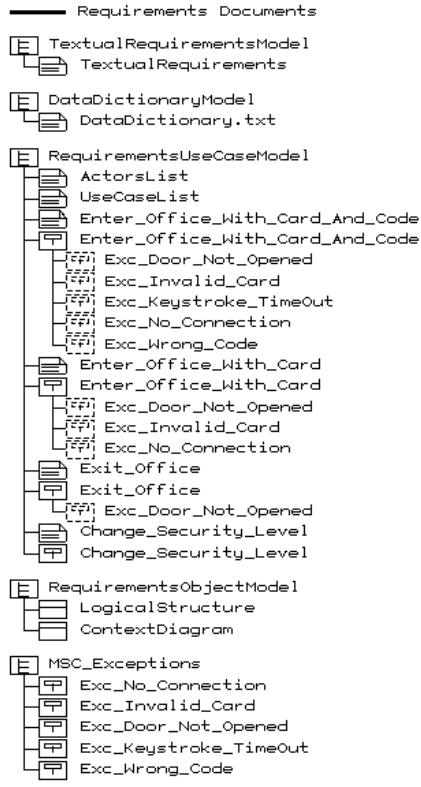

Figure 724: The entire requirements analysis document structure

Performing the System Analysis

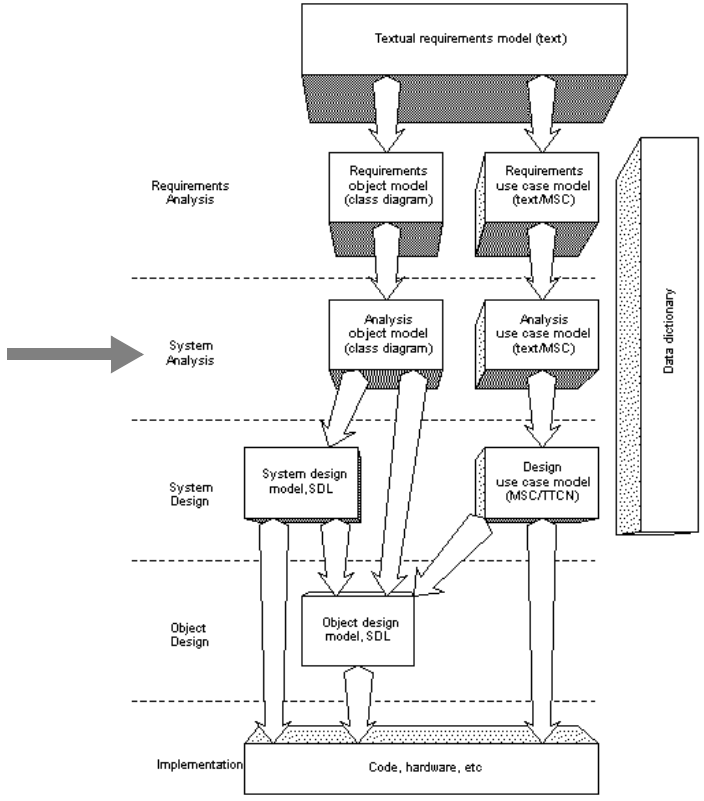

Figure 725: Overview of the SOMT process

What You Will Learn

- To identify and present the logical architecture of a system which includes refining the object model from the previous phase
- To refine use cases from the previous phase
- To use the *Paste As* mechanism

Introduction to the Exercise

In this exercise you will perform the system analysis activity. The purpose of the exercise is to outline a logical model of the Access Control system. This model will fulfil the requirements that were identified in the requirements analysis. In other words, the purpose of this activity is to identify the objects that are needed in the Access Control system and the services these objects should provide.

Producing a complete system analysis structure takes too much time. Thus, you will only perform parts of every step necessary to produce the complete structure.

The input to the system analysis activity is a complete requirements structure with the two main models:

- requirements object model
- requirements use case model

The output from the system analysis activity are the two models:

- analysis object model
- analysis use case model

These two models should be created in parallel through a number of iterations.

Preparing the Exercise

1. Open the system file `somttutorial/SysA/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\sysa\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/SysA/` **(on UNIX)**, or `somttutorial\sysa\` **(in Windows)**.

What you see in the Organizer window is a complete requirements analysis structure with all implinks made.

Creating the Analysis Object Model

The analysis object model is a refinement of the requirements object model. However, when transferring from the requirements analysis to the system analysis you change the focus.

During the requirements analysis the focus is on understanding the problem and the problem domain. In the system analysis the focus is to model a solution and to understand the logical structure of the system that will be the solution to the stated problem. This change of focus should be reflected by the analysis object model.

Little emphasis should be put on implementation aspects during the system analysis activity. Questions regarding the implementation of the solution will very likely hide our actual problem.

A glance at the headlines may give you the impression that the activity of creating the analysis object model is a sequential activity, but it is not. You will probably not first add all the necessary classes, then the relations, and finally specify the attributes and operations. This is, however, the way the text is structured here to make it readable and to highlight the important tasks of the activity.

Creating a Logical Architecture

Now it is time to create the logical architecture diagram.

1. Select the `AnalysisObjectModel` module in the `System Analysis Documents` chapter in the Organizer.
2. Add a new object model diagram and name it **`LogicalArchitecture`**.
3. Open the `LogicalStructure` diagram from the previous activity as well as the new `LogicalArchitecture` diagram in the OM Editor.

Adding Classes to the Logical Architecture

Now you should start adding classes to the logical architecture diagram. For information on how to find the classes, see "Finding Classes" on page 3874.

Several of the classes in the requirements object model can be transferred as-is to the analysis object model. The provided *Paste As* mechanism lets you transfer objects from one model to another while auto-

matically creating implinks between the objects in the two separate models. This mechanism should be used here, see below.

4. *Select* the class `CentralControl` in the `LogicalStructure` diagram and choose *Copy* in the *Edit* menu.
5. Go to the `LogicalArchitecture` diagram by selecting it in the *Diagrams* menu.
6. Choose *Paste As* in the *Edit* menu. The Paste As dialog is opened.
7. Set the option menu to *Class* and see to it that the *Create link* toggle button is **set**. Press the *Paste As* button.
8. Select where in the `LogicalArchitecture` diagram you want to place the `CentralControl` object.

Now you have created a class named `CentralControl` in your `LogicalArchitecture` diagram. The class is connected with an implink to the `CentralControl` class in the `LogicalStructure` diagram. The link is indicated by the filled triangle on the class symbol.

9. Repeat the procedure above with the class `Entrance` but name the new class **`EntranceUnit`** as this is a more descriptive name.
10. The classes `Cardreader`, `Keypad`, `Display` and `ExitButton` are obvious interfaces to our system and, also, parts of an `EntranceUnit`. Repeat the procedure above with these classes. As it is often useful to name objects according to their function, give the new classes names of the form **`xxxInterface`**.

One class in the requirements object model may result in several classes in the analysis object model. One reason may be that the class provides so much functionality that splitting the class into several smaller may be convenient. Another reason may be that a class identified in the requirements object model needs services from other, not yet introduced classes. Consequently these classes should be introduced at this point of the development process.

In practice, the actions in the two cases above are the same. The newly introduced classes should be linked to the original class in the requirements object model. In our example this is the case with the class Door. In the requirements object model we have one single class representing the door. Further analysis, however, shows that the door object includes both a door lock as well as a door sensor. This aggregation structure

should be shown in the analysis object model. See section "Finding Relations" on page 3876.

11. *Copy* the class `Door` in the `LogicalStructure` diagram.
12. *Paste* it three times *as* a class in the `LogicalArchitecture` diagram and see to it that the implinks are created at the same time. *Name* the new classes `DoorUnit`, `DoorLockInterface` and `DoorSensorInterface` respectively.

Now look at the classes you have so far in the `LogicalArchitecture` diagram. Think about the tasks of the different objects. As you can see there is no class that can handle the logic, that is, an object that is responsible for what happens at an entrance. Therefore, you should add such an object and name it **`EntranceCtrl`**, see below. The `EntranceCtrl` will be a part of the `EntranceUnit`.

13. *Copy* the class `Entrance` from the `LogicalStructure` diagram.
14. *Paste* it *as* a class in the `LogicalArchitecture` and rename the class, giving it the name `EntranceCtrl`.
15. Also, *copy* and *paste* the `SecurityLevel` class and its subclasses.

Classes in the requirements object model that only exist outside the system border or classes that do not provide any necessary services should not be transferred at all to the analysis object model.

Finding Classes

Useful sources where you can find objects that may be included in the analysis object model are:

- the requirements object model
- interfaces that the system will have to the environment
- use cases

When you intend to transfer objects from the requirements object model to the analysis object model, consider the following to validate each requirements object:

- Decide if the system needs information about the object to fulfill its task.
- If the answer is "yes" then add the class to the analysis object model.

Another useful way to find the objects is to examine which interfaces the system needs. Often the application area itself makes it obvious what interfaces must exist. To find the interface objects, go through the list of actors and, for each actor, decide which interfaces that are needed to the system.

The use cases from the system analysis activity are also a useful source for finding the objects. The problem is that we have not created these use cases yet. As stated before, the work done in each activity is often done iteratively. This is a typical situation where an iteration is needed as some objects in the analysis object model may not be found until we have created and inspected the analysis use case model.

When you have created the MSCs you should examine them to check which interface objects that are involved and which internal objects that are modified. Also, check if there is a control object that might handle the logic of the use case or if there is a need to introduce such an object in the analysis object model.

Adding Relations to the Logical Architecture

When you have identified all classes it is time to add the relations. For information about how to add relations, see "Finding Relations" on page 3876.

In our Access Control system example, most of the relations from the requirements object model can be preserved.

1. Connect the `CentralControl` to the `EntranceUnit` with an association. Add multiplicity to the association. The `CentralControl` may be connected to several `EntranceUnits` but the Access Control system has only one `CentralControl`.
2. Connect the `EntranceUnit` to the `DoorUnit` with an aggregation. One `EntranceUnit` consists of only one `DoorUnit`.
3. The `DoorUnit` consists of a `DoorLockInterface` and a `DoorSensorInterface`. Add aggregations from the `DoorUnit` to the `DoorLockInterface` and to the `DoorSensorInterface`.
4. Connect the rest of the classes you have added with necessary associations, aggregations and generalizations. Also consider if multiplicity is needed or not.

Finding Relations

Useful sources that may assist the process of finding relations are:

- the requirements object model (preserving and modifying existing relations)
- analysis use case model
- textual requirements

The process of finding new relations and verifying old ones are closely related to the process of creating the analysis use case model. It is mainly a question about which other objects the object needs to know about to be able to provide its services. Also generalizations and aggregational dependencies have to be considered, see "Identifying the Relations" on page 3864.

Adding Attributes to the Logical Architecture

Now we have come to the point where it is time to add the attributes. For information on how to find the attributes, see "Identifying Attributes" on page 3876.

There are not many classes in our `LogicalArchitecture` diagram that need any attributes. In fact there is only one, the class `Display`. This class must be able to display different text messages depending on the situation at hand. Therefore:

1. Define `Text` to be an attribute of the class `DisplayInterface`.

Identifying Attributes

Attributes can be found in:

- the requirements model (keeping existing attributes)
- the textual use cases
- the textual requirements

Attributes describe a property of an object and often correspond to nouns. For example, possible attributes of an object "Person" may be eye color, weight, shoe size, and so on. Attributes that may describe a vehicle are owner, color, current speed, current gear, and direction.

Adding Operations to the Logical Architecture

The last thing to add to the logical structure object model are the operations. For information on how to find the operations, see "Identifying Operations" on page 3878.

Add the operations `Open`, `Close`, `Lock` and `Unlock` to the assembly class `DoorUnit`. This implies that you also must add the operations `Open` and `Close` to the class `DoorSensorInterface` and `Unlock` and `Lock` to the class `DoorLockInterface`.

1. Select the class `DoorUnit`.
2. Click on the operations section in the class.
3. Add the operations `Open`, `Close`, `Lock` and `Unlock`.
4. Repeat the procedure for the classes `DoorLockInterface` and `DoorSensorInterface` adding their respective operations.
5. Continue to add the missing operations of the other classes in the diagram. When you are finished, your diagram should look something like in Figure 726.
6. Save the diagram.

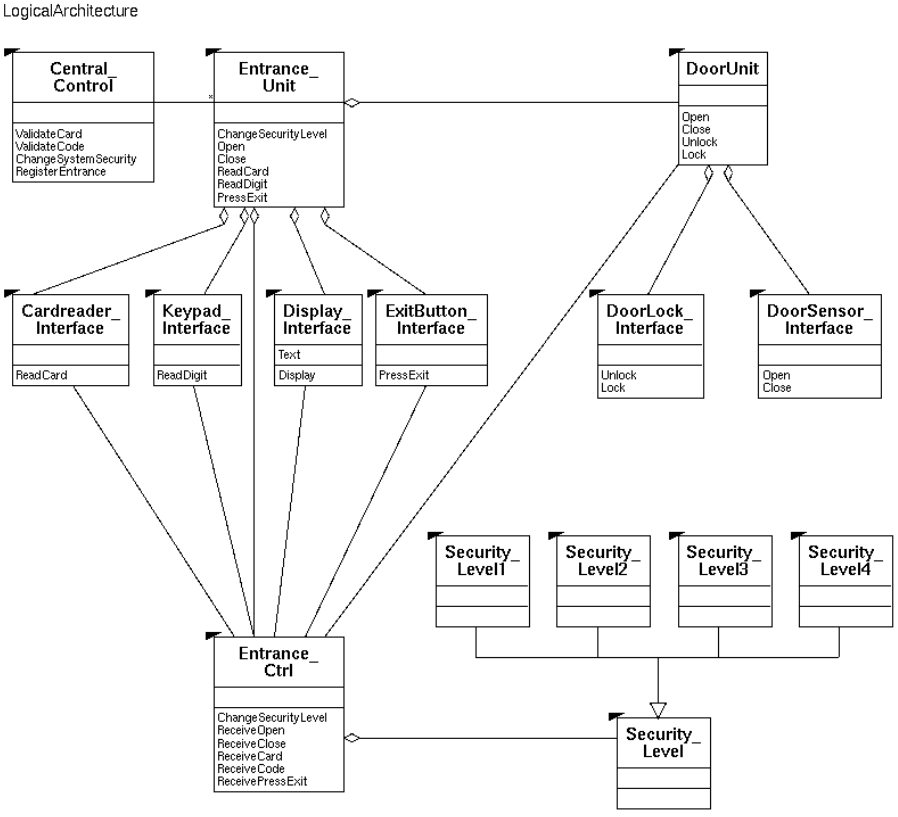

Figure 726: The logical architecture

Identifying Operations

Sources that may support the identification of operations are:

- the requirements object model
- the analysis use case model (the messages in the MSC diagrams)
- the data dictionary (the description of the objects)

By studying the responsibilities of each object is it possible to identify a set of operations that will provide the services assigned to the object.

Let one operation perform only one task. However, the class should not contain too many public operations. A large public interface of a class may indicate that the object is assigned too many responsibilities. Instead the object should probably be split and the responsibilities of the object should be distributed between several objects.

The easiest way to find the operations is probably to look at the MSC messages in the system analysis use cases. The messages can often be considered as operations in the analysis object model. A message received by an instance in an MSC corresponds to an operation on the corresponding class. Note that the operations on the classes representing subsystems define the interface of this subsystem and should often also exist as operations on some of the objects within the subsystem.

Creating an Information Diagram

Now it is time to create an information diagram. This diagram describes the concepts outside the system that the system must know of to fulfill its task. In our Access Control system example, `Card` and `Code` are two such concepts.

1. *Add* a *new* object model diagram to the Organizer in the module `AnalysisObjectModel`. Name the diagram **`InformationDiagram`**.
2. *Open* the `LogicalStructure` diagram from the previous activity and the new `InformationDiagram` in the OM Editor.
3. *Select* and *copy* the classes `Card` and `Code` in the `LogicalStructure` diagram and *paste* them *as* classes in the `InformationDiagram`, while automatically creating the implinks.
4. Consider if any or both of the classes should have any attributes.
5. Associate the classes with each other.
6. Save the diagram.

Your diagram should look like in Figure 727 when you are finished.

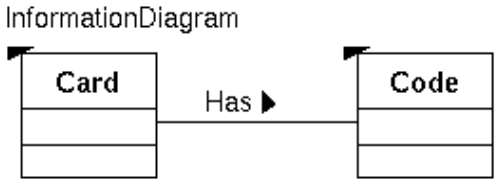

Figure 727: The Information diagram

Creating the Analysis Use Case Model

The design of the two models in the system analysis activity, the object model and the use case model, is usually going on in parallel. The models view the Access Control system from two different perspectives, a dynamic perspective and a static perspective.

The analysis use case model shows the dynamic aspects and consists of a set of MSC diagrams. These diagrams may be categorized into two types:

- Refined requirements use cases
- Behavior pattern use cases

Refined Requirements Use Case

A refined requirement use case is what it reads like. Each valid use case from the requirements use case model is transferred to the analysis use case model and redesigned and refined to the analysis object model.

The purpose of the refined use cases is to validate whether the analysis object model really implements the requirements. At the same time the analysis use case model is an important source of information for identifying operations on the classes in the object model.

In the analysis use case model, the use cases are documented preferably using MSCs. MSC diagrams are more formal and correspond better to the object model than textual use cases.

Each instance in the MSC diagram corresponds to an object or subsystem in the analysis object model. The level of abstraction you choose is a trade-off between detail and clarity.

An object which encapsulates interfaces and just transfers the calls to other objects without adding much functionality may be omitted, but objects providing crucial functionality should be part of the MSC.

Creating a Refined Requirements Use Case

Now you should create an analysis use case of the `Enter_Office_With_Card_And_Code` use case from the requirements analysis activity.

1. Create a new MSC diagram in the `AnalysisUseCaseModel` module and name it **`Enter_Office_With_Card_And_Code_SysA`**.
2. Open the requirements use case `Enter_Office_With_Card_And_Code`. The system analysis use case is based on the requirements use case and it is therefore useful to use the latter as a reference (using copy and paste).
3. Create the MSC on subsystem level, that is, replace the original system instance with instances of `EntranceUnit` and `CentralControl`.
4. Create endpoints of the three instances in the MSC diagram. (Select *Create Endpoint* from the *Link* submenu in the *Tools* menu.)
5. For each message in the requirements use case decide which instances that exchange that particular message in the analysis use case. Also consider which additional messages you need to add. All message exchanges **inside** the system that we did not consider in the requirements use cases have to be added at this point. Also, make references to exceptions at the points where these can occur.
6. Replace the four `ReadDigit` signals with an MSC reference symbol referring to the MSC ReadCode. ReadCode is a behavior pattern which you will create later, see "Behavior Pattern Use Cases" on page 3883.
7. Save the diagram.
8. In the Organizer, create an endpoint out of the newly created MSC diagram. (This is done in the same way as in the editors.)
9. Open the Link Manager and connect this endpoint to the `Enter_Office_With_Card_And_Code` MSC from the requirements use case model. Name the link **`Implementation Link`**.

The created MSC should look like in Figure 728.

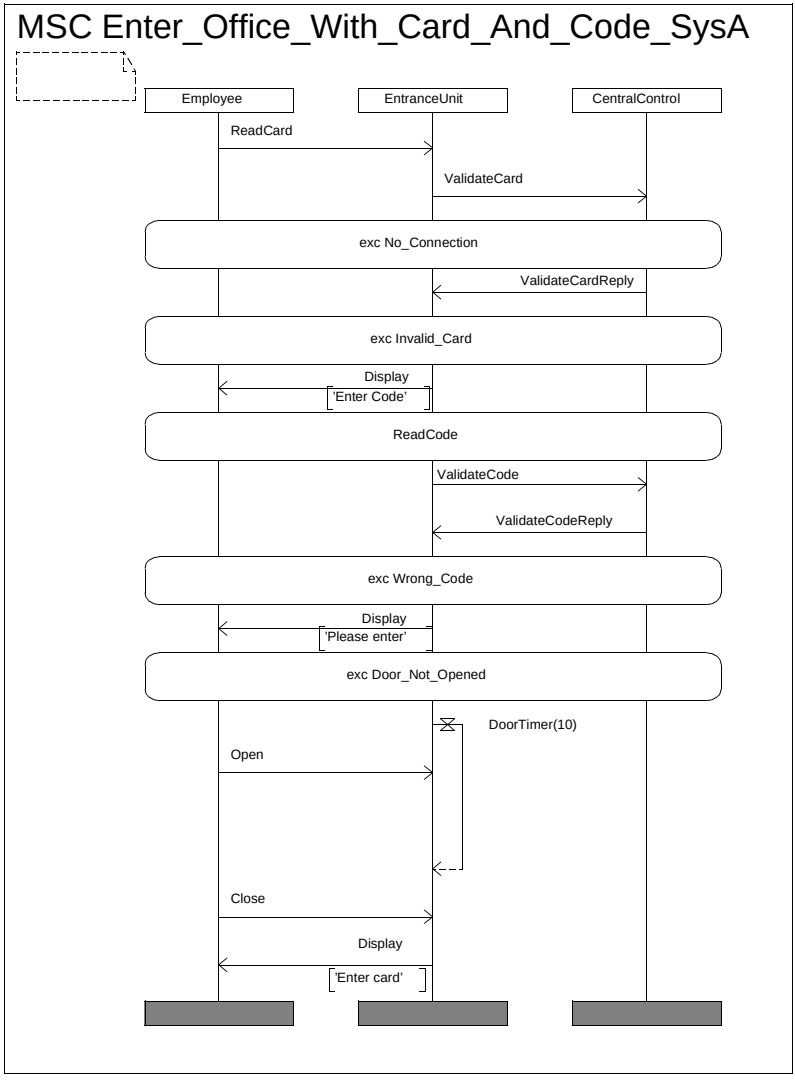

Figure 728: A system analysis MSC use case

The rest of the requirements use cases and their respective exceptions are refined in a similar way. This will not be done in this tutorial because that would take too much time.

Behavior Pattern Use Cases

A behavior pattern is a detailed use case that may be used to examine special communication patterns in detail. A behavior pattern is part of an ordinary use case and most often several use cases share a behavior pattern. A use case may include none or several behavior patterns.

Behavior patterns let refined requirements use cases be presented in a higher abstraction level, making these less complex and easier to understand. By focusing use cases on special parts of the system it is easier to understand and maintain the requirements on the involved objects.

Creating a Behavior Pattern

The refined use case that you just created was created on subsystem level. With the help of behavior patterns we can describe what really happens at a certain point in the use case, i.e. which objects that interact and the messages that they exchange. In our case, where we will create a behavior pattern for the task of reading a code, we have to replace the MSC instance `EntranceUnit` with the MSC instances `KeypadInterface` and `EntranceCtrl`.

1. Create a new Organizer module in the `System Analysis Documents` chapter. Name it **`BehaviorPatterns`**.
2. In the MSC Editor, create the behavior pattern `ReadCode`, see Figure 729.
3. Save the diagram using the name **`Behavior_Pattern_ReadCode`**.
4. In the Organizer, associate the behavior pattern MSC with the use case MSC which it really is a part of. That is, associate it with `Enter_Office_With_Card_And_Code_SysA`.

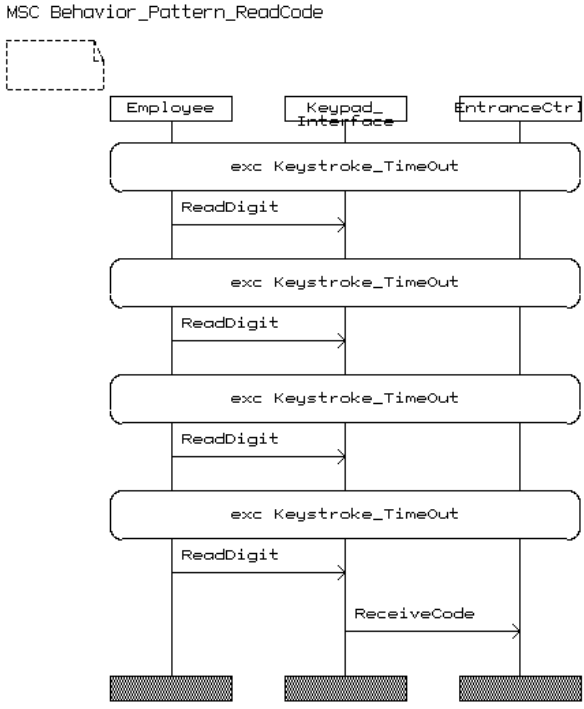

Figure 729: A behavior pattern example

The `System Analysis Documents` chapter should now look like in Figure 730.

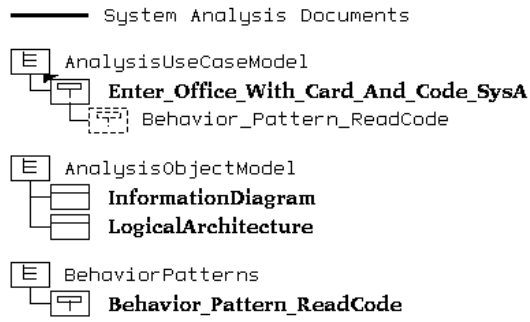

Figure 730: The System Analysis Documents chapter

Requirements Traceability

One important aspect in this activity is the relation between the models created here and the models created in the requirements analysis activity. We want to be able to check:

- That all requirements have been implemented
- Which system analysis object that implements a certain requirement
- Which requirement that is implemented by a certain object in the analysis object model

The means to check the issues above is through consistency checks. There are two types of consistency checks:

- Entity matches
- Link checks

We will start with a link check and then we will perform an entity match.

Link Check

The first thing to check is that all entities described in the logical structure in the requirements object model are either represented in the analysis object model or not really needed by the application.

1. *Open* the Link Manager. A link check can be performed in both entity and endpoint view, so it does not matter which view you have in the window.
2. Choose *Consistency Check* in the *Tools* menu to perform a link check.
3. The Consistency Check dialog pops up asking you to select the documents representing the **from** group. Select the `LogicalStructure` object model and press *Continue*.
4. Yet another Consistency Check dialog appears, now asking you to select the documents representing the **to** group. Select the `AnalysisObjectModel` module (this will also highlight the documents in the module) and press *Check.*
5. The Link Manager will show the result of the link check. You can see that all entities from the requirements, except the system operator, employee, database, management system and office, are represented in the analysis object model. These are concepts on the outside of the system and, thus, not really needed by the application.

To follow links from one model to another we use the *Traverse* command. To see how this works follow the steps below:

6. Go to the `LogicalArchitecture` diagram in the OM Editor and select the class `CentralControl`.
7. In the *Link* submenu in the *Tools* menu, choose *Traverse*.
8. The OM Editor will open the `LogicalStructure` diagram and the `CentralControl` class will be selected. Go yet another step backwards by choosing *Traverse* in this diagram.
9. The Traverse Link dialog pops up asking you to select a link to traverse. The class is the one we just came from so you should choose the text fragment and press *Traverse Link*.
10. The Text Editor opens and the endpoint `centralcontrol` is selected.

In the same way we traversed from system analysis to requirements here, you can also traverse from the requirements to system analysis. Try this!

Entity Match

Now it is time for another consistency check. This time you should check that the instances in the MSC diagram correspond to classes in the object model or to actors that interact with the system.

1. See to it that you have entity view in the Link Manager window. If not, press the *Show endpoint or entities* button in the tool bar.
2. Choose *Consistency Check* in the *Tools* menu. Set the *entity match* radio button in the Consistency Check dialog and press *Continue*.
3. As a document representing the from group, select the MSC `Enter_Office_With_Card_And_Code_SysA`.
4. As documents representing the to group, select the `AnalysisObjectModel` module and the `ActorsList`.
5. The result shows that all MSC instances really are described in the analysis object model or in the list of actors.

Summary

After having completed an entire system analysis, the `System Analysis Document` chapter would look like in Figure 731.

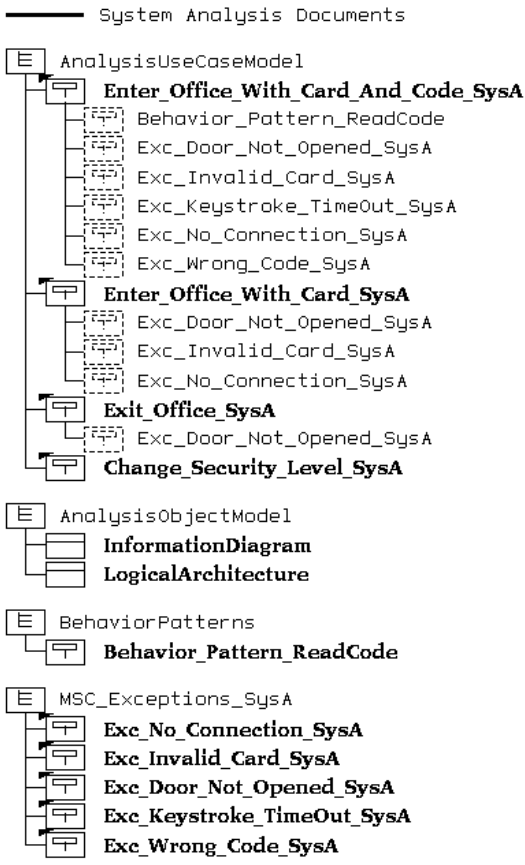

Figure 731: The entire system analysis document structure

Performing the System Design

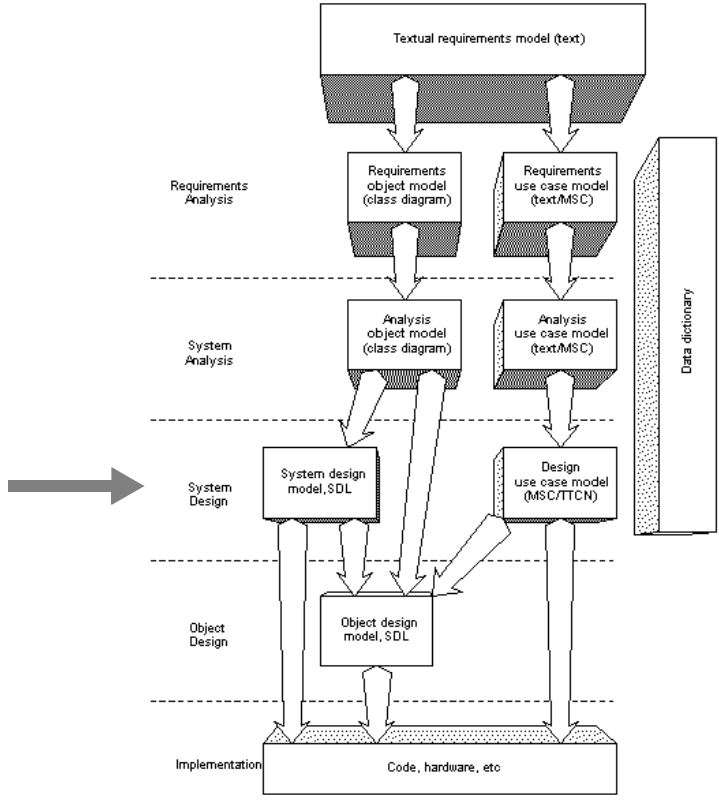

Figure 732: Overview of the SOMT process

What You Will Learn

- To create a design module structure
- To define the static interfaces in packages
- To make an architecture definition of the system
- To make formalized testable use cases
- To use the *Paste As* mechanism when transferring from the system analysis activity to the system design activity

Introduction to the Exercise

This is an exercise on system design. In this activity we no longer make use of object models; from now on SDL will be used. You will learn how to map concepts from the analysis object model from the previous activity into an SDL model. Mapping object-oriented concepts to SDL concepts forces you to make several design decisions. Support for these design decisions is provided through the *Paste As* mechanism. This exercise will teach you to make use of this support.

Useful sources for information in the system design activity are the analysis object model and the analysis use case model. The first provides information about the static structure and is useful when structuring the system into units. The latter provides information about the dynamic structure and is useful for the definition of the interfaces between the units.

Major tasks to perform in the system design are:

- Define the design module structure.
- Define the static interfaces.
- Create an SDL system structure as a starting point for the formalization of the architecture.
- Define the dynamic aspects of the interfaces by a continued use of use cases.

Producing a complete system design structure in the tutorial takes too much time. Thus, you will only perform parts of every step necessary to produce a system design structure.

Preparing the Exercise

1. Open the system file `somttutorial/SysD/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\sysd\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/SysD/` **(on UNIX)**, or `somttutorial\sysd\` **(in Windows)**.

What you now see in the Organizer window is a complete requirements analysis and system analysis structure.

Design Module Structure

An important part of the system design is to divide the system into units considering division of work, distribution of functionality and physical distribution.

The purpose of the design module structure is to show the actual source code modules the application will be built from. The most important aspect of the module structure is that it forms the basis for dividing the work load on different development teams.

In our Access Control system example we have two different subsystems, EntranceUnit and CentralControl. It might be the case that these subsystems are implemented by two different teams. Therefore, it seems natural to introduce two different modules, each module being described by the concept of a package. The contents of the packages will not be defined until the architecture definition, the task here is only to identify the modules needed.

The notation that will be used in this tutorial to describe the design module structure is the object model instance diagram where the instances represent the different modules.

Creating a Design Module Structure

1. Add a new object model diagram to the Organizer in the module `DesignModuleStructure` and name the document **`DesignModuleStructure`**.

2. In the OM Editor, create a first object instance symbol representing the whole SDL system. Name it using the name **`SDL_System_Access_Control`**.

3. Decide which modules, i.e. packages, the SDL system is to make use of. In this example it might be suitable to introduce two different modules, one for each subsystem. Therefore, create two more object instances and name them **`CentralControlPackage`** and **`EntranceUnitPackage`** respectively.

4. Draw associations from the SDL system module to the two new modules. The associations describe that the SDL system uses these two packages.

5. Create yet another module which will consist of all common types and signals. This package is used by the other two packages and thus

also by the SDL system itself. Name the module **`UtilityTypesPackage`** and draw the associations. Your diagram should now look like in Figure 733.

6. *Create endpoints* out of the three packages. (Endpoints are not created automatically for object instance symbols.)
7. Save the diagram giving it the name **`designmodulestructure.som`**.

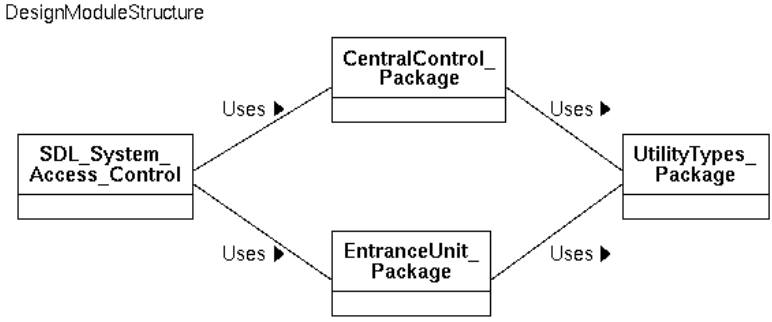

Figure 733: The Design Module Structure

Creating the Architecture Definition

When using SDL to design a system, the architecture is defined by block diagrams which define how the system is decomposed into blocks. The block diagrams are more or less a formalization of the analysis object model.

The first thing we will do when defining the architecture is to define the contents of the packages introduced in the design module structure, see "Defining the Packages" on page 3893. The `UtilityTypesPackage` will contain data type declarations that are common to the subsystems of the system. Signals/remote procedures that make up the interface between the subsystems will also be defined in the `UtilityTypesPackage`. The other two packages, `CentralControlPackage` and `EntranceUnitPackage`, will contain block types and the signals/remote procedures that make up the interface to the particular block. If you have many signals it is often useful to structure these into signal lists.

The basic mechanism used in SOMT when going from analysis to design is the *Paste As* mechanism. This is used here when defining the signal interfaces, the blocks and block types, and, to some extent, the data type declarations.

The second task to do when creating the architecture definition is to define the system by means of SDL block diagrams, see "Creating the SDL System Diagram" on page 3898.

Defining the Packages

Mapping Object Models to SDL Block Types

You should now define the two block types, `CentralControl` and `EntranceUnit` in their respective package using the *Paste As* mechanism.

1. *Add* three *new* SDL packages to the `ArchitectureDefinition` module in the Organizer and name them according to the modules in the design module structure.
2. *Open* the `LogicalArchitecture` diagram in the `AnalysisObjectModel` module.
3. Select the class `CentralControl` and *copy* it.
4. Go to the SDL Editor and the `CentralControlPackage` diagram and choose *Paste As*. The Paste As dialog is opened.
5. *Paste* the class `CentralControl` *as* a *Block Type* (use the option menu) and create an implementation link from the copied object to the pasted object at the same time. The link is automatically created by default.
6. *Copy* and *paste* the class `EntranceUnit` *as* a block type in the `EntranceUnitPackage` in a similar way.

Mapping Object Models to SDL Interface Definitions

Now it is time to design the interfaces of the newly pasted blocks. Interface definitions in SDL are defined using signals and/or remote procedure calls. Consequently, this is what is produced when mapping a class to an SDL interface.

Note that the signals that constitute the interface between our subsystems should be described in the `UtilityTypesPackage`. The rest of the signals, i.e. signals that are **not** exchanged between the subsystems should be described now. Signals from the environment to the subsystem and signals inside the subsystem are such signals.

1. Once again, go to the `LogicalArchitecture` diagram in the `AnalysisObjectModel` and *copy* the class `EntranceUnit`.
2. Go to the SDL Editor and the `EntranceUnitPackage` diagram and *paste* the class *as* a *Text symbol with SDL interface*. See to it that an implementation link is created at the same time.

   If you look at the signal definition you see that the `ChangeSecurityLevel` signal is present. This is a signal exchanged between our two subsystems and, thus, should not be declared here.
3. Delete the `ChangeSecurityLevel` signal.
4. Add the signal parameters. The signals `ReadCard` and `ReadDigit` are the only ones that need parameters. See Figure 734.
5. Save the diagram, giving it the name **`entranceunitpackage.sun`**.

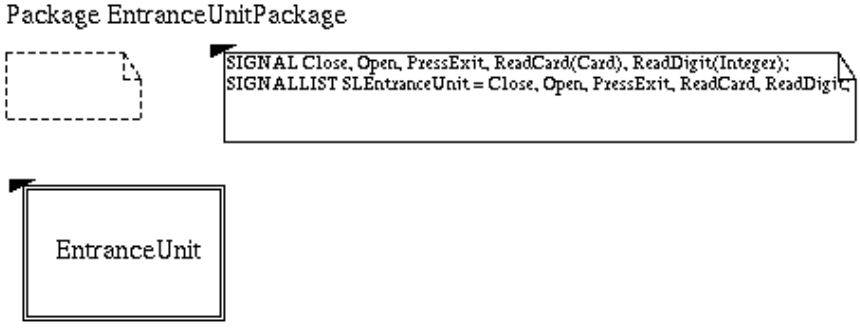

Figure 734: The EntranceUnitPackage

6. Go back to the `LogicalArchitecture` diagram and *copy* the class `CentralControl`.
7. *Paste* it *as* a text symbol with SDL interface in the `CentralControlPackage`.

8. If you look at the signal definition you realize that the `ValidateCard`, `ValidateCode` and `RegisterEntrance` signals are signals exchanged between our subsystems and, thus, should not be defined here. Therefore, delete these signals from the signal definition.
9. Add a signal parameter of the type integer to the `ChangeSystemSecurity` signal, see Figure 735.
10. Save the diagram, giving it the name **`centralcontrolpackage.sun`**.

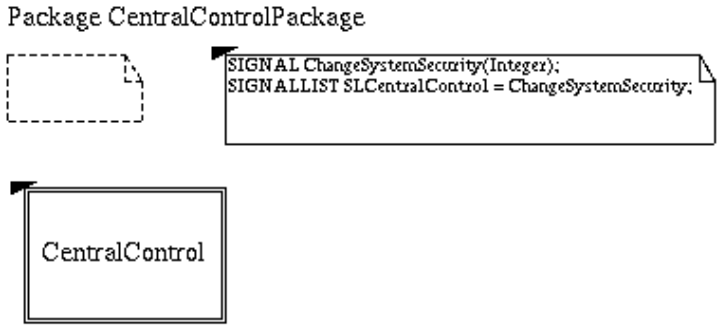

Figure 735: The CentralControlPackage

It is now time to define the last package, the `UtilityTypesPackage`. It will define the common data types needed in the subsystems as well as the interface between the subsystems. To define the package you should use the same procedure as above.

1. *Copy* the class `CentralControl` from the `LogicalArchitecture` diagram.
2. *Paste* it *as* a text symbol with SDL interface in the `UtilityTypesPackage` diagram.
3. Delete the signal `ChangeSystemSecurity` as this has already been defined in the `CentralControlPackage` (because it is a signal from the environment).
4. Add signal parameters to the three remaining signals.

5. Continue with copying the class `EntranceUnit` and *paste* it *as* a text symbol with SDL interface in the `UtilityTypesPackage`.

6. Remove all signals that have already been defined, i.e., all but the `ChangeSecurityLevel` signal.

Note that the Validate commands (`ValidateCard` and `ValidateCode`) have to be implemented with signals and not remote procedure calls. This is due to the fact that we must be able to keep track of how long it takes before we get the answer of a validation (to find out if there is a connection failure or not). As a consequence of this, the reply signals to the Validate commands have to be defined here.

7. Add the two signals `ValidateCardReply` and `ValidateCodeReply` to the latest pasted signal definition.

8. Add signal parameters to the three signals in the definition.

Mapping Object Models to SDL NewTypes

Now it is time to declare the common data types. In our Access Control system example the concepts of card and code are data types common to all parts of the system.

The data type `Card` is best declared as a SYNTYPE. You will have to do this declaration manually as there is no support in the *Paste As* mechanism for pasting something as a SYNTYPE. In declaring the data type `Code`, however, the *Paste As* mechanism can be used.

1. Declare the `Card` concept as a SYNTYPE in the `UtilityTypespackage`, see Figure 736.

2. *Open* the `InformationDiagram` in the `AnalysisObjectModel` and *copy* the class `Code`.

3. Go to the SDL Editor and the `UtilityTypesPackage` and choose *Paste As*. The Paste As dialog is opened.

4. *Paste* the class `Code` *as* a *NEWTYPE with graphical operator* or as *NEWTYPE with textual operator* (it does not matter which one you choose as the class Code has no operator) and see to it that an implementation link is created at the same time. This will only give you the structure, you have to fill in all relevant information yourself.

5. As it is a type we are declaring, change the name `Code` to **`CodeType`**. Define the `CodeType` concept to be an array consisting of four integers (you will have to delete the word STRUCT). The index type of the array should be integer and you will have to define this type too, see Figure 736.

6. Declare a SYNONYM **`NbrOfEntrances`** that will be used to alter the number of entrances we are to control in our system. For now, you can set the value of the variable to **`1`**, see Figure 736.

Before you save and close the diagram you should create an implementation link between the class `Card` from the InformationDiagram to the definition of `Card` in this diagram. You have earlier used the Link Manager to manually create links. We will now show how to use the *entity dictionary* for this.

The entity dictionary is available in all editors and lists all entities in the system, i.e. all modules, diagrams, and endpoints. The main usage of the entity dictionary is to reuse names of entities, but it can also be used to create links.

1. In the `UtilityTypesPackage` diagram, select the text symbol containing the `Card` definition.

2. From the *Window* menu, choose *Entity Dictionary*. The Entity Dictionary window is opened. You will recognize the structure and icons of chapters, modules and diagrams from the Organizer window. In addition, all endpoints are listed beneath the corresponding diagram.

3. Scroll the Entity Dictionary window until you find the class `Card` in the `InformationDiagram` in the `AnalysisObjectModel` module. Select the icon representing the class `Card`. (Make sure not to double-click an icon, since that will copy the name of the icon into the current diagram!)

4. In the Entity dictionary, press the *Create Link* quick button.

5. The Create Link dialog pops up. (This is the same dialog as when you created links in the Link Manager.) Set the *from* radio button, name the link **`Implementation Link`** and press *Create*.

6. *Close* the Entity dictionary by using the *Close* quick button.

7. *Save* the diagram, giving it the name **`utilitytypespackage.sun`**.

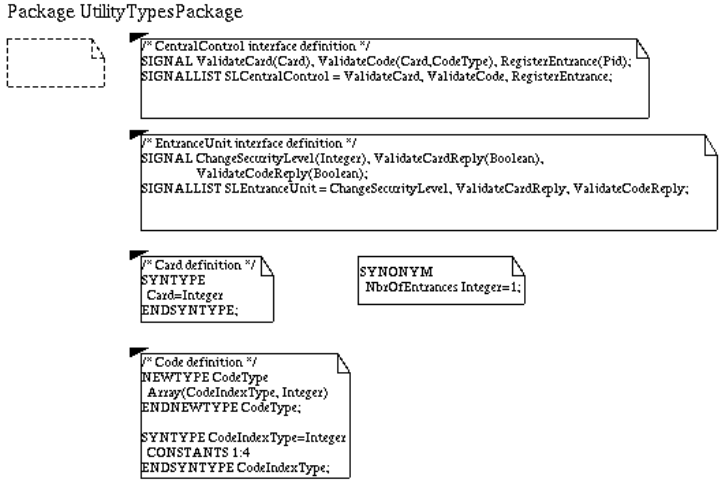

Figure 736: The UtilityTypespackage

8. In the Organizer, *create endpoints* out of the three packages.

Creating the SDL System Diagram

Now you should define the system structure, something which is to be done by means of SDL blocks in a system diagram.

1. *Add* an SDL system diagram to the `ArchitectureDefinition` module in the Organizer and name it **`AccessControl`**.

2. *Copy* the class `CentralControl` in the `LogicalArchitecture` diagram and *paste* it *as* a *Block* in the system diagram. See to it that an implementation link is created at the same time.

3. Also, *copy* and *paste* the class `EntranceUnit` as a *block.*

4. Change the name of the block instances, from `CentralControl` to **`theCentralControl`** and from `EntranceUnit` to **`theEntranceUnit`**. Also, define which block type the blocks are an instance of, see Figure 737.

5. There will often be more than one entrance in a building and therefore you should define the block `theEntranceUnit` as a *block instance set*. (Use the variable `NbrOfEntrances` defined in the `UtilityTypesPackage`.)

The two blocks `theCentralControl` and `theEntranceUnit` must be able to communicate with each other as well as with the environment. The next step is to create all necessary channels in the system diagram.

6. Draw the channels through which the blocks communicate and name them and the gates with suitable names. Also, draw the channels to/from the environment. State which signals that run on a certain channel. To identify the signals consult the `AnalysisObjectModel` made in the system analysis.

7. At this point we see the need to introduce a new signal, the `EnvDisplay` signal which goes from the `EntranceUnit` to the environment (i.e, to the display hardware). This signal contains information that is to be read by persons in the system environment. The parameter of this signal is `Charstring`. Define it in the `EntranceUnitPackage`.

8. Reference the packages the SDL system makes use of in a USE clause in the top of the diagram. Your system diagram should now look like in Figure 737.

9. Save the diagram, giving it the name **`accesscontrol.ssy`**.

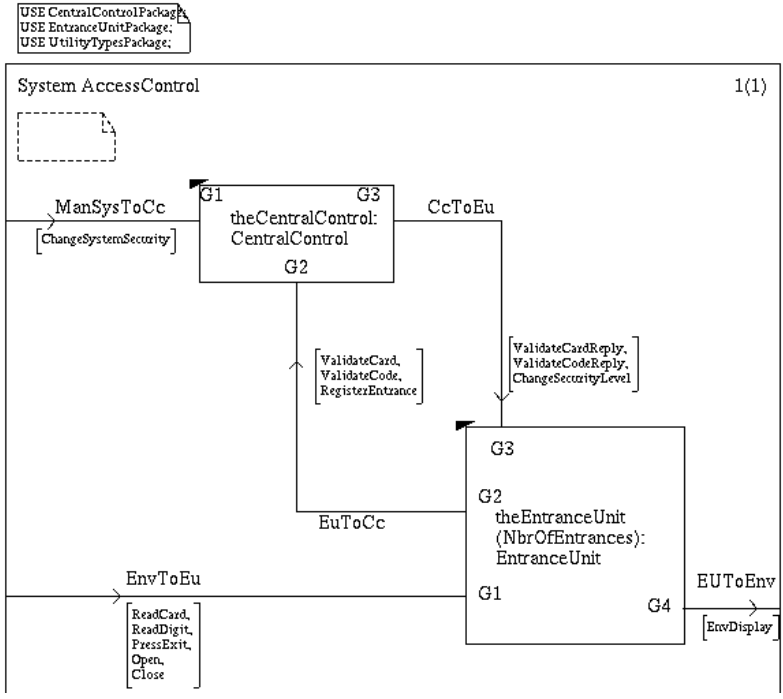

Figure 737: The Access Control system diagram

Refining the EntranceUnit – Mapping Aggregations

Considering the aggregation structure in the `LogicalArchitecture` diagram, the block `EntranceUnit` may be divided into several sub-blocks.

When mapping an aggregation structure from the analysis object model to an SDL diagram, the most common choice is to map the assembly class to a block type. Classes which are part of the assembly class will be mapped to process types, or, if the part class itself is an assembly class, to block types.

The block type corresponding to an assembly class will thus contain processes or blocks.

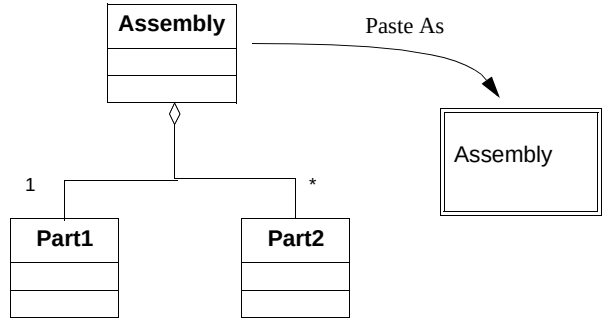

Figure 738: Mapping aggregation to a block type

The process types and block types should be placed in a suitable package to enable reuse. The instances of the types should then be placed in the appropriate system or block diagram to form the system.

Mapping Classes to Block Types and Signal Interfaces

1. Select the class `DoorUnit` in the `LogicalArchitecture` diagram. *Copy* it and *paste* it *as* a block type in the `EntranceUnitPackage`.
2. Call the *Paste As* command again. This time, choose to paste the class as a text symbol with SDL interface in the `EntranceUnitPackage`.
3. The resulting signal definition contains two signals that already have been defined in the package, namely `Open` and `Close`. Therefore, delete these signals in the newly pasted signal definition.

Mapping Classes to Blocks

Now it is time to place a block instance of the `DoorUnit` block type in the `EntranceUnit` block type.

1. Double-click on the `EntranceUnit` block type in the `EntranceUnitPackage` diagram. The Edit dialog pops up, press *OK*. The Add Page dialog pops up. Create a *block interaction page*.
2. For the third time, choose *Paste As* and paste the class `DoorUnit` as a block in the `EntranceUnit` block diagram. The block will be named *DoorUnit*.

3. Change the block name so it is marked as an instance of the block type according to the SDL syntax. Change the name to **`theDoorUnit:DoorUnit`**.

Introducing new Block Types

Since SDL does not allow processes and blocks at the same level you have to create two more block types to be placed in the `EntranceUnitPackage`. These block types will be used as containers to the remaining classes that will be mapped to processes and process types. Give the first block type the name **`EntranceInterface.`** We introduce this block as a generic term for all those classes that represent an interface to the system, i.e. the cardreader, the keypad, the display and the exitbutton. The second block type to be placed in the `EntranceUnitPackage` is **`EntranceCtrl`**.

1. *Copy* the class `DisplayInterface` in the `LogicalArchitecture` diagram.
2. *Paste* it *as* a block type and *as* a text symbol with SDL interface in the `EntranceUnitPackage`.
3. Rename the block type and give it the name `EntranceInterface`.
4. Add a signal parameter called `MessageType` to the `Display` signal definition. By using this type for messages the `EntranceCtrl` does not have to handle strings. This solution makes the system independent of the language used, see "Performing an Iteration" on page 3928 for an example on how this works.
5. Define the NEWTYPE `MessageType`, see Figure 739.
6. Also, *paste* the class `DisplayInterface` *as* a block in the `EntranceUnit` block diagram. Rename the block and give it the name **`theEntranceInterface:EntranceInterface`**.
7. Using the Entity Dictionary, create implementation links between the CardReaderInterface, KeypadInterface and ExitButtonInterface classes in the `LogicalArchitecture` diagram and the `EntranceInterface` block type **and** block.

   Note that you do not have to paste these three classes as SDL interfaces as their operations have already been defined as signals in the `EntranceUnitPackage`.

8. *Now, copy* and *paste* the class `EntranceCtrl` as a block type and as a text symbol with SDL interface in the `EntranceUnitPackage`.
9. Delete the signal `ChangeSecurityLevel` as this already has been defined in the `UtilityTypesPackage`.
10. Add signal parameters where they are needed i.e. to the signals `ReceiveCard` and `ReceiveCode`.
11. Also, *paste* the class *a*s a block in the block type `EntranceUnit` diagram. Give the block the name **`theEntranceCtrl:EntranceCtrl`**.

The `EntranceUnitPackage` will look like in Figure 739.

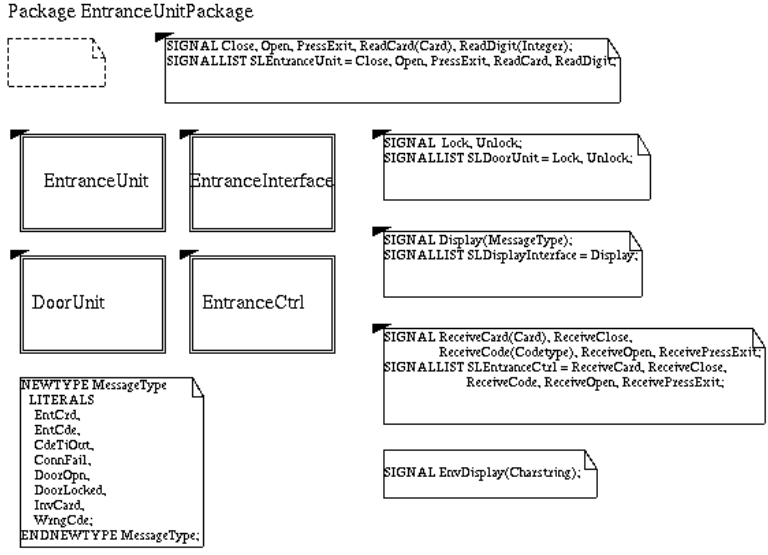

Figure 739: The EntranceUnitPackage

The block type `EntranceUnit` diagram should now contain three blocks: `theDoorUnit`, `theEntranceInterface` and `theEntranceCtrl`.

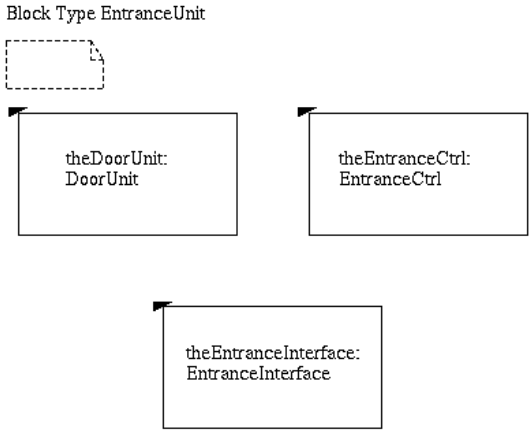

Figure 740: The block Type EntranceUnit

Defining the Communication Structure

Now you should define the communication structures within the block type `EntranceUnit`. Define the channels needed for the three blocks `theEntranceCtrl`, `theEntranceInterface` and `theDoorUnit` to communicate.

To define the channels in the block type `EntranceUnit`, follow the steps described below.

1. Identify necessary channels.
2. Identify the signals that should be carried by the channels. You get a lot of information from the `AnalysisObjectModel` here.
3. Name the channels and the gates.
4. The complete block diagram for `EntranceUnit` should look something like in Figure 741.

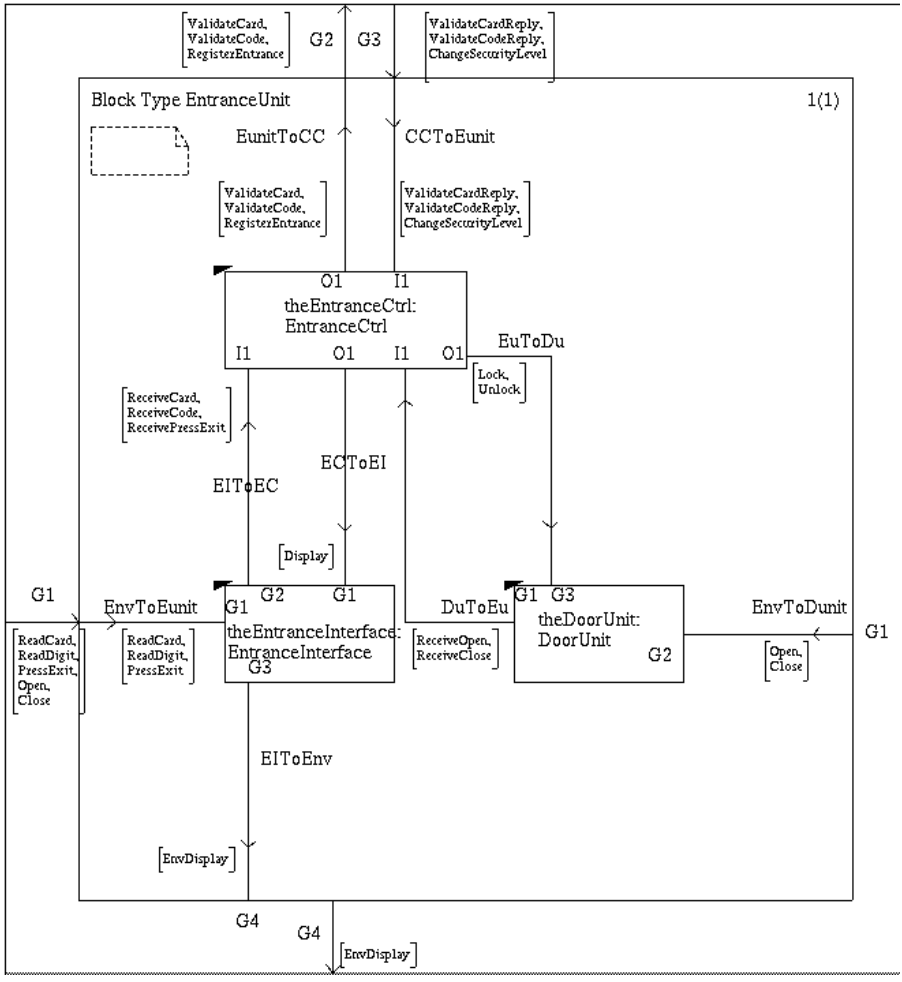

Figure 741: The complete structure of the block Type EntranceUnit

When you have completed the architecture definition, the `System Design Documents` chapter should look like in Figure 742.

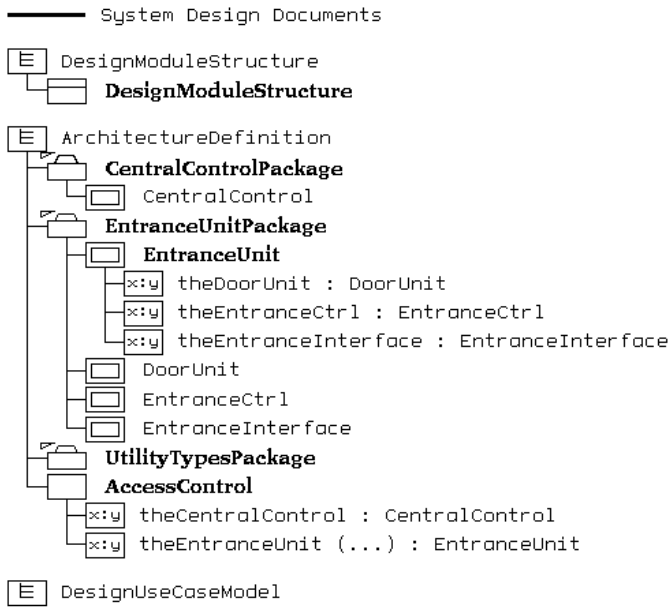

Figure 742: The System Design Documents chapter

Creating the Design Use Case Model

In system design we continue with the use of use cases, this time to define the dynamic interfaces between the blocks in the system.

In our example, the block theEntranceUnit in the system diagram is split into the three blocks, theEntranceCtrl, theEntranceInterface and theDoorUnit. The use cases will consist of the block instances theCentralControl, theEntranceCtrl, theEntranceInterface and theDoorUnit as well as an instance representing the environment.

The use cases must be formalized to a sufficient degree of detail, a level that is consistent with the level of detail found in the static interface definition. Also, the level of detail must be precise enough to make the design use cases act as detailed test specifications.

Formalizing Use Cases

A number of things have to be done when formalizing and refining analysis use cases to design use cases.

- Check that each MSC instance head has a corresponding block in the architecture definition. The use cases define the dynamic interface between the blocks in the system and thus all blocks have to be represented.
- The MSC instances corresponding to actors in the system environment should have **`env_`** stated in their instance head before the actual name, e.g. **`env_Employee`**.
- The MSC instances must have names corresponding to block instances as it is the instances that communicate. That is, the name of those instances that have been defined as block types in the SDL package structure must be changed to the corresponding block instance name. E.g. the MSC instance `CentralControl` can no longer have this name; the name has to be changed to `theCentralControl` according to the architecture definition.
- The messages often have to be replaced with a sequence of message exchanges.
- The MSC messages have to be complemented with parameters according to the definition of signals and remote procedures in the architecture definition. It should not be the **name** of the parameter that should be added but a **value** that the parameter can take. Alternatively the parameters can be skipped entirely.

Creating a Formalized Use Case

Now you should begin to formalize and refine one of the use cases from the system analysis.

1. *Add* a *new* MSC diagram to the `DesignUseCaseModel` module in the Organizer.
2. In the *Add New* dialog, set the toggle button *Copy existing fil*e. As the file to be copied, select the MSC `Enter_Office_With_Card_And_Code_SysA` from the system analysis activity. This file can be found in the `somttutorial/sysanalysis` directory.

3. In the MSC Editor, rename the instance head `Employee` to **`env_Employee`** and change/add the other instance head names. They should conform to the names used on the block instances in the SDL system diagram and the blocks in the `EntranceUnit` block diagram in the architecture definition.

4. For each message, decide if it has to be exchanged with a sequence of messages. If so, add these new messages to the MSC.

5. Add parameters to the messages. Look at the definition of the signals in the architecture definition and add parameters to those messages that are defined to have parameters. E.g. the message `ReadCard` should have a parameter consisting of the card. Note that it is not the parameter name `Card` that is to be used here but an example of a value that the parameter can take, e.g. **`123`**.

6. Rename the use case and save it under its new name, `Enter_Office_With_Card_And_Code_SysD` (use *Save As*).

The exceptions and behavior patterns to this use case must also be formalized.

The other design use cases are created in a similar way. All design use cases should also be connected with implementation links to the corresponding analysis use case and exception. This is done in order to make it possible to check that all use cases from the requirements analysis and system analysis have been refined to design use cases.

Consistency Checks

Now the time has come to performing consistency checks on the models created in system design.

Entity Match

The first thing to check is that the actual modules (SDL packages) used in the design are consistent with the design module structure.

1. In the Link Manager, choose *Consistency Check* in the *Tools* menu and set the *entity match* radio button. Note that you have to be in **entity mode** to be able to perform an entity match.

2. Let the packages in the `ArchitectureDefinition` form the **from** group and the `DesignModuleStructure` module the **to** group.

The result shows that all packages in the `ArchitectureDefinition` really are described in the `DesignModuleStructure`. The consistency view (i.e. the resulting Link Manager window) also shows the contents of the packages. As you can see there are no matching entities to the block references, block types, etc. in the `DesignModuleStructure`. There should not be any, so just ignore this.

Link Check

By doing a link check we will first check that all objects in the analysis object model are mapped to the architecture definition.

1. Once more, choose *Consistency Check* in the *Tools* menu and perform a link check. It does not matter which view you have in the link Manager window, a link check can be performed in either view.

2. Let the `AnalysisObjectModel` form the **from** group and the `ArchitectureDefinition` module form the **to** group.

The result shows that most of the objects from the analysis object model are described as block types and block references in the architecture definition. Many of the objects have also been mapped to an interface definition. The fact that some classes have no corresponding mapping indicates that these classes probably should reside as processes inside some of the mapped blocks. This holds e.g. for the `DoorSensorInterface` and the `DoorLockInterface` as well as for the `SecurityLevel` classes.

At this point you can also select any block or block type in the architecture definition and choose *Traverse link*. The corresponding class in the analysis object model will then be selected. By choosing *Traverse link* again you can follow a link all the way back to the textual requirements. It is also possible to follow a link in the other direction, i.e. from the textual requirements via the object models, to the design. Try this!

Summary

After having completed an entire system design the corresponding document structure in the Organizer would look like in Figure 743.

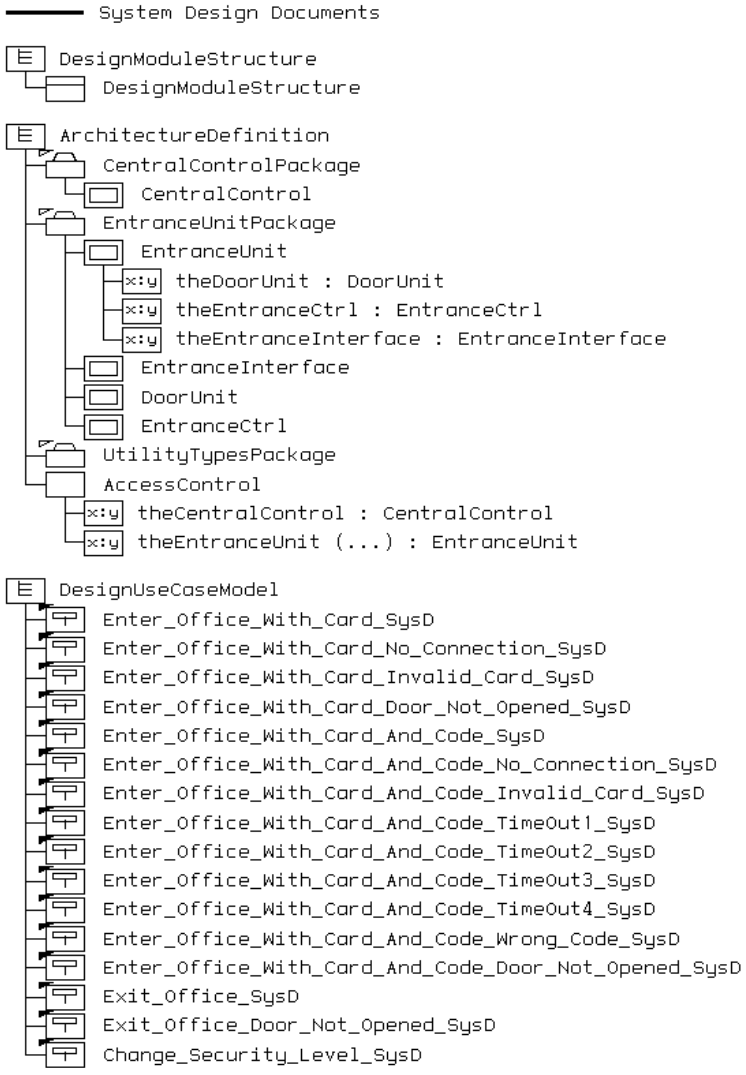

Figure 743: The entire System Design Documents structure

Performing the Object Design

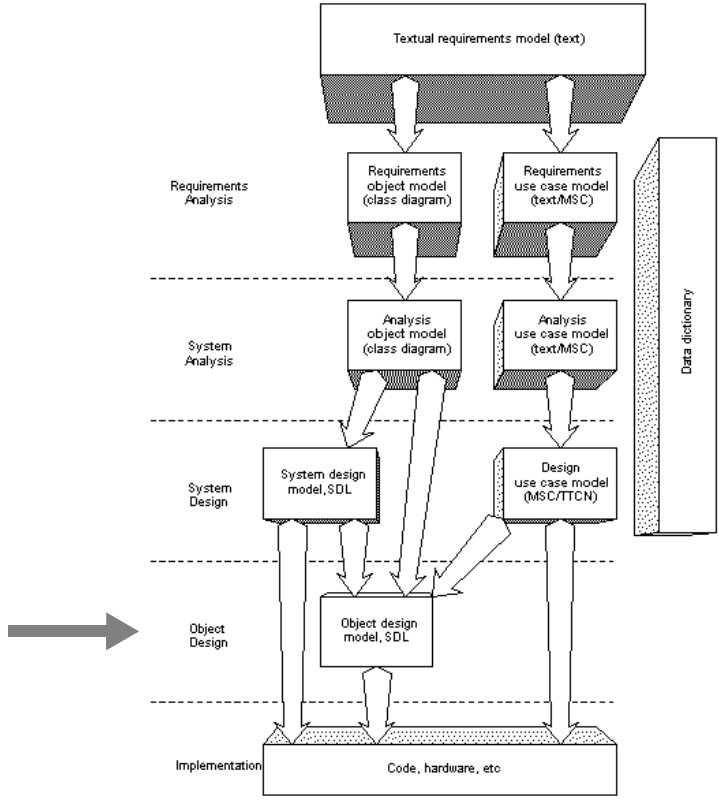

Figure 744: Overview of the SOMT process

What You Will Learn

- To transfer the analysis object model into a consistent object design model in SDL
- To use the *Paste As* functionality to assist the task
- To perform design level testing

Introduction to the Exercise

In this exercise you will perform the object design activity. This activity is, like the system design activity, focused on SDL. However, while the system design is focused on how to structure the architecture and how to decompose the system into blocks, the object design is focused on decomposing the blocks into processes and defining the behavior of the single processes.

The object design activity may be divided into three separate tasks:

1. Map the classes and associations in the analysis object model to suitable SDL concepts.
2. Choose a set of essential use cases and define the behavior of the SDL processes and data types that implement these use cases.
3. Elaborate the design by introducing more use cases and refine the SDL design to handle also these use cases.

Preparing the Exercise

The input to this activity should be a complete requirements analysis, system analysis and system design structure.

1. Open the system file `somttutorial/ObjD/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\objd\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/ObjD/` **(on UNIX)**, or `somttutorial\objd\` **(in Windows)**.

Mapping Active Objects to SDL

An object with its own behavior is called active object. The opposite is an object which acts as an information container - a passive object. Active objects are, most often, mapped to SDL process types. Active objects may also sometimes, as was the case in the system design activity, be mapped to block types.

The default choice in the *Paste As* mechanism is to paste a class copied from an object model diagram as a process type in an SDL diagram.

The attributes of the copied object will be pasted as variables. The operations will be pasted either as signals or as remote procedures of the

process type. This depends on whether the operations are synchronous or asynchronous.

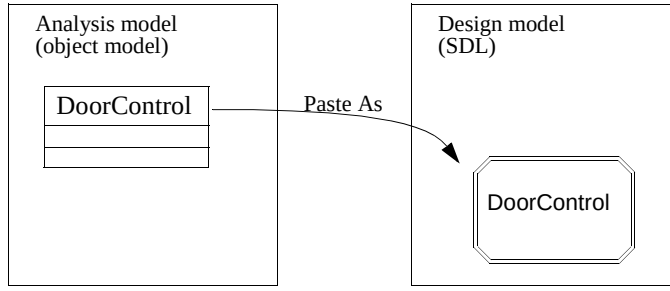

Figure 745: Mapping a class to a process type

Mapping to Process Types

Now it is time to map the classes in the analysis object model that should be mapped to SDL process types. In this example you should map the processes which will reside within the block type EntranceInterface.

The classes which should reside as processes in the block `EntranceInterface` are: the `CardreaderInterface`, the `KeypadInterface`, the `DisplayInterface` and the `ExitButtonInterface`.

1. *Copy* and *paste* each class from the `LogicalArchitecture` diagram as a *Process Type* in the `EntranceUnitPackage`.
2. *Open* the block type `EntranceInterface` diagram in the SDL Editor by double-clicking on the corresponding block type symbol in the `EntranceUnitPackage`. Press *OK* in the Edit dialog. In the Add Page dialog choose to create a *process interaction* page.
3. *Copy* each class once again and *paste* it as a *Process* in the block type `EntranceInterface` diagram.

When pasting e.g. the class `KeypadInterface` as a process, the process will get the same name as the class. This name should be changed since the syntax for process instances requires both an instance name and the name of the corresponding process type. The number of statically and dynamically created instances must also be stated.

4. Name the process **`theKeypadInterface`**. The process `theKeyPadInterface` should have one statically created instance and it should not be possible to create any instances dynamically. The following text should thus be written in the process name area: **`theKeyPadInterface(1,1):KeyPadInterface`**.

   (If the *Remove Reference Symbol* dialog appears, just click *OK*.)

5. Change the other three process names too, according to the above.

Now the block type `EntranceInterface` should contain four processes named: `theCardreaderInterface`, `theKeyPadInterface`, `theExitButtonInterface` and `theDisplayInterface`.

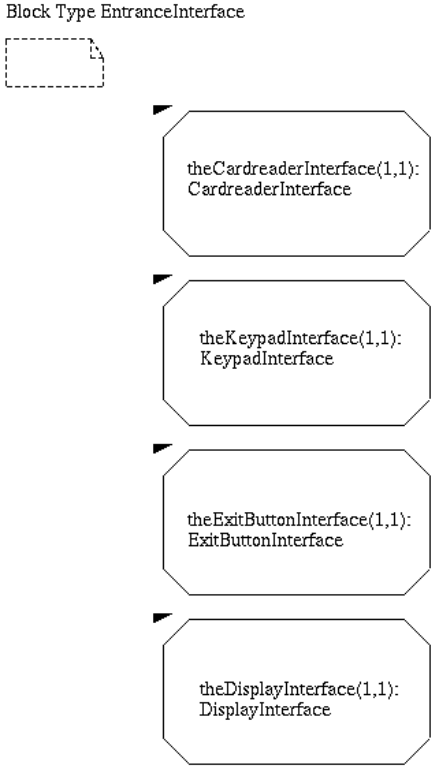

Figure 746: The block type EntranceInterface

Defining the Communication Structure

You have already specified the channels to and from the block `theEntranceInterface` (this was done in the block type `EntranceUnit`). Now you have to connect the processes within this block with the outside.

Creating a communication structure between processes and the border of the block is made in the same way as creating communication structures between blocks, with one difference. The terminology specifies *signal routes* instead of channels as the name for the communication structures at this level. However, there is no practical difference between signal routes and channels.

Now, edit the block type `EntranceInterface` diagram:

1. Connect each process with the border of the block with a signal route in each direction.
2. Give the gate of the input signal to each process the name *Entry* and the other gate the name *Exit*.
3. Specify the signals that are transported on each signal route. Consult the system analysis object model and the specification of the block `theEntranceUnit` to find all signals you should specify.
4. Connect the signal routes with the channels by specifying the appropriate gate for each signal route.

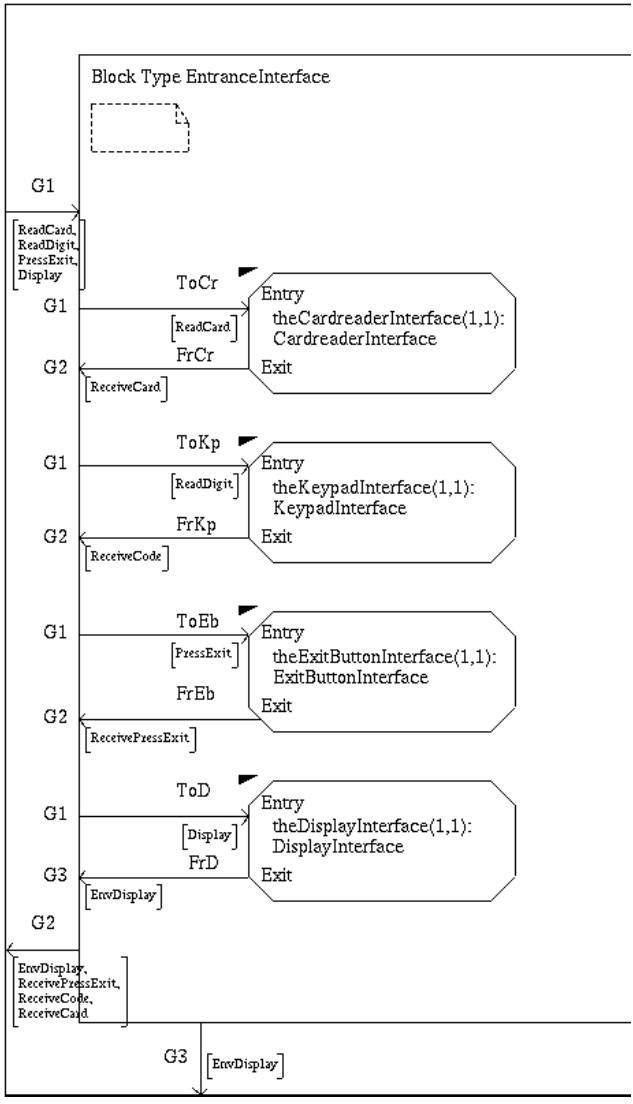

Figure 747: The complete structure of the block type EntranceInterface

Defining the Object Behavior

The activity of describing object behavior is a pure SDL design activity. This section is intended to describe how to structure this activity. It will not focus on specific SDL details.

The most important source of information in this activity are the use cases which specify the signal calls and responses to and from the blocks and processes in the system.

The design of the processes is best made iteratively:

- Select a subset of use cases which describe the most common interaction sequences. Create a basic version of the processes that correspond to these use cases.
- Second, you should introduce a couple of more use cases. Edit the behavior of the processes making them correspond to the extended subset of use cases.
- Continue with introducing more use cases. Edit the processes to cover these, until the behavior of the objects correspond to the complete set of use cases.

Using MSCs to describe the use cases makes it fairly simple to identify the states and transitions of the processes. A transition in a process graph is an input signal, often followed by one or more output signals from the actual process in a use case.

If the situation occurs that two use cases are difficult to combine, you should consider to split the process in question into two separate processes, one process for each use case.

An example is the block type `EntranceInterface` which have a quite complex structure of input and output signals. However, by dividing the block type into four separate processes, each one of these processes becomes fairly simple.

Defining the Basic Behavior of a Process

1. Take a look at the process type `KeypadInterface`. You will see that the process has a signal set of only two signals: the input signal `ReadDigit` and the output signal `ReceiveCode`.

2. If you study the MSC diagram `Enter_Office_With_Card_And_Code_SysD` and the behavior pattern ReadCode you see that four `ReadDigit` signals generate the output signal `ReceiveCode`.

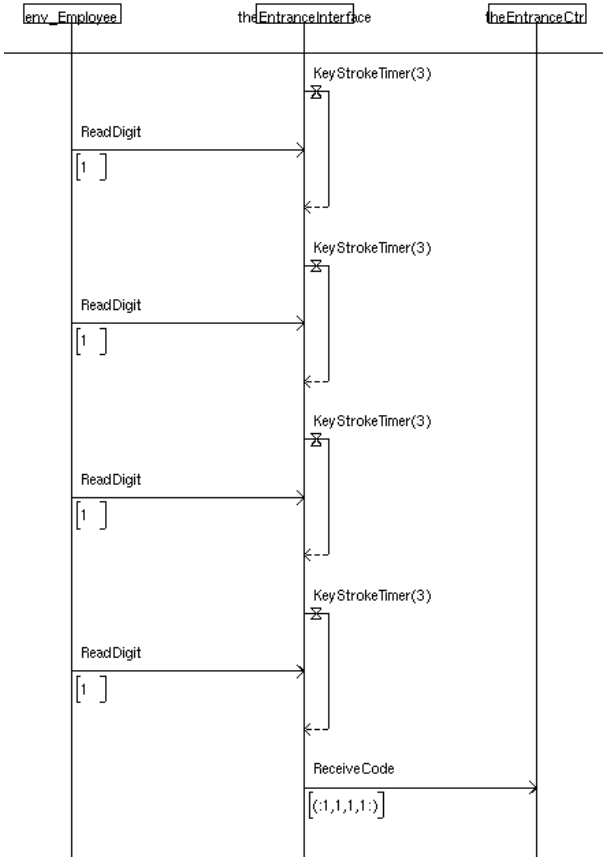

Figure 748: The signal sequence of the process theKeyPadInterface

Creating the behavior of the process out of this information is a quite easy task. Adding the timer, which provides the time-out functionality, will make the first version of the behavior specification of the process complete.

3. In the `EntranceUnitPackage`, double-click on the `KeypadInterface` process type symbol.

4. In the process type diagram, define the basic behavior of the process.

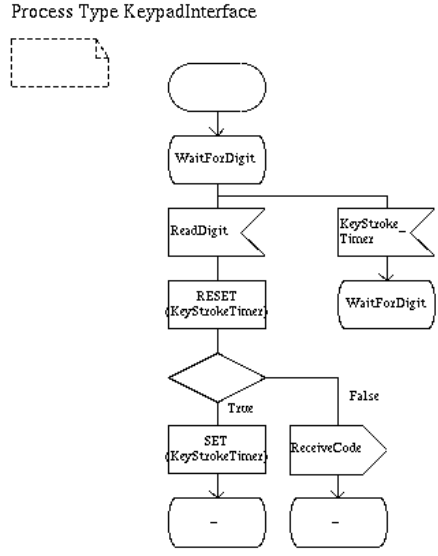

Figure 749: Basic behavior of the KeypadInterface process type

Defining the Data of a Process

The processes will of course also need some data containers, entity variables and control variables. The control variables, such as loop counters and flags, are identified during the object design. Entity variables are most often identified during the system analysis and they may be mapped to the process diagrams from the analysis diagrams.

If we take a look at the process `theKeyPadInterface` again we will notice that we need to introduce a counter to control the number of times a digit will be read before the signal `ReceiveCode` will be sent. We will

also need an index to place the read digit in the correct position of the `Code` array.

1. Place a text symbol in the diagram and declare a `CodeIndexType` named `i` in the process.
2. The parameter of the `ReadDigit` signal is of type integer. Declare a variable named `Digit` of type integer.
3. The parameter of the `ReceiveCode` signal is of type `CodeType`. Therefore, declare a variable `Code` of this type.
4. Change the name of the gate `GKeypadInterface` to `Entry`.
5. Also, add an `Exit` gate and the signal `ReceiveCode` going out of the process type.
6. Declare a timer `KeyStrokeTimer` with duration 3.
7. Refine the behavior of the process.
8. Save the diagram.

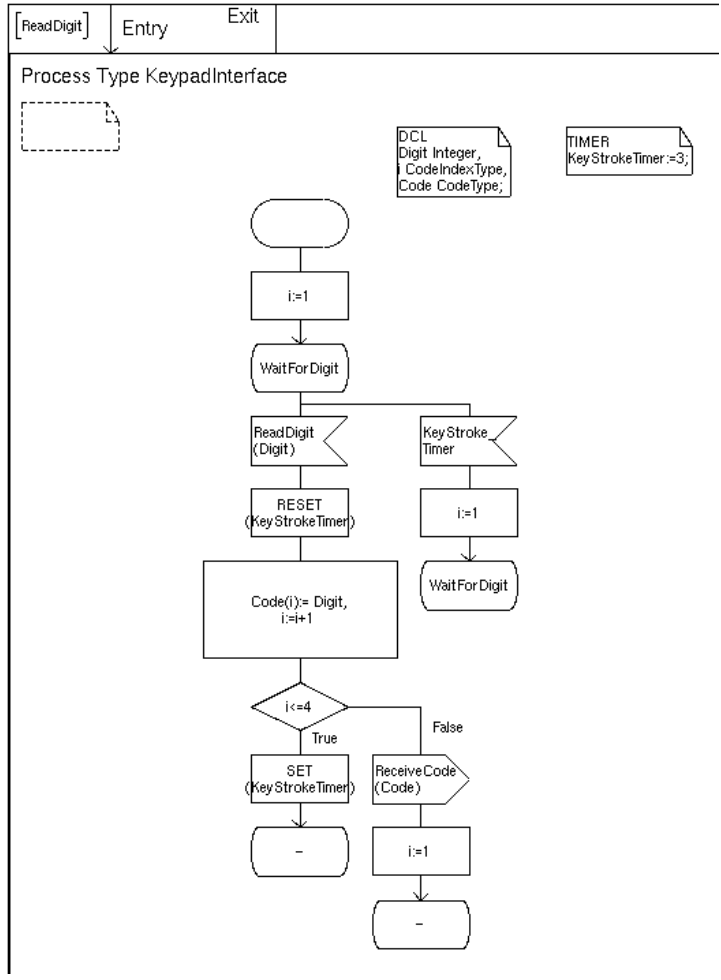

Figure 750: The complete process type KeypadInterface

The rest of the processes in the system are created in a similar way. However, this will not be done in this tutorial.

Design Testing

SDL makes it possible to test the system already during the design. It is possible to simulate an SDL system taking both concurrency and distribution into account. It is also possible to verify requirements specified in MSCs using the Validator.

The MSC diagrams may, with no or little effort, be used directly as input to the Validator. This makes the requirements verification simple and efficient.

Preparing the Design Testing

To be able to test your design you must have a complete system.

1. Open the system file
   `somttutorial/objdesign/accesscontrol.sdt` **(on UNIX)**, or
   `somttutorial\objdesign\accesscontrol.sdt` **(in Windows)**.
   What you see in the Organizer now is a complete system structure. All four documentation chapters representing activities from the SOMT method are complete.
2. Check that the Source directory is set to
   `somttutorial/objdesign/` **(on UNIX)**, or
   `somttutorial\objdesign\` **(in Windows)**.

Simulating the System

Simulating the system gives information about how different parts respond to certain inputs and how different parts of the system interacts with other parts. Simulating is often done during the object design to test different parts of the system.

The complete system should also be tested using the simulator to verify that the whole system works as it is intended to.

Try to simulate the system AccessControl. Follow the steps below to make a simulator version of the system AccessControl.

1. Select the system AccessControl in the Organizer.
2. Choose the *Make* option in the *Generate* menu.
3. In the Make dialog, choose the code generator to be *Cbasic*.
4. Choose the standard kernel to be *Simulation*.

5. Press *Full Make*. The system will be automatically analyzed. If there are any warnings you can ignore them. These warnings show up because we have not used all our declared signal lists. As they are just warnings, not errors, you can ignore them.

6. When the make process is finished, start the Simulator by choosing Simulator UI in the *SDL* submenu in the *Tools* menu.

7. In the SDL Simulator UI window, open the file `accesscontrol1_smb.sct` **(on UNIX)**, or `accesscontrol1_smb.exe` **(in Windows)**. Now you can simulate the system, as you have learned in previous tutorials.

Validating the System

When the design of the system is finished you want to verify that the system meets the requirements. This is quite easily done in the SDL suite by using the SDL Validator.

MSCs from the system design are used as input to the Validator.

The requirements use case model is the essential part of the requirements and is often the specification of the system which the customer and the contractor agree upon. Thus, by verifying the system with the MSCs from the system design you are verifying that the system meets the customers requirements.

By making it possible to verify the customers requirements already in the object design and not in a special system design test phase, as in an ordinary design process, you save a lot of effort and time.

Now you should validate some of the MSCs from the system design activity.

1. Select the system AccessControl in the Organizer.

2. Choose the *Make* option in the *Generate* menu.

3. In the Make dialog, choose the code generator to be *Cbasic*.

4. Choose the standard kernel to be *Validation.*

5. When the Make process is finished, start the *Validator UI* and open the file `accesscontrol1_vlb.val` **(on UNIX)**, or `accesscontrol1_vlb.exe` **(in Windows)**. Now the system is ready to be validated.

6. Press the *Verify MSC* button and choose the system design MSC diagram `enter_office_with_card_sysd.msc` which you can find in the directory `somttutorial/sysdesign`.

7. If the verification succeeds, you will get the message "** MSC <Diagram name> verified **".

It is perhaps too much work to verify all the MSC diagrams of the system design activity during this tutorial and you may quit when you feel that you have understood the principle of how to verify an SDL system.

If all diagrams can be verified against the system, then it is verified that the system also meets the requirements specified in the beginning of this tutorial.

Consistency Checks

There are mainly two consistency checks that should be performed in the object design activity:

- Check that all objects from the analysis object model have been implemented in the design.
- Check that the design model correctly implements the requirements from the design use cases.

The second consistency check is done through design level testing, see "Design Testing" on page 3922. The first one will be performed through a link check, see below.

Link Check

In our complete Access Control system there are implementation links from the system analysis models to both the system design models and the object design models. This implies that we must check our analysis object model against both these design models to see if all the classes have been implemented in the design.

- Perform a link check. Let the analysis object model form the from group. The architecture definition and the SDL design model will form the to groups.

The result shows that all classes but the class `SecurityLevel` have corresponding processes, blocks, signal interfaces or procedures in the design. The `SecurityLevel` does not have any behavior of its own, it is

implemented through its subclasses, and, therefore, the result is just as we want it.

We have now completed the design of the system. It is possible to pick an endpoint in the textual requirements and follow the implink from it, through the object models, to SDL. It is also possible to traverse links the other way, i.e. from design to requirements. Try this!

The use cases are also connected to each other, from requirements to design, as can be seen in the view of the link file in the Link Manager window.

Summary

After having completed an entire object design the corresponding document structure in the Organizer would look like in Figure 751.

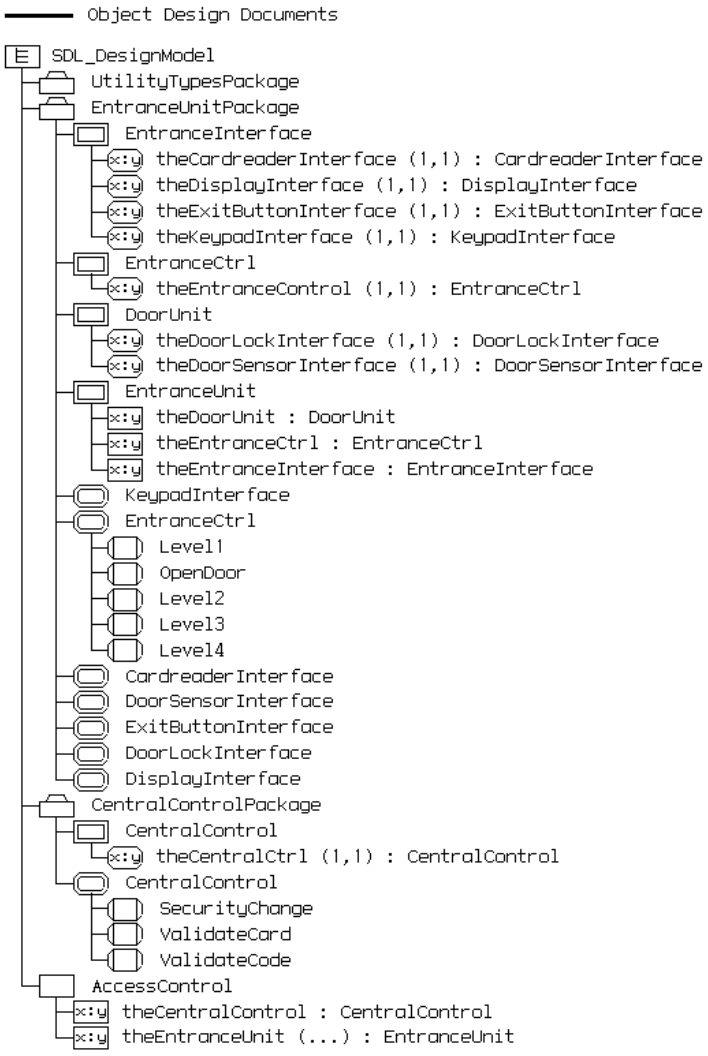

Figure 751: The entire Object Design Documents structure

Implementation

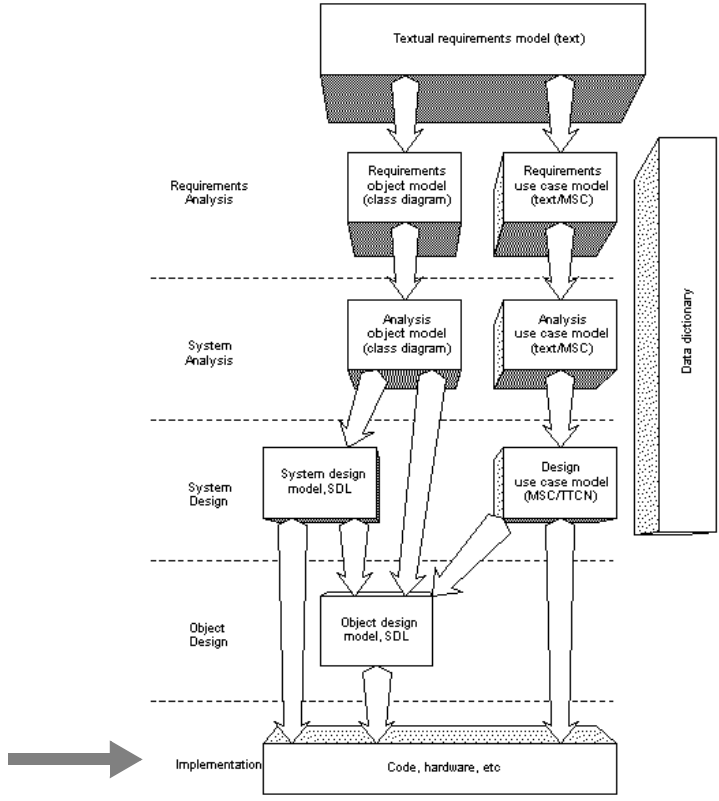

Figure 752: Overview of the SOMT process

The implementation of the system lies outside the scope of this tutorial. For information about the implementation activity, see the SOMT Methodology Guidelines starting in chapter 69 in the User's Manual.

Performing an Iteration

What You Will Learn

- To introduce changes to a system in a controlled way
- To use the *implementation links* to assist you in the activity

Introduction to the Exercise

During the life time of a system new requirements and changes in existing requirements are almost always introduced. We have to be able to handle these requirement changes and adapt the system to the new situation in a controlled way. We call this process iteration. An iteration may also be planned in advance in, for example, incremental development.

This section will describe the scenario of an iteration caused by the introduction of an additional requirement.

Preparing the Exercise

As input to this exercise we will use the complete Access Control system.

1. Open the system file `somttutorial/Iter/accesscontrol.sdt` **(on UNIX)**, or `somttutorial\iter\accesscontrol.sdt` **(in Windows)**.
2. Check that the Source directory is set to `somttutorial/Iter/` **(on UNIX)**, or `somttutorial\iter\` **(in Windows)**.
3. Create a new chapter and name it **`Iteration Documents`**.
4. Add a new module called **`AdditionalTextualRequirements`** to the new chapter.
5. Add the existing file `AdditionalTextualRequirements.txt` to the new module.

Studying the Additional Requirements

- *Open* the additional textual requirements document if it is not already open. The example below shows the document.

Example 612: Additional requirements ––––––––––––––––––––––––

The system should be able handle the languages English, German and French. One version of the system should handle a specific language and the system should be easily configured to handle a new language.

––

As stated in the example above, the task is to redesign the system so it can handle different languages. The design should be made in a way that makes it easy to configure the system to new languages.

The words English, German and French are marked as endpoints in the document.

Examining the Consequences

Now it is time to validate what consequences the new requirement has on the system. The new requirement identify one object that may be affected, the `Display` object.

1. Study the requirements regarding the `Display` in the original textual requirements document. You should find that the new requirement does not contradict with the original requirements.
2. *Traverse* the implementation link from the text fragment `Display` in the original textual requirements. The logical structure diagram in the requirements object model will pop up with the class Display selected.
3. With the class still selected, choose *Traverse Link* once more and follow the link to the logical architecture diagram in the analysis object model. The class `DisplayInterface`, with operation `Display` and attribute `Text` will be selected. The class is a part of the class `EntranceUnit`. It also has a connection to the class `EntranceCtrl`, see Figure 753.

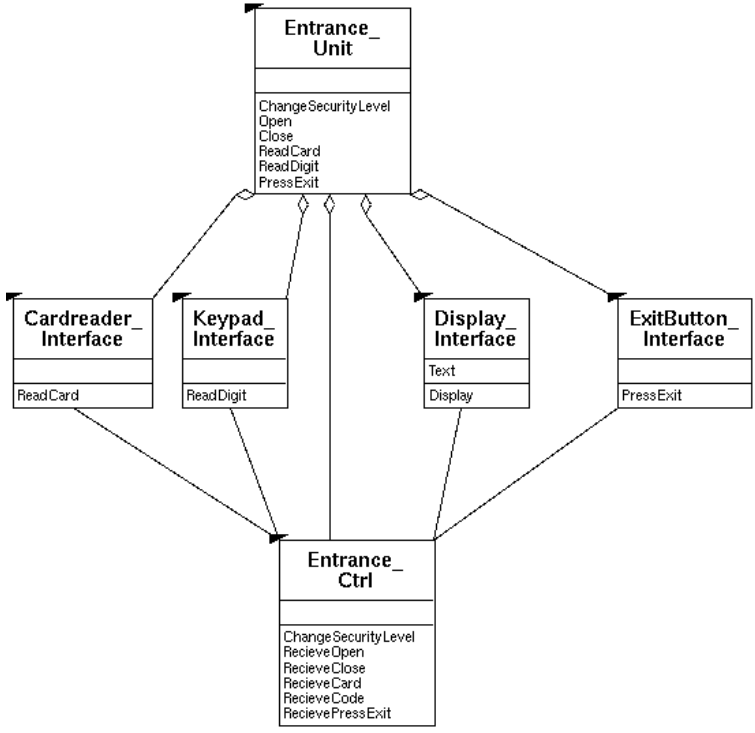

Figure 753: Part of the Logical Architecture diagram

4. Continue to follow the link of the class `DisplayInterface`, from the logical architecture to the package `EntranceUnitPackage`. The class is connected with the block type `EntranceInterface` and the process type `DisplayInterface` as well as with the signal interface defining the signal `Display`.

5. You can also follow the implementation links from the class `DisplayInterface` to the block `theEntranceInterface` (within the block type `EntranceUnit`) and to the process `theDisplayInterface` (within the block type `EntranceInterface`).

6. *Open* the block type `EntranceInterface` in the `Object Design Documents` structure. Study the signal routes leading to and from the process `theDisplayInterface`. You will notice two signals:

`Display` and `EnvDisplay`. Examining the two signals with help of the Signal Dictionary will give you information about the signals. `EnvDisplay` has the parameter `Charstring` and the signal `Display` has the parameter `MessageType`.

7. *Open* the process type `DisplayInterface`. Study the behavior. You will notice that it is possible to change the content of the `EnvDisplay` signal without affecting the behavior of the process type. It also seems like the required changes to the system are limited to the process type `DisplayInterface`.

Introducing Changes in Documents

Now you should introduce the necessary changes to the system to get the desired behavior. Changes should be made in a controlled way. All documents affected of the changes should be edited, to keep the consistency.

Updating the Data Dictionary

1. *Open* the data dictionary. Include information presented in the additional requirements to the data dictionary.

2. *Save* the file as `DataDictionary.txt` in the current directory.

Updating the Requirements Object Model

As we saw earlier, the change we have to introduce to the system is focused on the process type `DisplayInterface`. The new requirement specify that one version of the system should handle a specific language and the only language dependent part of the system is the process type DisplayInterface.

It seems natural to introduce a class for each one of the languages English, German and French in the logical structure diagram. These classes should be subclasses to the class `Display`, i.e. an inheritance structure is needed.

1. *Add* the three new classes to the logical structure diagram in the requirements object model. Name the classes **`FrenchDisplay`**, **`EnglishDisplay`** and **`GermanDisplay`** respectively.

2. *Link* the classes with the corresponding text in the additional textual requirements document.

3. *Save* the diagram *as* `logicalstructure.som` in the current directory.

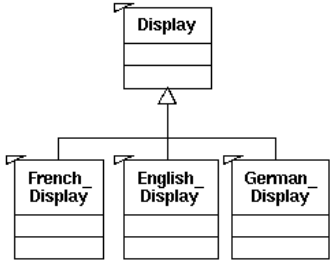

Figure 754: The class Display and its subclasses

4. Make a link from the class `Display` to the class `DisplayInterface` in the logical architecture diagram in the analysis object model.

Updating the Analysis Object Model

In the logical architecture diagram we have to create an inheritance hierarchy corresponding to the inheritance hierarchy in the logical structure.

1. Create the three subclasses to the class `DisplayInterface`. Name the classes **`EnglishDisplayInterface`**, **`GermanDisplayInterface`** and **`FrenchDisplayInterface`**.

2. *Create* implementation *links* to the corresponding classes in the logical structure diagram. (An alternative here would have been to *copy* the classes from the logical structure diagram and *paste* them *as* classes in the logical architecture diagram. The implementation links would then have been created automatically.)

3. *Traverse* the *link* from the class `DisplayInterface` to the process type `DisplayInterface` in the `EntranceUnitPackage`.

Updating the SDL Design

Now it is time to make some changes to the SDL design, making it correspond to the logical architecture. The inheritance hierarchy of the `DisplayInterface` classes should be mapped to an inheritance hierarchy of process types.

1. Create two new process types in the `EntranceUnitPackage`. Name the process types **`GermanDisplayInterface`** and **`FrenchDisplayInterface`**. The already existing `DisplayInterface` process type will function as the `EnglishDisplayInterface`.

2. Mark these process types as endpoints and create links between them and the corresponding classes in the logical architecture. (Alternatively, use the Copy-Paste As mechanism.)

   Now, edit the `DisplayInterface` process type to make it more general and suitable for reuse.

3. Double-click on the `DisplayInterface` process type in the `EntranceUnitPackage`.

4. Create a variable of type `charstring` for each possible message that can be sent to the environment. You will need eight such variables.

5. Put the text **`Virtual`** in the start symbol.

6. Insert a task symbol just after the start symbol.

7. In the task symbol you should initialize the message variables to the corresponding message.

8. Edit each output symbol making it use the corresponding message variable as a parameter instead of a text string. The process type `DisplayInterface` will use English language.

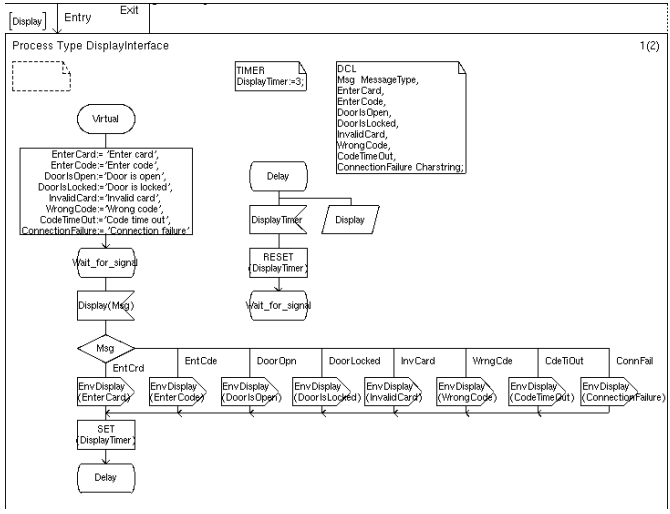

Figure 755: The process type DisplayInterface

9. *Save* the diagram in the current working directory.

After editing the `DisplayInterface` process type it is time to create the process types `GermanDisplayInterface` and the `FrenchDisplayInterface`. For each process follow the steps below:

10. Open a graph page for each process type by double-clicking on the corresponding process type symbol in the `EntranceUnitPackage`. Press *OK* in the edit dialog.

11. In the additional heading you should enter the text:
    `INHERITS DisplayInterface ADDING;`

12. Place a start symbol in the diagram and enter the text **`REDEFINED`** in the symbol.

13. Connect a task symbol to the start symbol. The task is to initialize the message variables to the appropriate messages. Do not use national characters.

14. Connect a state symbol with the task symbol and name it **`Wait_for_signal`**.

15. Save the diagrams.

Now the necessary behavior is described and it is time to show how to configure the system to make use of the new design.

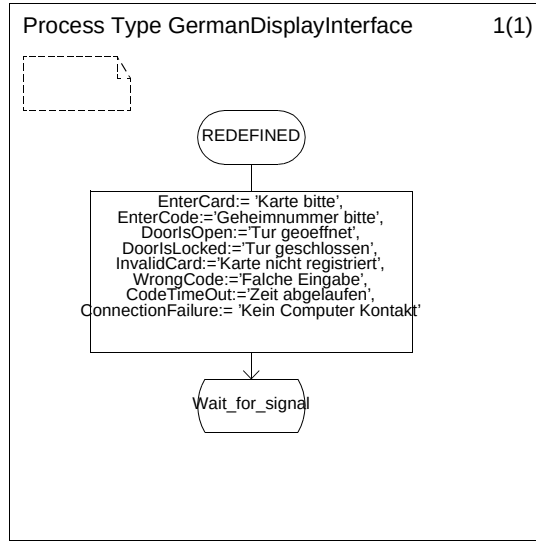

Figure 756: The process type GermanDisplayInterface

Configuring the System

The necessary design is done and we want to configure our system to make use of the new design. Following the steps below will configure the system to a German version.

1. Open the block type `EntranceInterface`.
2. Select the process `theDisplayInterface` and change the text `theDisplayInterface(1,1):DisplayInterface` to **`theDisplayInterface(1,1):GermanDisplayInterface`**
3. Now you can analyze the system. Perhaps you want to simulate the new version of the system; if so, follow the steps described in "Simulating the System" on page 3922.

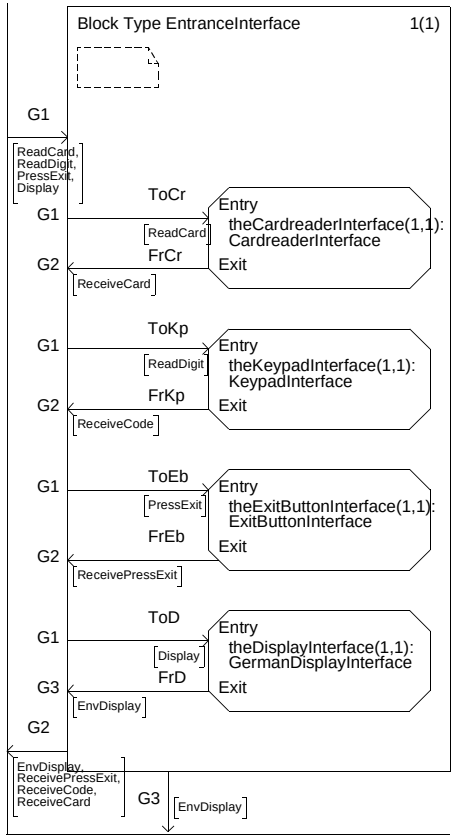

Figure 757: The block type EntranceInterface

Now you have completed the iteration exercise.

To Conclude...

You have now learned the steps of the SOMT method, and we hope you have enjoyed the tour.

Once again we would like to point out that the activities are presented in a sequential order in this tutorial just to simplify the reading. In practise, the task of developing a system using SOMT is a highly iterative process. One activity may start before the preceding activity is completed and the models inside an activity are usually created in parallel.

The SOMT method is intended to support the development process, not to control it. In other words, it is a **proposed** way of working. For your own work, you should not feel that you are locked by SOMT, but pick the parts that suit you best.

Chapter 78 SOMT Tutorial