# System Development Framework

## Handbook for Version II

W.R. van Joolingen
A. Th. Schreiber

# Contents



 

 

## Preface

In 1993 the system development framework was developed by CIBIT as an educational device, in order to bring the principles of knowledge modelling, knowledge engineering and knowledge technology to students of the Master of Science programme in information and knowledge technology. The target audience of the framework consisted of software and information engineers who wanted to study modern developments in knowledge technology. In order to teach people how proceed with knowledge technology, one should teach them a methodology, a coherent set of methods and techniques that prescribe what to do in order to build knowledge based systems. The prevailing methodology for knowledge engineering in those days was KADS, developed at the SWI, the department of Social Science Informatics at the University of Amsterdam, created as the result of a series of EC-funded research projects. The problem with this methodology was that the documentation available was very research oriented, and not directed towards practical application. Another drawback was KADS' tendency towards isolated knowledge based systems, rather than knowledge intensive parts of mainstream information systems.

Given the value of KADS, CIBIT decided to create a new methodology based on the strong points of KADS and the solid value of a mainstream method for systems analysis: Yourdon Systems Method (YSM). The result, SDF version I, was a methodology largely based on YSM, but with the possibility to mark processes as knowledge intensive and to analyse these in a way specified by KADS. Later, SDF was bought in licence by two companies, Bolesian and Everest, to use as the basic methodology for the development of knowledge intensive systems.

Times have changed. YSM is no longer as mainstream as it used to be. Object oriented methodologies have taken the lead. Moreover, KADS itself is reincarnated as CommonKADS, with its own manual (Schreiber et al., 1999) written for a larger audience. These movements create the need for a new look on SDF. Still there is need for integration, only at a different level. CommonKADS itself has become much more useable by mainstream developers (who have developed themselves as well), but is not intrinsically object oriented. OO-methodologies do not explicitly address the issue of knowledge intensive systems. Not everyone wants to use OO anyway. These needs issued the need for a new version of SDF: SDF-II, for which the handbook lies before you.

The level of integration is higher, the contributing methods are left in its value. No attempt has been made to really integrate CommonKADS with other methods. Instead, we offer a framework for combining CommonKADS with any UML-based methodology.

SDF-II targets any system developer involved with systems that require explicit attention to the role of knowledge. SDF-II allows users to use their own UML-based method. It adds instruments to this method to analyse the business and the knowledge involved in the task to be carried out by the system. The knowledge analysis gets a flavour of Catalysis, one of the emerging OO/CBD methodologies

 

(D'Souze & Wills, 1999). In this way SDF-II can help anyone with the analysis and design of knowledge intensive systems.

This handbook was written as a result of a co-operation of several people, and of different companies. The authors would like to thank the fruitful discussions with partners in the CUPIDO platform, and in particular Gertjan Beijer, Mark Willems, and Ron Korevaar. Arie den Ouden made a significant contribution to the section on project management by reworking our original version from a process-based to a product-based approach.

Special thanks go to Patrick Vorgers, who, as an assignment for finishing his MSc. joined our group and did a great job in summarising the discussions and asking the right questions at the right moment.

Utrecht, January – June 1999

 

# Introduction

## *Aim of This Handbook*

SDF-II is a system-development framework that enables software analysts and developers to use knowledge modelling techniques within an object-oriented setting. SDF-II is an integration framework and not yet another methodology. This manual provides guidelines and recipes for using proven knowledge-technology methods within an O-O approach.
SDF-II is based on two pillars. Firstly, it takes the UML notations as a *de facto* standard for describing system-analysis models in object-oriented fashion. It is reasonable to expect that future generations of software analysts will have knowledge of UML in their standard repertoire. Secondly we adopt the basics of the knowledge-modelling approach followed in CommonKADS. This methodology is a proven approach for knowledge-system development. A short introduction into UML and CommonKADS is given further on in this section.

## *Target Audience and Reader Background*

This manual is aimed at system analysts who want to analyse and model knowledge-intensive problems from an object-oriented perspective. We assume that the reader has a background in information modelling.

As said before, SDF-II is a methodology integration framework. To be able to use this handbook some basic knowledge about UML and about CommonKADS are required. From the UML side we assume you know the basic analysis notations:

- class diagram,
- use-case diagram,
- activity diagram,
- state diagram,
- sequence diagram, and
- collaboration diagram.

For CommonKADS we assume that you are acquainted with the following aspects:

- basic modelling principles,
- business-modelling techniques,
- knowledge-modelling framework, and
- knowledge-modelling templates (“patterns”).

For UML you can use the “UML User Guide” as baseline text (Booch *et al*., 1998). For CommonKADS you can find the information in Chapters 2, 3, 5 and 6 of the CommonKADS textbook (Schreiber *et al*., 1999).

In designing the framework, much inspiration and some notational utilities were used from the *Catalysis* method (D’Souza, 1999). This method provides a new view on modelling object oriented and component based systems. By adding a few notations, realising that in many cases it is best to postpone decisions on where to allocate functionality and making a strong distinction between classes and types, this method contributed significantly to shaping the framework. However, no prior knowledge on

 

this method is required to understand this manual. Wherever notations are used that are specific to Catalysis, they will be explained.

### *Knowledge Technology and CommonKADS*

Knowledge technology has come a long way since the early days of “expert systems”. The new generation of knowledge-engineering approaches that appeared around the mid-eighties has matured. Also, the methods used in knowledge engineering have come much closer to mainstream software engineering. This makes the use of knowledge technology much more feasible than before, both from a technical and from a business perspective.

CommonKADS (Schreiber et al., 1999)is the best-known representative of the new generation knowledge technology. CommonKADS is the result of some 15 years of R&D effort from a group of companies and academic institutions, working together in a sequence of ESPRIT projects funded by the European Commission. Recently, a textbook has been published that describes the basic approach (Schreiber et al., 1999). In this handbook we use part of the CommonKADS textbook as baseline (see previous section). The CommonKADS book uses UML as a baseline notation for many of the concepts it uses, which makes it well suitable for the purposes of SDF-II. In this handbook we take this in approach one step further, and show to you how you can use CommonKADS *with only UML notations*, and modelling the complete system within an object oriented paradigm. This handbook introduces knowledge modelling into the world of object oriented and component-based development.

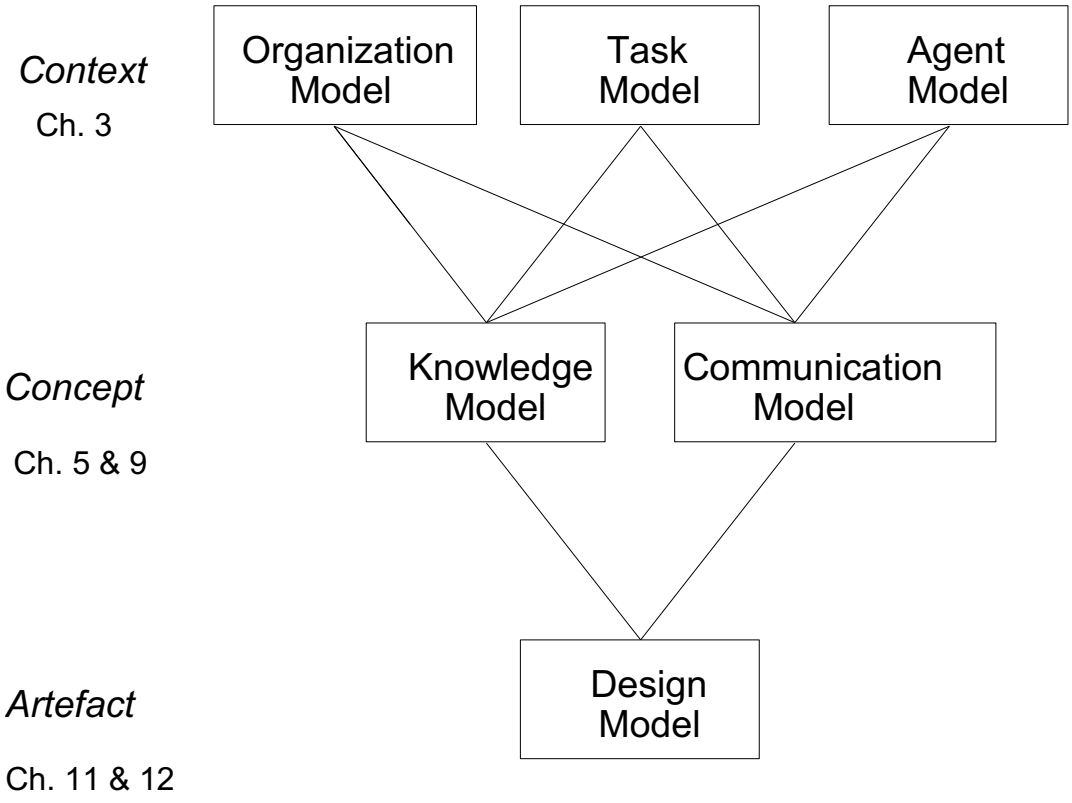


**Figure 1** The CommonKADS model set. The chapter numbers refer to the chapters in Schreiber et al. (1999).

CommonKADS places, like all modern system-development frameworks a major emphasis on system analysis and on reuse. The basic mechanism for this is dividing the complete model of the world into a set of models. Figure 1 shows the CommonKADS model set as defined in Schreiber et al. (1999). Six models define the essence of a knowledge intensive information system, its context and its design. The models serve as complementary views on the world, for instance the organisation model provides a view on the organisation seen in its context and the role that the system should play within this context. This provides a purely outside view on the system. On the other hand and *inside* view is provided by the knowledge model which describes the essence of the knowledge intensive processes in the system.
Reuse in CommonKADS is provided by the presence of *template models*. For reoccurring knowledge intensive processes, CommonKADS provides standard models that can, possibly after fine-tuning for a specific situation, be used in many cases. For instance, knowledge intensive tasks like assessment, assignment or diagnosis, can be described relatively independent of their context. By generalising their descriptions and making them suitable for re-use, CommonKADS relieves the burden of having to reinvent the wheel when you encounter a new situation. The idea is that, for instance, the common features of diagnosing a patient and diagnosing a car are abstracted into a model template that can be used in both situations.

### *The Object-Oriented Paradigm and UML*

One of the major recent innovations in software analysis and design is the introduction of the object oriented (OO) paradigm. In order to deal with the complexity of large information systems, the OO paradigm describes these systems as a collection of more or less independent objects that communicate with each other by passing messages to each other. Objects often represent real world entities, making an OO-program to essentially be a simulation of a part of reality. The OO paradigm has been around since the end of the 70s of this century (A major milestone was the Smalltalk-80 programming environment), and has gained impetus near the end of the 80s, with the introduction of a number of object oriented languages like Eiffel, C++, and, later, Java.

The power of the paradigm is that the design of information systems becomes scalable. It is possible to divide the world into a set of objects, each with its own interface (set of messages it can respond to) to the outside world. Designing and creating objects can then take place independently. This approach to systems design finds its culmination in *Component-Based Development* that sees information systems as created of *components* with a general purpose that can be developed completely independent of other components and be reused unchanged in many different contexts. One of the strong points of the OO paradigm is that the same paradigm can be used at different levels of abstraction. From global systems analysis to the implementation of small components, the expression language and concepts (classes, use cases methods, messages) are the same.

The *lingua franca* for O-O analysis and design currently is the *Unified Modelling Language* (UML, Booch et al, 1999). UML builds on earlier generation languages to model OO-systems, and combines a number of notation techniques to model the structure of systems, their dynamic behaviour as well as many other aspects of the system. UML can best be seen as a toolbox with instruments to describe virtually any aspect of an information system and its context.

 

In this handbook we take UML notations as the language for expressing the models of information systems and models of the knowledge for the knowledge intensive parts of these systems. A few notations, notably worksheets, stemming from CommonKADS, augment UML in order to raise the expression power of our framework. These worksheets are used to collect information about the organisation in a structured textual format.

CommonKADS itself does not strictly adhere to the object-oriented paradigm, although some of its concepts certainly resemble OO concepts. However, the CommonKADS knowledge model sees knowledge as a process and does not model these in a complete object oriented way. Many of the CommonKADS notations are already expressed in UML, but others use a specific notation. Especially the knowledge model uses notations that are not part of the UML. This situation makes it difficult to use CommonKADS techniques for knowledge engineering within a project using object oriented design of systems.

The goal of the current work is to provide a framework in which CommonKADS techniques can be expressed in an OO language, and that the modelling techniques stemming from CommonKADS and OO methods can be combined in a seamless way. SDF-II provides a general structure in which methodologies can be combined and a detailed elaboration of this structure for the case that CommonKADS is integrated in the OO-world. Note that the result is *not* a methodology for object oriented knowledge modelling (although it can be used in this way) but a general framework for combining knowledge intensive modelling with mainstream methodologies.

### *Document Overview*

This manual provides a detailed overview of the SDF-II methodology framework. The next chapter describes the main principles of SDF-II, including a general description of system development methods and the way they can be combined in a mixed environment. This chapter is followed by a chapter which in describes in detail the various models that the SDF-II framework uses in its instantiation where CommonKADS and UML-based methods are defined. Each model and the notations used for those model is described in detail using extensive examples.

Ways of looking at the *process* of developing a model of a knowledge intensive information system are described in the chapter on Project Management. This chapter shows how the modelling approach laid out in the chapter before can be put into action in an actual project. SDF-II does not prescribe a project management method, but in this chapter some examples of linking modelling activities to project management methods are given, including a number of templates for using SDF-II models under a risk driven project management method.

The book concludes with a description of a few items of the SDF-II library, a bridge between CommonKADS template knowledge models and the way these models are described in SDF.

## Principles Underlying the SDF-II Framework

### *Rationale*

There are now an overwhelming number of methods, tools, techniques and methodologies used in practical software engineering. Although there are some efforts towards unification, in practice we have to live with the fact that people use different methods. This poses the question how one deals with multiple methods.

Let's take the CommonKADS example. In principle, CommonKADS is a complete methodology, spanning the spectrum problem statement to coding and testing, and also defining its own project-management approach (Schreiber, *et al.*, 1999) . However, when an organization wants to apply CommonKADS in a project, it will seldom use it in its full glory. For example, many companies have their own software-project management standards, which they want to apply. The CommonKADS developers state that their methodology is "configurable", meaning that those parts needed for a certain project. This is a nice feature, but does not provide a complete solution to the problem. One also needs to define how the ingredients of the methods can be linked. For example, suppose a project management approach defines a "definition study" deliverable. In that case we need to identify which CommonKADS model elements together provide this product.

### *Approach*

The discussion above implies that we need to define "bridges" between methodologies (in our case: between CommonKADS and UML/OO) to be able to support joint usage. For constructing bridges it is convenient to view elements of methodologies at three levels:

**Project management level**

A methodology may provide a project-management approach, prescribing a life-cycle model (LCM) with certain activities ("risk analysis", "review", "plan", etc.) and products/deliverables ("definition study", "requirements document", "test report", …).

A PM approach is often standardized organization-wide. Example methodologies include PRINCE 2 and PERFORM.

**Model development**

This constitutes the "hart" of many methodologies. Here, the methodology indicates what steps and products need to be developed to build the software. The products are usually called "models". Examples are the OMT models (object model, dynamic model, functional model, design model), Yourdon's YSM models and the CommonKADS model set.

If one methodology defines both a project-management and a model-development approach, there is usually a direct match between the project-management products and the model-development products. However, it is often the case that we need to link the standard project-management approach with different model-development approaches. For example, we want to use the

 

CommonKADS model set (i.e. the model-development approach) with our company-specific project-management approach.

**Notations**

Finally, methodologies often prescribe a certain set of notation to be used. For example, YSM deploys entity-relationship diagrams, state diagrams, and data-flow diagrams. Traditionally, there has been a wide variety of notations used in software engineering; both tutors and students of software engineering will have experienced that to their own distress. The book by Connor (1992) gives a good indication of the spectrum of techniques, and the subtle ways in which these overlap.

The abundance of notations and the resulting jargon jungle has made it difficult to use multiple methodologies within a single project. In this light the UML effort is a useful development. UML (Booch, Rumbaugh & Jacobson, 1998) is a proposal for a set of standard nations that can be used for system analysis and design. Although one can criticize certain decision made with respect to the chosen set, the UML is certain a step in the right direction (and will be accepted gratefully by tutors).

Note that UML does not provide notations for all MD products. Is emphasis lies on system analysis, with also two notations for system design. Other information, such as contained in the CommonKADS worksheets for context modeling, falls for the largest part outside the scope of the UML. Therefore, some notations remain methodology-specific for the time being.

**Table 1:** Description of sample methodologies

| **Aspect/ Methodology** | **PM approach** | **MD approach** | **Notations used** |
|---|---|---|---|
| **OMT** | - | analysis model:<br>* object model<br>* dynamic model<br>* functional model<br>design model | class diagram<br>data-flow diagram<br>state diagram |
| **YSM** | - | enterprise essential model<br>system essential model<br>implementation models | ER diagram<br>abstract data types<br>data-flow diagram<br>state diagram |
| **CommonKADS** | risk-driven spiral | organization model<br>task model<br>agent model<br>knowledge model<br>communication model<br>design model | UML diagram (extended)<br>UML state diagram<br>UML activity diagram<br>UML use case diagram<br>inference structure<br>worksheets |
| **Catalysis** | evolutionary | business model<br>system boundary<br>component specification<br>internal specification | UML class diagram (extended)<br>UML use case diagram (extended'<br>UML sequence diagram<br>UML collaboration diagram |

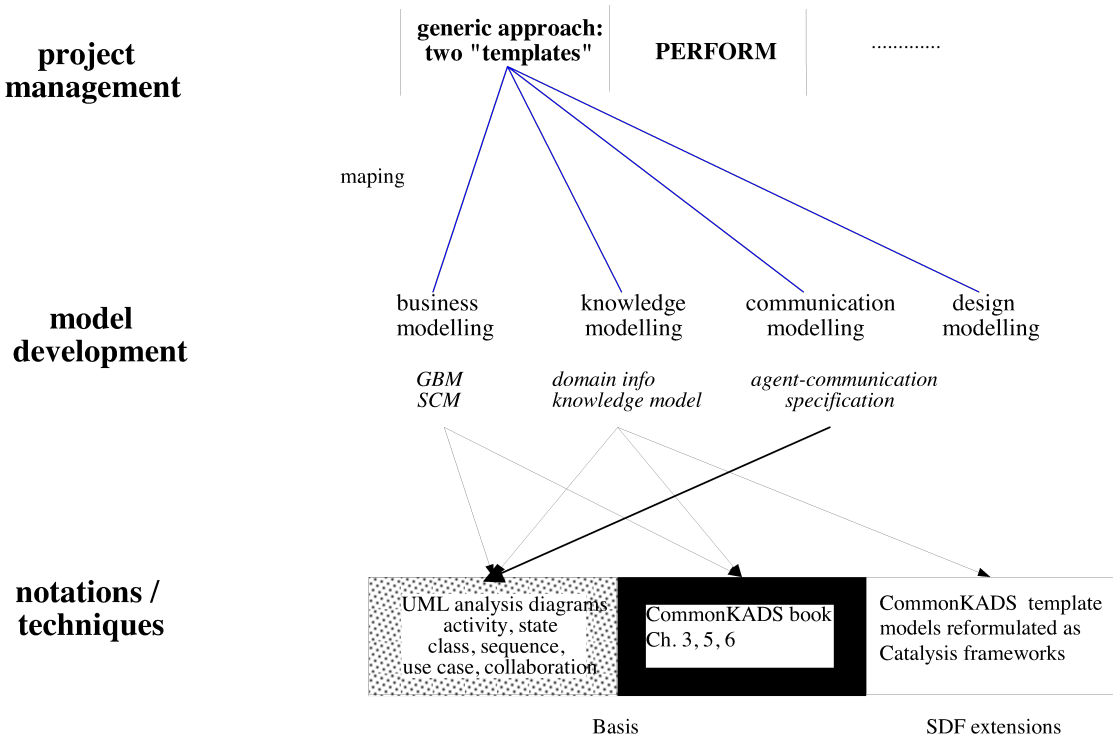


**Figure 2:** SDF framework overview

Methodologies differ in the way they cover these three aspects: project management, software development and notations. Table 1 shows a few sample methodologies using the distinctions made.

### *SDF-II Framework Overview*

Figure 2 shows these three levels as they appear in SDF-II. At the top we see the project-management level. There is a large diversity of approaches we could consider here; the names in the figure are just some examples. Also, these project-management approaches are often confidential. Therefore we have adopted in SDF-II a pragmatic approach. We describe in this handbook some typical project-management scenarios that should serve as insightful examples. The scenarios represent frequently occurring situations in developing knowledge-intensive applications. In the scenarios we indicate the bridges from the project management activities and products to the model-development level (i.e. instances of the lines connecting the two levels). We expect that companies will want to specify their own company-specific project-management strategy. The scenarios provided serve as guidelines for this process.

The middle level represents the *models* that are part of the description of the system. Models are the main products of the systems analysis and design activity. There are models that describe the context of an information system, models for the functions it needs to perform as well as models that show how the system is divided into components. Models relate to the project management levels, because they are (part of) the deliverables that are required by the project management. For instance if a project management method requires a functional specification, this document would contain a model providing these specifications.

At the bottom we see the notations used. Notations are used to represent models. In SDF we mainly use UML notation. For the inclusion of structured text used in the

 

description of business models, especially those describing the organisation structure we use some of the worksheets defined by CommonKADS. The choice for this notation language is justified by the fact that UML is rapidly evolving as the world's standard notation language for object-oriented modelling and design.

## SDF-II Framework Definition

The SDF-II framework assumes that the goal of the analysis of a knowledge intensive system is a *model* or better a set of models of the system that should be created and the context in which it will function. By creating a set of models, one chooses to divide the world into manageable parts, and modelling each part separately. SDF-II chooses to grossly follow the way CommonKADS has divided the world into models, like depicted in Figure 1. In SDF we distinguish the business model, consisting of the three models at the context level in Figure 1, as well as the knowledge model, communication model and design model.

The *business model* describes the context in which the knowledge intensive system must operate. What is the organisation like, what are its goals, which are the actors and what are the processes going on in the organisation? The goal of the business model is to position the system within the organisation and to describe the responsibilities the system has towards other actors in the organisation and vice-versa.
Quite often, we see that for the business model actually two models are created, an *as-is* and a *to-be* model. This recognises the fact that often information systems development is part of a business process redesign activity.

The *knowledge model* is very specific to SDF-II, as the presence of knowledge in a system forms the criterion for a system to be knowledge intensive. SDF-II allows to model the processes or actions that are knowledge intensive and to identify the knowledge needed to perform these processes. Also the *domains* in which the knowledge intensive actions are operating are modelled here.

The *communication model* models how the various actors in an organisation or within a system exchange information. The actors can be people inside the organisation, people outside the organisation (e.g. clients), other organisations (the organisation of a supplier) or software systems.

The *design model* specifies how eventually the models can be realised in a system. The design model serves as the bridge between the analysis of the system in terms of abstract concepts and concrete concepts related to programming and other technical issues.

Business model, knowledge model, and communication model together provide the system-analysis documentation. It should be stressed that in modern system analysis the focus is on conceptual, “application-world”, distinctions: the analysis models describe real-world objects, and not implementation objects. In knowledge engineering this is called the knowledge-level principle: knowledge should be analysed in implementation-independent terminology[1]. UML and Catalysis advocate the same principle.

[1] The rationale of this principle, formulated by Newell (1982) dates back to the earlier days of expert systems, when one tried to capture knowledge directly in the format of an implementation formalism, such as production rules.

 

The principle has some practical implications for the use of notational techniques. For example, if we talk about using a class diagram at some stage during system analysis, we mean that the analyst should use the class-diagram notation to model *real-world objects*. For this reason Catalysis introduces the term “type model” to denote a class diagram in which real world objects are being described; the term “class” is reserved for implementation objects. We do not adopt this terminology, as we consider class diagram to be primary an analysis notation (cf. the UML guide). By definition a class diagram that is used during analysis contains definitions of real-world objects. It should be added that the some of the class-diagram notation of UML looks suspiciously implementation-specific, e.g., the public, protected, or private nature of an attribute. We discourage the use of such detailed specifications at the analysis stage.

## *Business modelling*

### Modelling concepts

Business modeling is concerned with modeling business processes in which we are interested from an IT point of view. We can distinguish two types of business models, which are used for purposes:

**General business models**

General business models describe (a larger part of) an organization. The focus is not (yet) on one particular software system. The description of a general business model typically includes:

- organization structure: departments, branches
- organizational roles, people
- other “agents”, i.e. software systems
- process work-flows
- resources
- culture & power

General business models are often used for many different purposes in the organization. They tend to play in key role in knowledge management.

**System context models**

System context models describe the direct organizational environment of a software system. System context models are typically used in application development projects in order to indicate how the system should interact with its environment.

System context models typically have a smaller scope than general business models. Their content is more system-analysis oriented. System context models describe information and control flow between the system and its environment. An example of a system context model can be found in Figure 9 below.

### General business model

Each application project should specify a general business model, relating to the relevant part of the organization for the project. Ideally, this model is already present for a large part. And is maintained by the organization’s knowledge manager.

 

There can be two instances of the general business model: one for the current organization, and a second one for the new organization.

A general business model should at least contain the following elements:

- One copy of the CommonKADS Worksheet OM-1 which describes the problem/solution portfolio as well as some organizational invariants.
- A UML activity diagram, describing the major process work flows. Use swim lanes to indicate the place in the organization where the process takes place. Use objects only for the major relevant objects. Avoid diving into too much detail (you cannot be complete at this coarse level of description).

  In case of complex workflow, one can include a second hierarchical level of more complicated workflow in separate activity diagrams. Two levels should be the maximum, however.

Optional components of the general business model are:

- Worksheet OM-2 ("Organization focus area description"): extends the process model and enables a more complete organization description.
- Worksheet OM-3 ("Process breakdown"): characterize each process in terms of its significance and the fact whether it is considered knowledge-intensive.
- Worksheet OM-4 for describing knowledge assets needed in knowledge-intensive tasks at a general level.

### System context model

A system context model is required for every application. Note that this model is sometimes called the "business model". Its scope is, however, limited to the direct environment of the system.

The technique of choice for specifying a system context model is the UML use case diagram. You can use the extended notation proposed in Catalysis, in which actions and objects play a more prominent role.

Optionally, you can also include the following information:

- A UML state diagram (in case of asynchronous control) or a UML activity diagram (in case of synchronous control) of the interaction control between the external actors and the system.
- Include major object flows in the UML activity diagram.
- A UML class diagram, with all classes representing *types*, to describe the static structure of the information exchanged between the external actors and the system.
- Worksheet TM-2 for knowledge assets that the system should own.
- Worksheet AM-1 for the external actors in the use case diagram.

### Example: the elevator-design domain

**Problem statement.** A company specialises in the construction of elevators for buildings. The company is organised in a number of departments. The three main departments involved in this study are the sales department, the design department and the production department. The design department is suffering from a chronic

lack of adequately trained personnel. The sales department suffers from this, because they are dealing with customers that complain about the long periods required to make a tender. For such a tender the availability of a design is mandatory, because it provides the basis for the cost calculation. Interviews with the design department have revealed that about 90% of the elevator design is actually "standard stuff", meaning that the design is based on relatively simple variations on a standard elevator design. Therefore, the head of the design department has proposed to construct a software system, that should be able to propose an elevator design in such a standard situation. The human designers could then concentrate their efforts on the difficult 10% of non-standard designs.

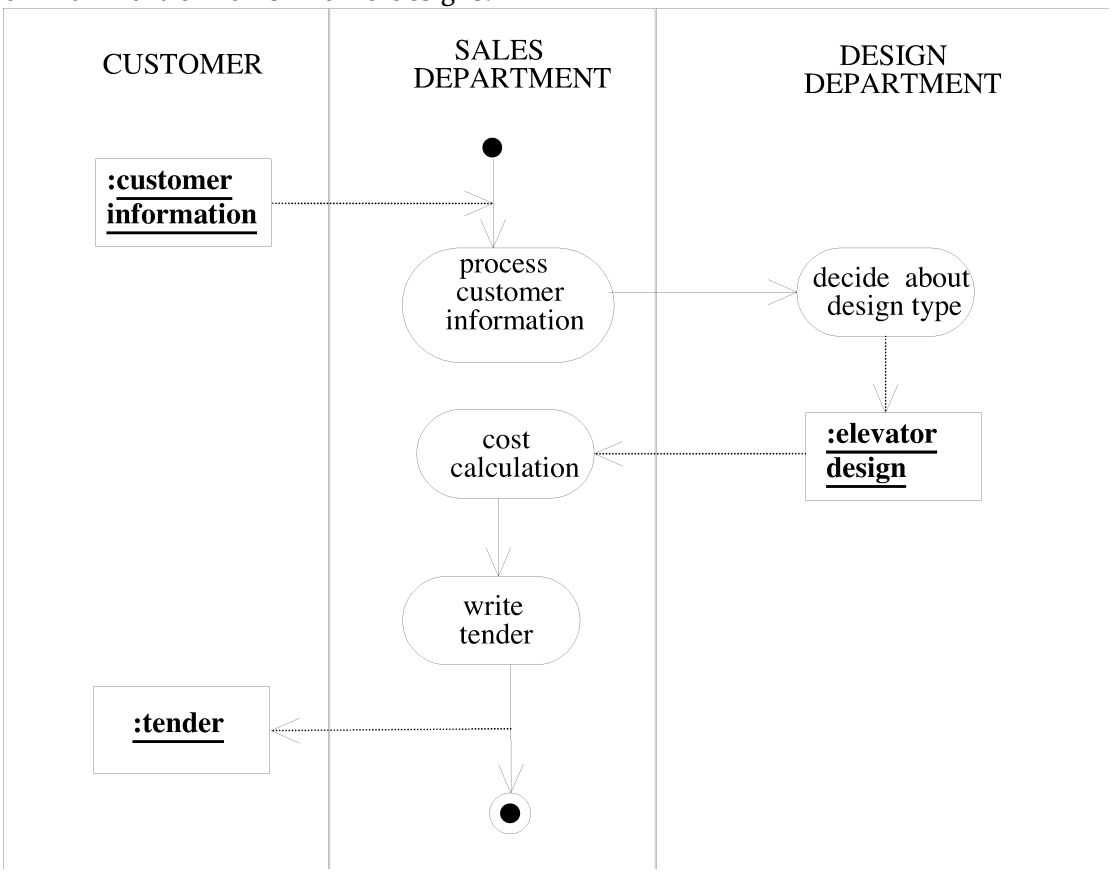


**Figure 3** Activity diagram of the current business process. This diagram is part of the General Business Model

Figure 3 and Figure 4 show, respectively, the current and the future business process. The diagrams are UML activity diagrams. Together with the worksheet OM-1 these form a minimal General Business Model. Note that the solution in the worksheet is reflected in the new organisations structure and process in Figure 4.

 

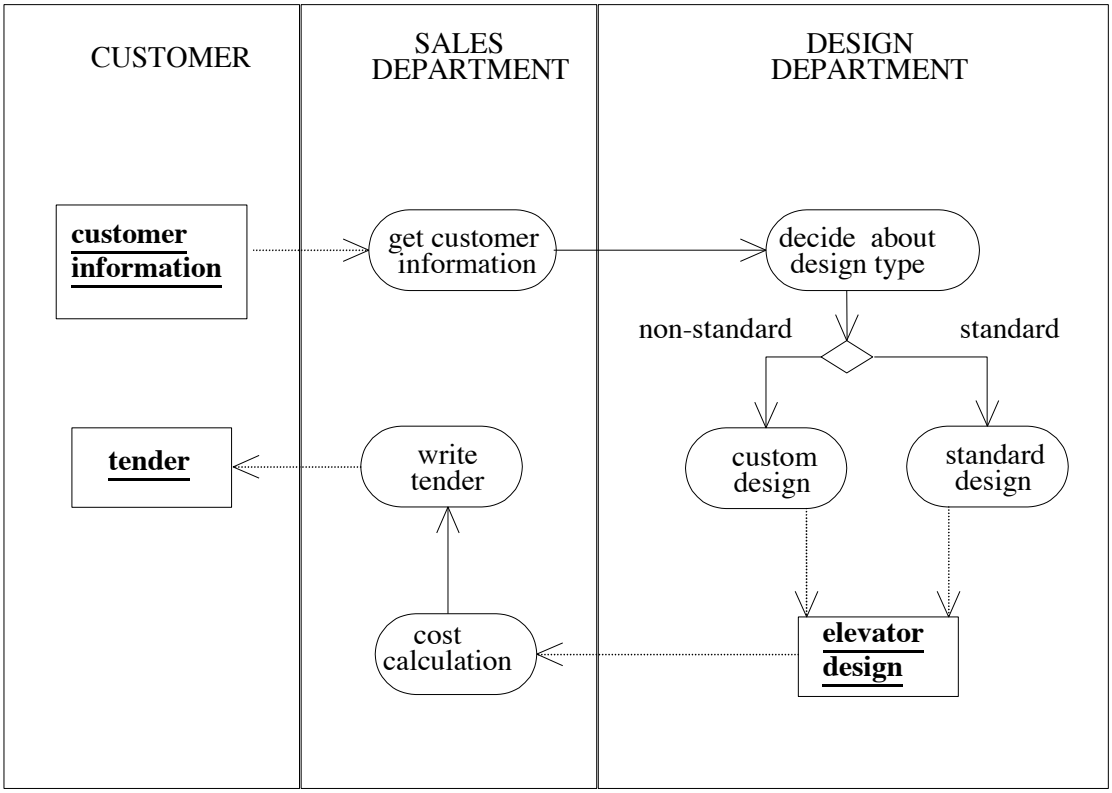


**Figure 4** Activity diagram of the new situation. This is also part of the general business model.

| **Organisation Model** | **Problems and Opportunities Worksheet OM-1** |
|---|---|
| *Problems and opportunities* | 1. making a tender after a customer request takes too long<br>2. there is some social friction between the design and sales department |
| *Organisational context* | 1. Mission: commercial company, image of good employer<br>2. External factors: Safety regulations w.r.t. elevators in buildings<br>3. Strategy: operate fast in competitive market |
| *Solutions* | Solution 1:<br>a. Construct software system for standard design tasks<br>b. Select and train liaison person (recruited from design department).<br>c. Reorganize design department for handling the non-standard designs. |

 

## *Knowledge modelling*

### Mapping CommonKADS concepts to UML

The knowledge model takes a central place in CommonKADS. This model describes the structure of knowledge intensive tasks as well as the knowledge needed for performing these tasks. The CommonKADS methodology takes a task-centred stance towards the modelling of knowledge. Tasks are decomposed into subtasks up to a level of elementary *inferences* that are not decomposed further. An inference specifies a step in a reasoning process in terms of the inputs of the step, the outputs and the knowledge needed for the step, as depicted in Figure 5. The basic assumption behind this way of modelling knowledge is that knowledge is a *process*, i.e. as something that can help to convert one type of information or knowledge into another type. A task is composed of a number of combined inferences yielding an *inference diagram*. The inputs and outputs of an inference are specified as *roles*. A role is a 'placeholder' for a domain element, or object that can play the role specified. For instance, in specifying a knowledge intensive system that assesses whether people are eligible for a certain loan, one can specify an inference diagram using a role of *applicant*. In a concrete system this role can eventually be mapped to a *person* registered in the database of the company giving the loan. This yields a three-layered view on knowledge in CommonKADS: a *task layer*, providing the task decomposition and the control over the task, an *inference layer*, providing the roles and inference diagram and a *domain layer* specifying concrete (domain) objects and knowledge bases. Mappings between these layers are the glue keeping together the complete knowledge model.
**Figure 6** displays the structure of the CommonKADS knowledge model using the three-layered architecture.

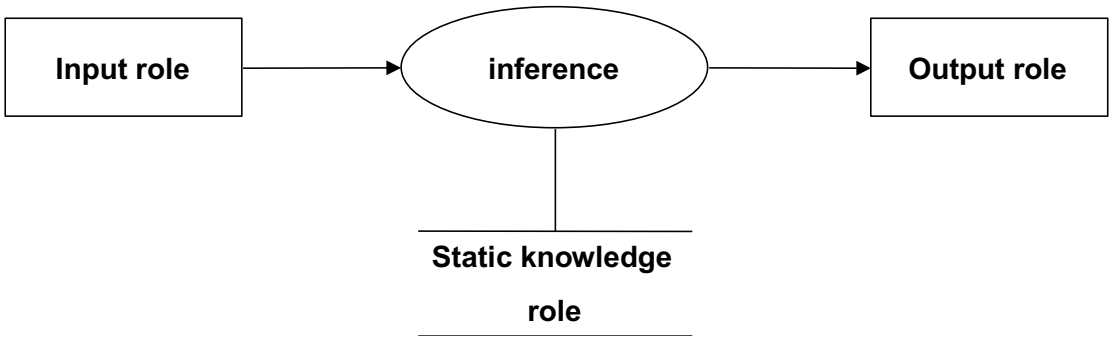


**Figure 5** A CommonKADS inference

 

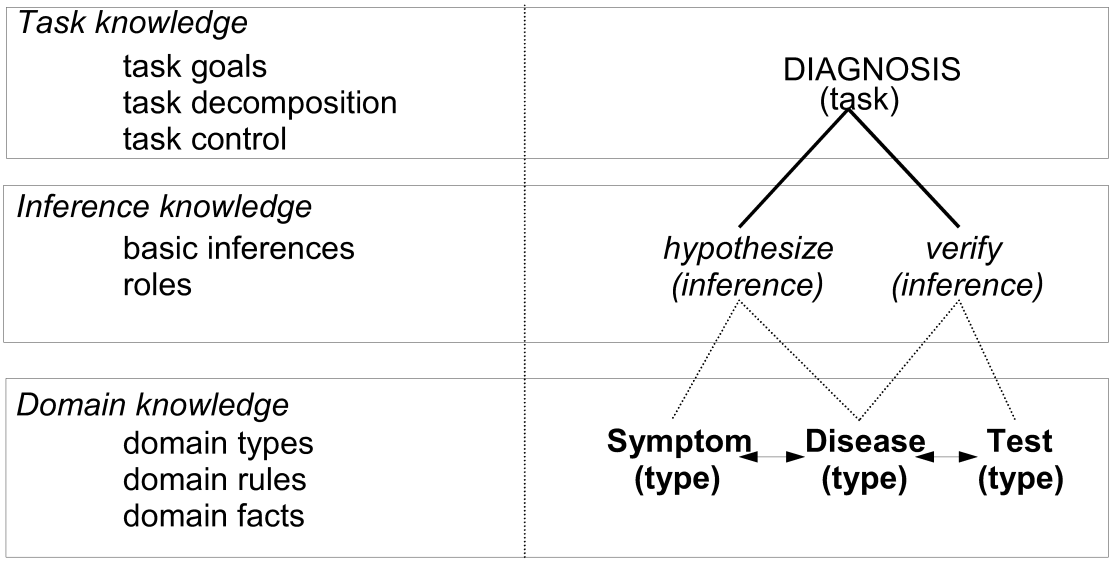


**Figure 6** The three layers of the CommonKADS knowledge model. From Schreiber et al. (1999, p. 79).

The ingredients of a knowledge model allow describing a task in great detail and to indicate the knowledge intensive parts of the task. The essential ingredients of a model are:

- *Tasks.* Tasks form the top level unit of analysis. A task can be, for instance, assessing an insurance application, diagnosing a patient, or making a roster for a school semester. Tasks can be decomposed into subtasks or into basic inferences. Upon decomposition, the *control* over the subtasks should be specified: which tasks to perform first, tasks to iterate etc.

- *Inferences.* An inference is a reasoning step that is seen as being elementary, i.e. that is not decomposed any further. Examples of inferences are abstracting an object, decomposing an object, generalising an object.
- *Dynamic roles.* Dynamic roles are placeholders for the objects in the domain that play a role in the reasoning process. For instance, insurance applications, candidate solutions for a problem and illnesses can be dynamic roles in a knowledge model. Dynamic roles are the things the model can reason *about*.
- *Static roles.* Static roles represent the units of knowledge that are used in the reasoning process. Examples are the set of rules that are used to assess a person's file, heuristics to find possible solutions of a problem, taxonomies to classify cases. Static roles are the things the model can reason *with*.
- *Rule types.* Rule types specify the structure of the domain knowledge contained in a static role. They constrain the types of expressions that can be part of the domain knowledge. For instance, for abstracting a case a rule can be

```
patient.temperature>37.5 abstracts-to patient.fever=true.
```

This rule follows a pattern:

 

```
<expression about patient>
        abstracts-to
<expression about patient>.
```

A rule type defines such a pattern and hence defines the knowledge that is present in a domain representation.

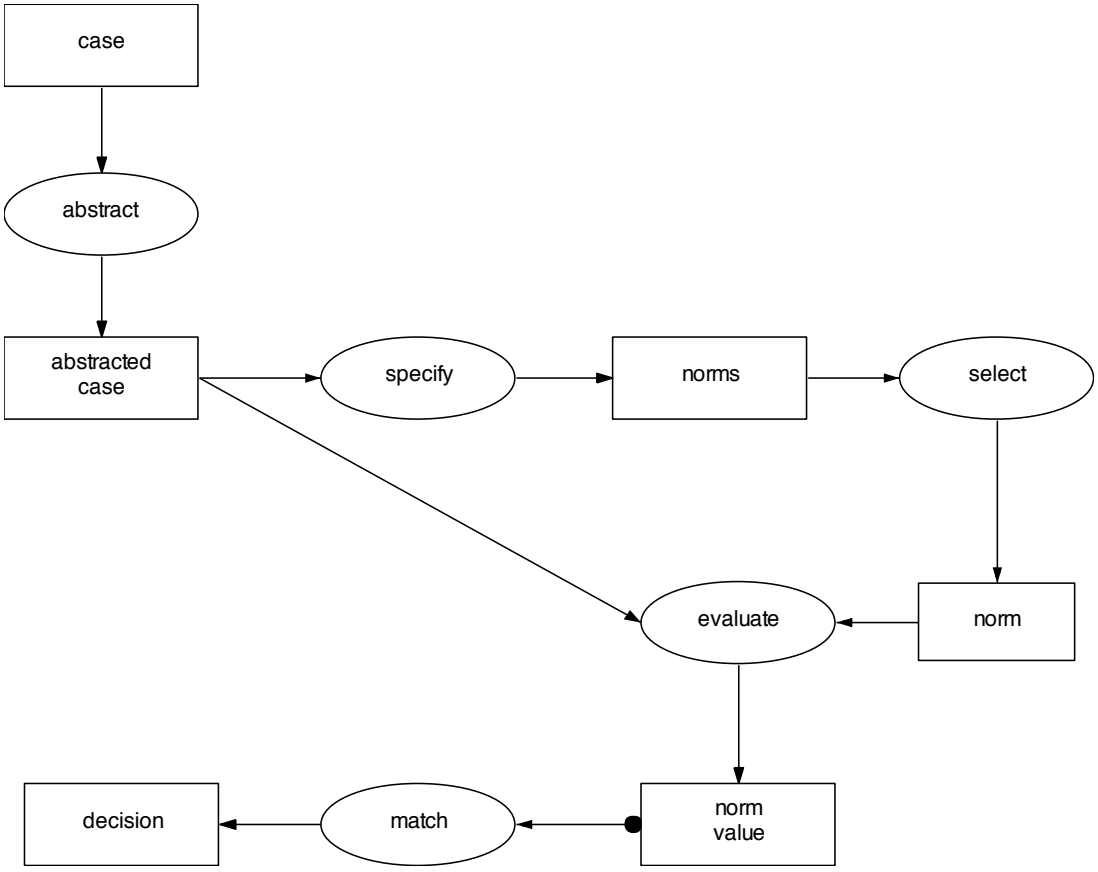


**Figure 7** An example inference structure for an assessment task. From Schreiber et al. (1999, p. 121).

Using these elements one can describe a complete model of a domain.

Figure 7 displays a complete inference structure for an assessment task. This inference structure is based on a task decomposition of the assessment task into an abstraction and a matching step, the latter to be further decomposed into specification of norms, selecting norms to evaluate and then match the evaluation to a decision. Inference models like this form the core of the CommonKADS knowledge-modelling framework. Therefore in order to build a bridge from CommonKADS to UML, the building blocks of the task level models and inference models need to be mapped onto UML concepts.

UML has some concepts that can be considered that are suitably matched to the CommonKADS concepts of Tasks, Inferences and Roles. The following table shows how such a mapping can be made. The two main concepts that are uses are use cases and types. A Use case is a description of a process that takes place within the context that is being described. For instance, a use case could represent the process of

 

ordering a product, accepting and insurance or whatever. A use case diagram consists of a representation of the use case and the *actors* in the use case. The use case is completed with a step by step description of what takes place within the use case.

Catalysis (D'Souza & Wills, 1999) offers an extension of the use case concept. In Catalysis use cases are described precisely using pre- and postconditions, describing respectively what the conditions are for a use case to take place and the result of the use case in terms of changes to the state of the actors. For instance, a use case for buying a product for a post order company may have as precondition that the person is registered as a client and as a postcondition that the person owns the product and that the price is paid to the company. Also in Catalysis, use cases may be decomposed. For instance buying a product can be decomposed into ordering, shipping and paying the product. Each part of the use case is considered to be a use case in its own right. Catalysis often terms use cases as being *actions*.

*Types* are extensions of the class concept in UML. A type represents a set of related responsibilities, represented as operations on the type. Where classes also include attributes, types have related types that are part of their definition. A specific class, on the design level can realise the behaviour represented in a type. For instance in the case of a person buying a product, a specific class can realise the type of client. Classes can implement multiple types just like concepts in CommonKADS can play multiple roles.

The mapping between CommonKADS concepts and UML/Catalysis entities takes place according to Table 2.

**Table 2** Rules for mapping CommonKADS concepts to UML elements.

| **CommonKADS concept** | **UML concept** | Mapping |
|---|---|---|
| Task | Use case/Action | Actions indicate that something is happening with the involved actors, represented as types. |
| Inference | Use case/Action | An inference is seen as a singular action. |
| Dynamic role<br>Static role<br>Rule type | Type<br>Type<br>Type | Types represent all the objects that play a role in the model. The specification of the type defines the kind of role the object plays. |

This table provides a way to adapt a knowledge model in CommonKADS into UML. The mapping is relatively straightforward. As an example Figure 8 depicts the UNM version of the inference structure in Figure 7. Some differences draw attention here. First, some of the roles in Figure 7 have been combined. For instance case and abstracted case have been combined into one type, the same is true for norms, norm and norm value. In fact in the CommonKADS diagrams the combined roles actually represent the same things, abstracted case and case both refer to the same case, only the abstracted case contains a number of extra abstracted data. Norms is actually a collection of Norm, and Norm value is an attribute of Norm, so these three roles refer to the same concept as well. The inferences are the same, however, the interpretation

 

of the diagram is somewhat different in Figure 7 than in Figure 8. In Figure 7 an inference means that a *process* takes a role as input and generates another role as output. In Figure 8, an action means that the objects participating in the action together make sure that a certain end state of all objects is reached. That can be realised by a separate process, or by the objects themselves, as in object oriented environments objects themselves can include behaviour. Both specifications are in itself independent of the actual implementation.

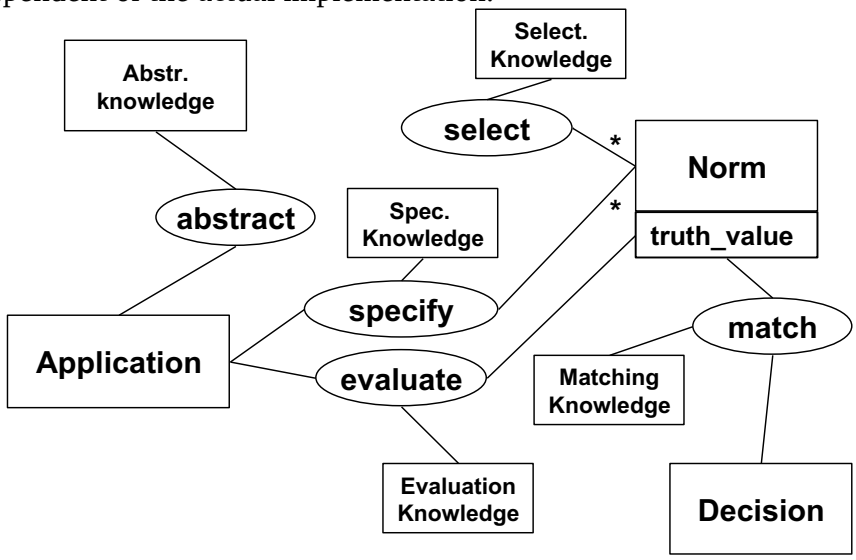


**Figure 8** UML version of the inference structure represented in Figure 7

The actions depicted in Figure 8 do not show inputs and outputs, as process representations would. Actions actually do not have the notion of in- and output. However, they can have a notion of initiative. For instance, the application may take the initiative for the abstraction. Initiative can be indicated in collaboration diagrams by drawing an arrow from the initiator to the action, instead of a line. This is not necessary and should only be done when this increases understanding of the diagram. Of course, with just replacing one notational convention by the other, the integration is hardly done. Figure 8 only is an end result of a knowledge modelling *process*. The process leading to diagrams like Figure 8 will be discussed in the next section. Another point is the specification of the *content* of the action. What exactly does "specify" mean in Figure 8. Catalysis specifies the content of actions by *pre-* and *postconditions*, expressions that must be true before and after the action has taken place. In the next section we will also address the way the content of the action is specified in SDF-II and the role of knowledge herein.

## Creating a knowledge model using refinements

A knowledge model in SDF-II will be represented by a set of *types* (representing objects in the real world, including objects representing knowledge) and *actions*, representing joint responsibilities of objects to perform a certain task, reach a certain state or to actuate a process. This section deals with the question how business processes can be decomposed into smaller processes and actions, into the level of detail needed for system specification. Especially the difference in approach between knowledge intensive and not-knowledge intensive parts will be discussed.

 

As an example we will take an insurance company. A person who wants to insure a certain object or take a life insurance will apply for this at the company, the company will assess the application and accept or reject the application. In both cases the person will receive a notification and in case of acceptance, the application will be processed into an insurance policy.

The system context diagram for such a process is displayed in Figure 9. This diagram is not a part of the knowledge model - it belongs to the business model – but it is the starting point of knowledge intensive analysis. The “apply” action is a *business process* that is taking place, being a joint responsibility of both the client (filling in the application form, answering questions) and the company (processing, assessing and finalizing the application). The post condition indicates that the end result of the action is that the client is notified of the result, and that the result is a consequent of the business rules that the company uses in assessing this kind of applications. The “application” type is part of the glossary that defines the central terms in the description and the conditions for the action. For reasons of space, not all terms are depicted here. Especially the term “business rules” may be a candidate for including in the glossary.

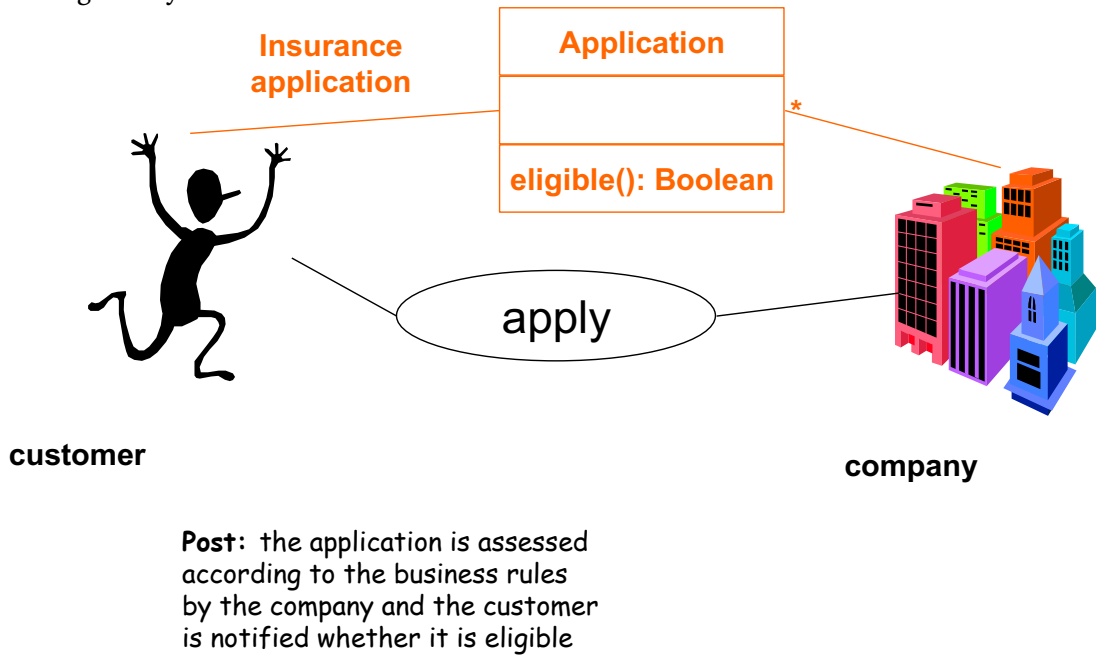


**Figure 9** System context diagram for an insurance application, including a postcondition.

The system context model provides a top-level description of the conditions in which the new system must operate: who are the actors (human, organisational and technical) in the system’s environment an in which way does the system co-operate with these actors. The system context model can be elaborated a bit more like shown in Figure 10. This figure makes use of a convention used in Catalysis (D’Souza & Wills, 1999) on using the attributes box pf a class or type to denote the glossary of related types. For types it is seen as not useful to indicate attributes, because they are seen as specific for a certain implementation. A *glossary* instead defines the language for defining the types used to express ideas, rules or constraints on the types. Here we see that inside the insurance company, the same application as in the glossary in

 

Figure 9. Also two actors have been added: an assessment *system* and a knowledge manager. The task of the latter is to keep the knowledge used by the assessment system up to date, for instance with changing legislation or changing company policy. This role was introduced here in order to indicate a requirement on the nature of the knowledge representation of the assessment system. The knowledge should be represented in such a way that this kind of user can access and change the knowledge inside the system. This usually is a main reason for representing knowledge explicitly, rather than putting the knowledge directly into the code of the system.

In our system we would label the *assess* action to be knowledge intensive, meaning that this operation requires explicit representation of the knowledge which would be acquired and represented using techniques of CommonKADS. This would mean that the actions used to decompose such a knowledge intensive action would have the interpretation of being a CommonKADS inference. This means that they all will incorporate an explicit knowledge role in their specification.

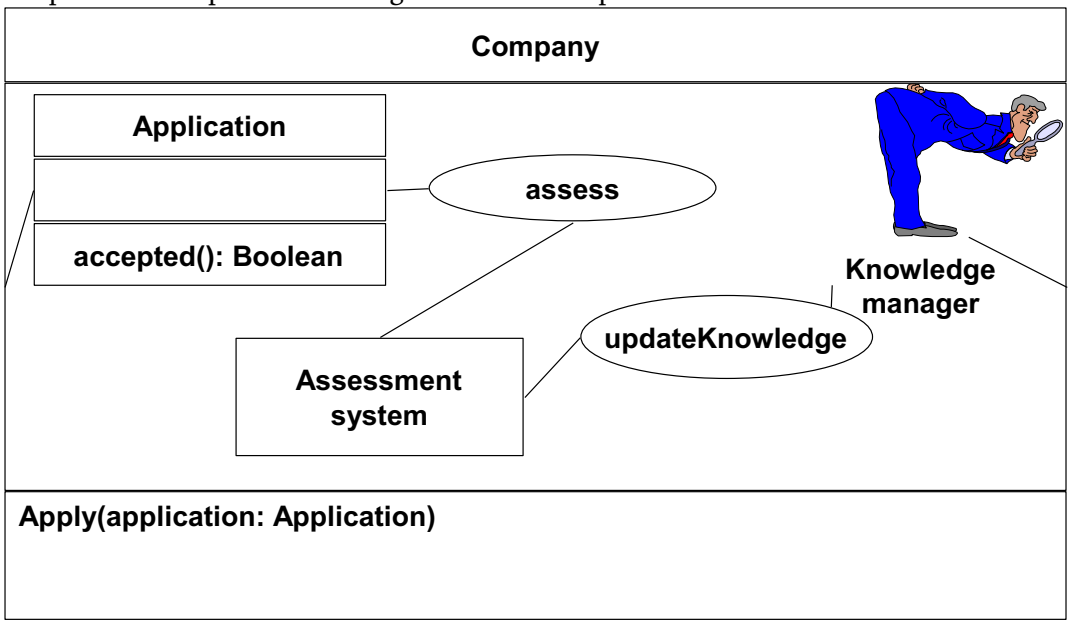


**Figure 10** Class (or type) diagram for the company, itself viewed as a type.

The assessment system can be further decomposed as in Figure 11. Here actually two steps are done in one. First, following Figure 10, the assess *action* is converted to an assess *operation* on the assessment system. This step means that the responsibility for taking the action is fully assigned to the assessment system. The second step is that the assess operation is implemented by a collaboration between the application, the decision to be made and the knowledge inside the company. The result of these two steps is that the assess operation is now an action inside the assessment system, that it is clear that it is knowledge intensive (due to the explicit presence of a knowledge role). The typical post condition for a knowledge intensive action is that the knowledge is applied to yield a result in one of the other actors.
It would be possible to leave out the explicit role of a knowledge type and express the knowledge itself in terms of the post conditions, for instance using a language like OCL. The advantage would be that the knowledge would be directly visible on a diagram like the one drawn in Figure 11. Major drawbacks, however, are the fact that each time the knowledge changes, the model using this knowledge should also change and that knowledge would be spread over the model and not be expressed in

 

a structured, decomposable form. Therefore, in SDF-II we choose for the presence of explicit types for knowledge, each representing parts of the knowledge needed for performing a specific action. Post conditions express that the knowledge attached to the action is applied in a proper way.

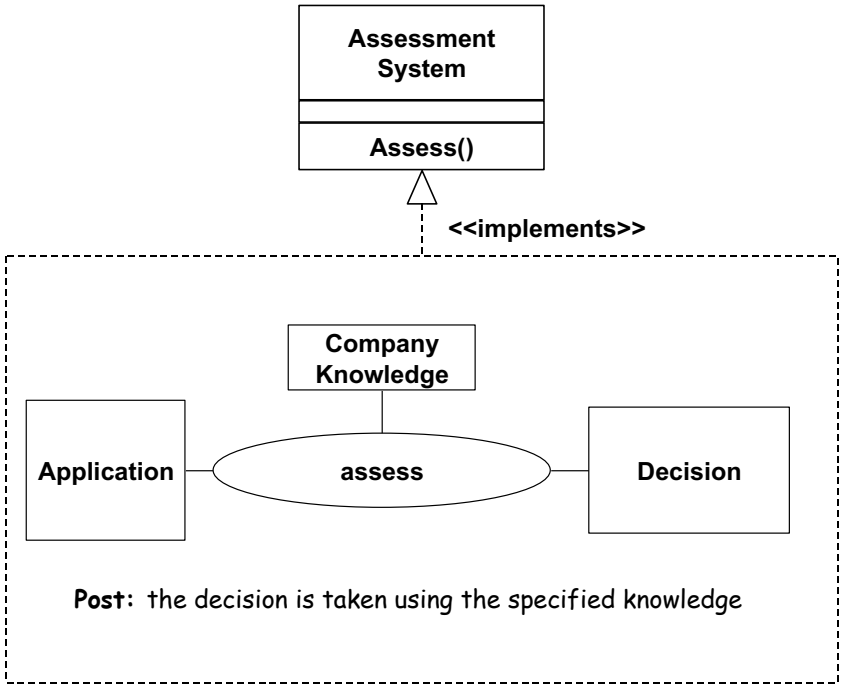


**Figure 11** The assessment action implementing the assessment responsibility of the assessment system, explicitly making use of knowledge.

After identifying knowledge intensive actions, the following step is their decomposition in sub-actions (subtasks) and basic inferences. Along with the actions, also the knowledge needed to perform them should be decomposed. The object of type 'Company Knowledge' depicted in Figure 11 represents the whole of knowledge that is needed in the system to complete the assessment operation. When this operation is decomposed, the knowledge should also be decomposed in order to indicate which parts of the knowledge are needed for which parts of the action.

In SDF-II we establish decomposition of actions and knowledge using *refinements*. A refinement is a relation between two models in UML which states that both models have the same goal and model the same system but that the model that refines offers information at a more detailed (or less abstract) level. A basic requirement on a refinement relation between two models is the existing of a *refinement model*, which specifies the relation between the concepts in the two models.

The bottom part (the collaboration) in Figure 11 can be further refined as in Figure 12. For the decomposition of the assess action we made use of the assessment template model from CommonKADS, as depicted in Figure 7. The refinement model displays the exact decomposition of the assess operation and hence explains the presence of actions like "abstract", "select" etc. in the refined model.

A similar decomposition should take place for the knowledge that takes part in the interaction. In detail company knowledge would decompose into the 5 knowledge types as visible in Figure 12. Following Figure 12, all inference actions are considered

 

being knowledge intensive. The original CommonKADS inference model did not see *select* as being a knowledge intensive task. The presence of a selection knowledge type indicates that here this inference is knowledge intensive. Of course, the decision to make a certain action knowledge intensive is domain dependent. If in this system selection would take place randomly, “select” would not be knowledge intensive and the selection knowledge would not have been present in Figure 12.

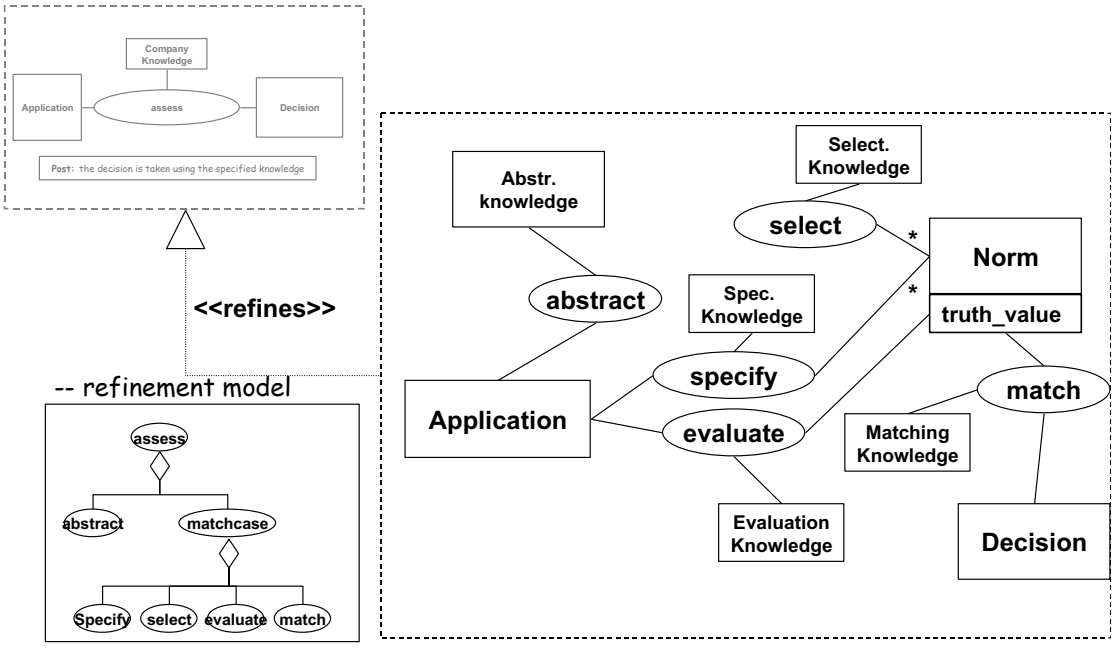


**Figure 12** Refinement of the assess operation. The small model top-left is the original model from Figure 11, the model bottom left is the refinement model, and the model on the right is the refined model.

Figure 12 shows *which* actions will take place. It does not specify the control over these actions, which will be discussed below, and not *what* the actions are. The what aspect of the actions is defined in terms of conditions and invariants, defining in declarative terms the effect of an action taking place. These conditions are elaborated in Table 3. In the case of modelling knowledge intensive actions, the post conditions always explicitly refer to the role of the knowledge involved in the action.

Figure 12 plays the role of the CommonKADS inference diagram. It shows, in terms of actions, how the various parts of the knowledge and the domain elements co-operate in order to yield a certain result. The interpretation of the collaboration depicted here is more of an object-oriented kind. In the end, all actions will be decomposed in terms of responsibilities for the domain objects present. There are three ways of doing this:

1. Assigning a responsibility directly to one of the participants of the action. This means that there will be an operation defined on this participant which will perform the action. Probably, the other participants will be passed as an argument into this operation.
2. Create a new *agent* object, which will provide the service of performing the action. This agent will not represent a concrete domain object, but will represent the task or inference that the action stands for. This option may be useful whenever it is difficult to assign the responsibility to one participant

 

Leave the assignment of the responsibility unspecified at a level and divide it further in new levels of refinement, down to the concrete design level of the system. For instance, abstracting an application is a co-operation of the application itself (providing information, storing the newly generated abstracted information) and the knowledge (generating new information by applying rules). The exact division of responsibility may be to detailed to specify at the level of system's analysis and knowledge modelling. In the end, at the design level, the decision has to be made.

**Table 3** Conditions specifying the actions in Figure 12

| |
|---|
| *Abstract* |
| **Post:** Application is in its abstracted state. Abstracted facts have been added generated by the abstraction knowledge. |
| *Specify* |
| **Pre**: Application is in its abstracted state. |
| **Post**: A set of relevant norms has been generated using Application information and the relevant specification knowledge |
| *Select* |
| **Pre:** Norms are available. |
| **Post:** One norm is selected as the current norm to evaluate, using the available selection knowledge. |
| *Evaluate* |
| **Pre:** One norm is current |
| **Post:** The value of the current norm is specified according to the evaluation knowledge. |
| *Match* |
| **Pre:** At least one norm has a truth value. |
| **Post:** If the known values of the norms lead to a decision, according to the matching knowledge, the decision is generated. |

The exact choice for method is dependent of the characteristics of the domain. As a general guideline one can say that it is wise to specify things only when you have to and when you are sure that the choice can be made correctly. The third option, leaving things unspecified therefore should be the default consideration.

An important issue in interpreting diagrams like Figure 12 is that of *control*. It should be stressed that, just like an inference diagram in CommonKADS, there is no implicit or explicit specification of control, i.e. the diagram does not specify the order in which the actions take place. UML offers several techniques for doing this. *Sequence diagrams* show timelines of the action sequence to display scenarios of actions. *Activity diagrams* can define a full control structure.

 

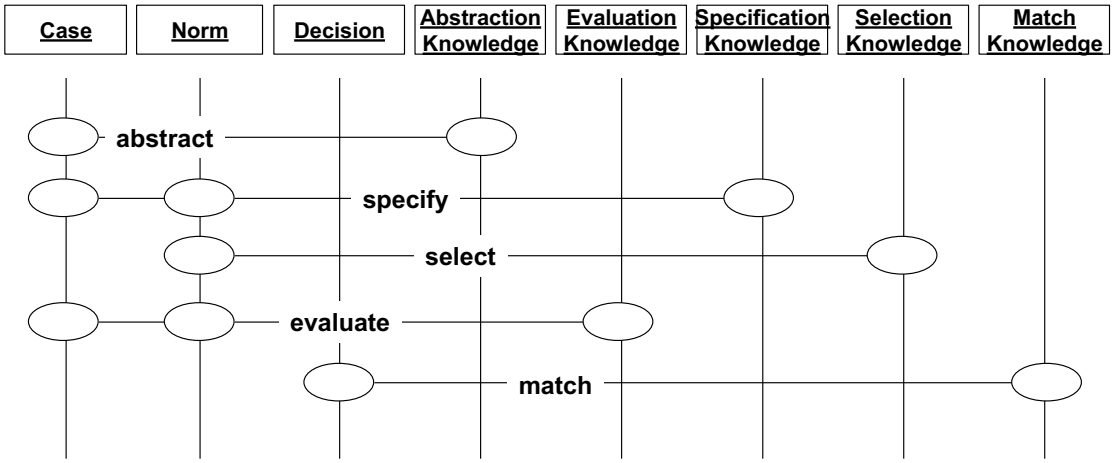


**Figure 13** Sequence diagram representing a data driven scenario for the actions represented in Figure 12

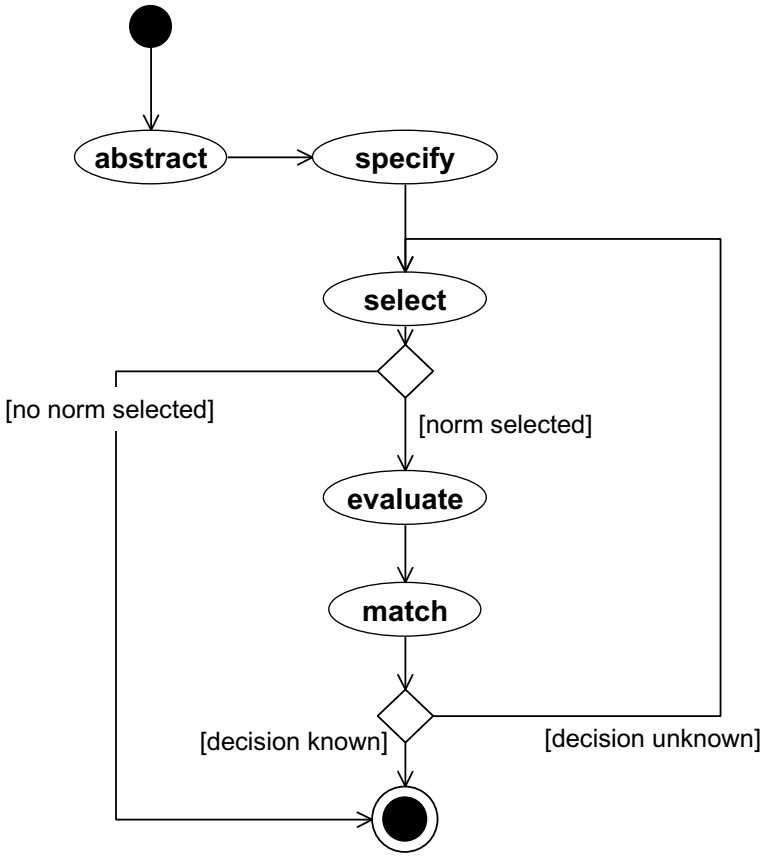


**Figure 14** Activity diagram defining a control structure for data driven inferences using the actions in Figure 12

Figure 13 and Figure 14 both define a specific control structure on the actions defined in Figure 12. The two kinds of diagram differ in terms of what they specify and what they show. The sequence diagram in Figure 13 shows a typical data driven scenario that can be applied to the actions. The vertical lines represent instances of the types present in Figure 12. Time runs from the top to the bottom and the actions, represented here as horizontal lines with blobs on the lines of the instances participating in the action. This particular notation for sequence diagrams was

 

adopted from Catalysis (D'Souza & Wills, 1999). The sequence is data driven because it follows a typical input-output pattern: the availability of case data triggers the first action, followed by other actions in a more or less logical sequence indicated by the flow of objects as visible in Figure 7. The activity diagram, Figure 14, (adapted from Vorgers, 1999) shows a structure for data driven inference. The main difference is that this figure does not represent a single scenario, but a control structure over all actions present in the diagram, including iteration.

The advantage of a sequence diagram is that it explicitly and naturally shows the sequence of actions in time as well as the participants in those actions. It can be very convenient to elaborate on several sequences in order to get a feeling and explain how the system will operate in practice. The activity diagram is more suited when the control is more complex and when a more formal definition of the control structure is needed.

Data driven inference is one way of using Figure 12. In this case, abstraction is the first action, meaning that this operation must generate all abstractions that *may* be needed in the rest of the process. For reasons of efficiency or elegance, however, it may be better to generate only those abstractions that are actually needed. In that case, the abstraction should take place *after* norms have been specified, as the specific norms drive the kind of abstraction that is needed. In this case an activity diagram would look as depicted in Figure 15 (adapted from Vorgers, 1999). This strategy is called a *goal driven strategy* because the abstraction only takes place to serve a specific goal, specified by the results of earlier actions.

The presence of two different inference strategies on the same collaboration diagrams shows that the control really is independent of the actions themselves. This lead to the fact that diagrams like Figure 12 can be quite generic and applicable with multiple problem-solving strategies.

 

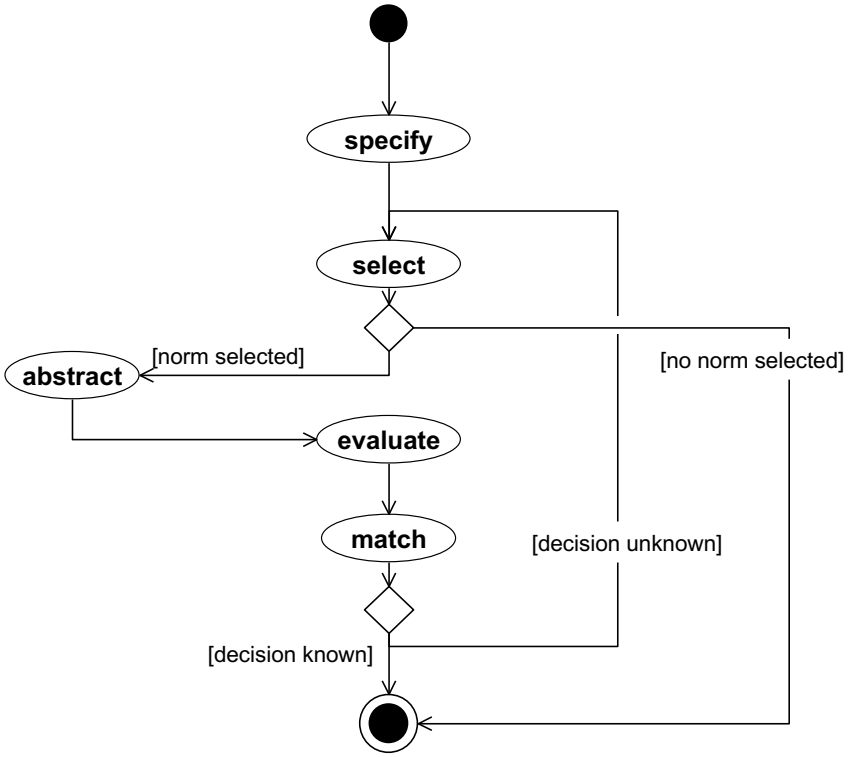


**Figure 15** Goal driven problem-solving strategy for the assessment example.

## Representing knowledge types

A central new element in the method for modelling knowledge intensive tasks and processes is the knowledge type. In the diagrams they appear as “normal” types, with a special role. For clarity, users may create a special stereotype for knowledge type to indicate their common role in knowledge intensive systems. The knowledge type represents the knowledge that is involved in performing an action. This means that a knowledge type gets the responsibility for applying the knowledge within the performance of the action. Concretely, this means that the knowledge type can have responsibilities of publishing the knowledge it contains and to apply the knowledge to a given instance of the information it refers to. Knowledge therefore can be modelled as a relation, or association class between the domain types that the knowledge acts upon. Such a model places the knowledge type itself in the domain model in a similar way as done by the CommonKADS *typical domain schema*. The knowledge type specifies the kind of relation that exists between the two domain types and represents in the implementation the actual container of the knowledge. Figure 16 and Figure 17 show a general definition of a knowledge type and an application for a part of the assessment domain model.

 

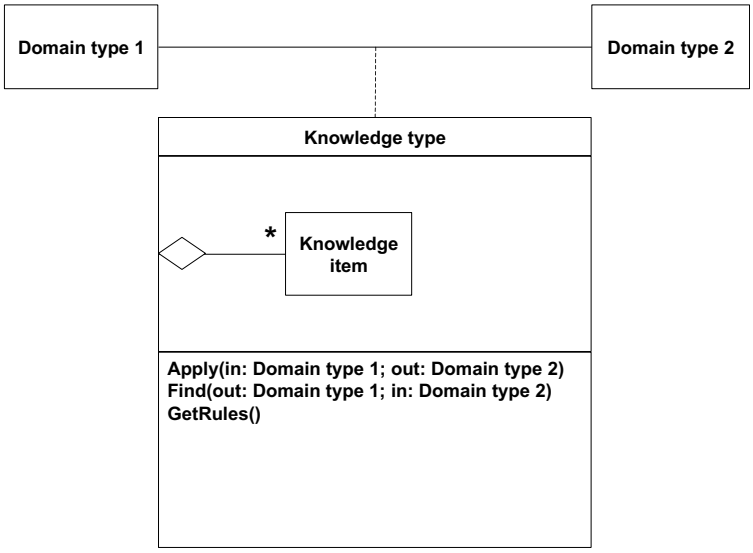


**Figure 16** Scheme for knowledge types, expressing the relation between two domain types in terms of knowledge.

The knowledge types are represented as collections of knowledge items. In the evaluation example in Figure 17 this is instantiated as *rules*. Rules can serve two goals in creating knowledge intensive systems. At the modelling level rules may be used to *specify* the knowledge needed for a certain process. As such these rules are considered to be part of the knowledge model. At the implementation level, rules may actually be the implementation of the knowledge. For instance Aion, a tool produced by Platinum Technology provides a direct implementation of a rule engine that evaluates rules as part of the reasoning process. In this part of the manual we are talking about rules a *modelling* constructs, decoupled from the actual way these rules are realised. The rules specified at the modelling level can still be implemented as C code, Aion Rules, Prolog clauses, or any other technique that seems suitable in a specific project.

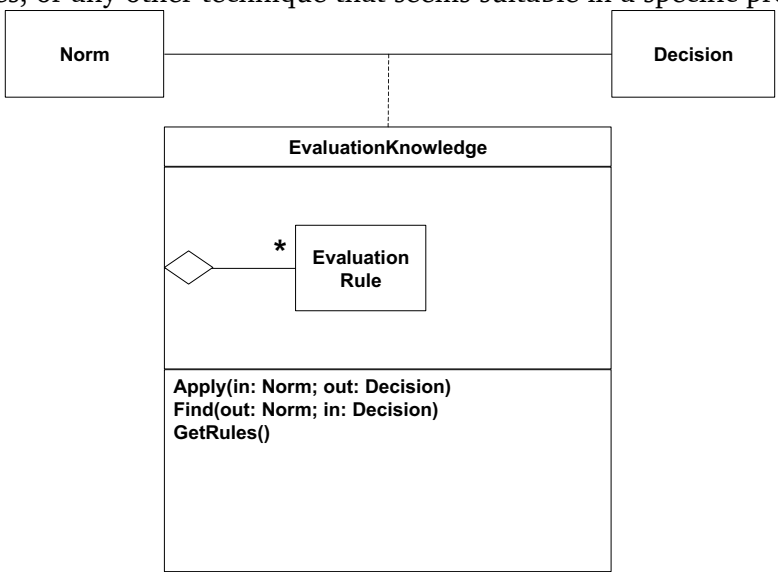


**Figure 17** Instantiation of a knowledge type for the relation between Norm and Decision in the assessment example.

 

**Table 4** Age categories for the insurance example

| Age | AgeCategory |
|---|---|
| <18 | Minor |
| ≥18, <23 | Young |
| ≥23, <60 | Middle |
| ≥60 | Old |

CommonKADS defines exactly what a rule is about. A rule is a certain *relation* that holds between *expressions*. For instance, when an insurance company uses specific age categories, abstraction rules can be used to put people into the various age categories. Assume that Table 4 contains the definitions of the age categories used by the company then the rules would take a form like:

```
Age < 18 abstracts-to AgeCategory = Minor
```

This means that this rule is a relation between the *expressions* "Age < 18" and "AgeCategory = Minor". The rule type EvaluationRule, typically is expressed as an *association* between two *types*, viz. Application and Application (in this case the abstracted knowledge is stored inside the same type. Using an UML template for rules can bridge this difference.
Figure 18 shows how this is done. The left part of this figure shows the definition of an expression as consisting of an operator, an object and a value (for instance >, Age, and 18). Object is a parameter that can be filled in by any object. This is done in the right part, where the expressions are used in the conditions and actions of a rule. The rule itself is a template for specific usage in knowledge types. For instance Figure 19 uses the rule template in defining the nature of the evaluation knowledge as shown in Figure 17.

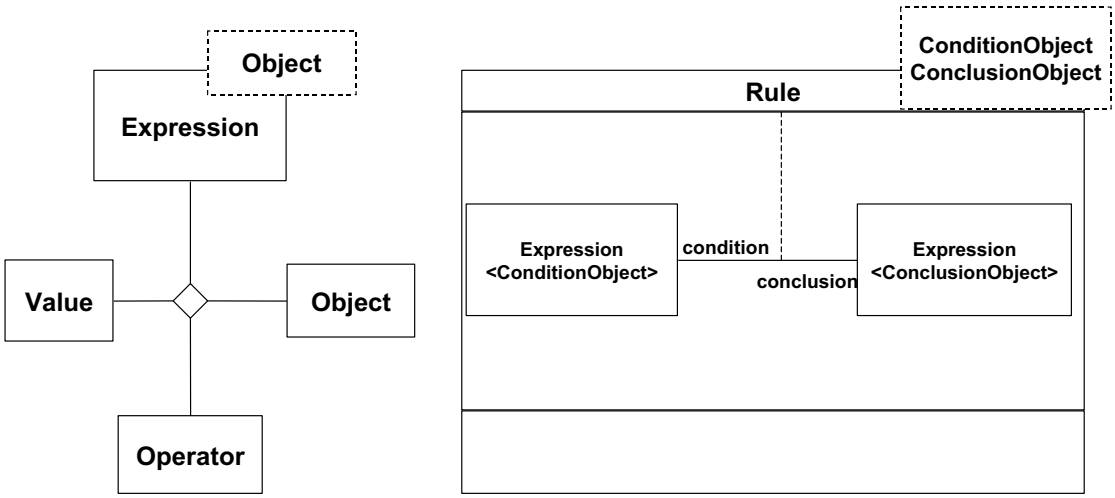


**Figure 18** Using UML templates for expressing rules

 

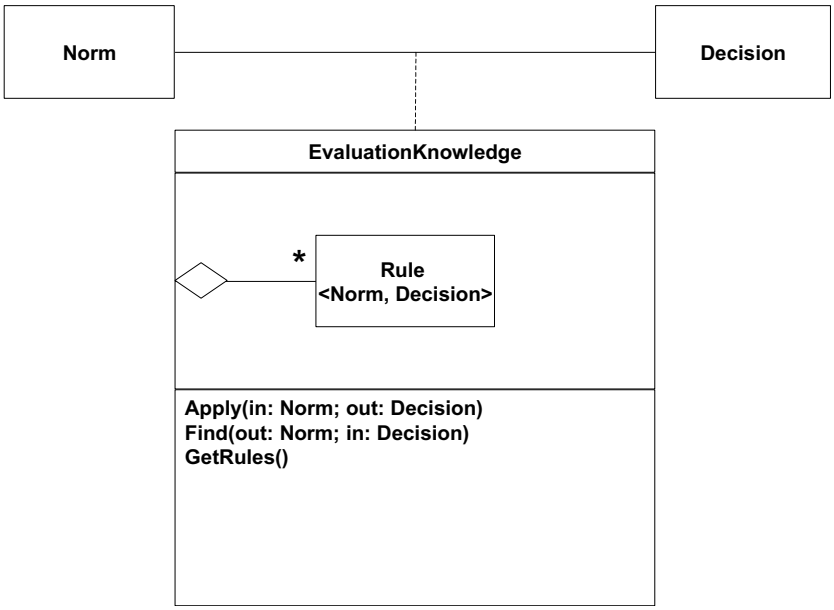


**Figure 19** The rule template used to define the nature of evaluation knowledge.

It should be noted that this exercise of defining rules should take place only once. In principle, Figure 18 can now be reused in any project to state that a certain type of rules exist that describe the relation between two domain types. By instantiating the template as done in Figure 19, one defines the nature of this relation. What has not been specified in Figure 18 is the *connection symbol*, the abstracts-to symbol in the example rule on page 28. This is usually irrelevant for the definition of the rule itself, mostly this term is implicit in the kind of rule and it is up to the modeller to choose a term when actually describing the rules.
Modelling knowledge in this way means that separate from a diagram like Figure 19, the model should also contain the actual rules instantiating the relation. For instance, for abstracting *Age* to *AgeCategory*, one would need to add Table 4, or a set of rules expressing the same, to the documentation of the model. This immediately shows one of the strong points of the approach to knowledge modelling outlined here. There is a clear separation between the *form* of the knowledge, expressed in the knowledge type and its *content* expressed in actual rules or tables. This also shows how the tasks of the knowledge manager from Figure 10 would fit in. He would have a task in updating the content of the knowledge types, not of the structure.
It should be noted that Figure 18 only defines rule types for rules containing singular expressions, like age >18 or income = 20000. More complex, like age>18 AND income = 2000 expressions require a more complex structure, which is easy to create but would go to much into technical detail for the scope of this manual.

This closes the discussion on the basic principles of linking CommonKADS concepts to UML. Basically what is offered is a mapping between notations and an interpretation, as well as a standard approach to knowledge modelling using knowledge types. The next section will explain how another important asset from the CommonKADS methodology can be incorporated into SDF: the use of template knowledge models.

## Linking type models to domains

The models described above are all *type models* meaning that they are expressed in abstract types that represent a role that a certain concrete domain object can play.

 

The idea behind this is that it is probably easier to find reusable parts of software designs when the design is abstracted from the concrete application. Of course, in the end the type models have to be linked to domain concepts. In this section it will be describe how this is done.

The domain concepts can be represented in a UML class diagram. For instance Figure 20 depicts a part of the class diagram for a company's administrative system. This concrete class diagram, representing the domain concept the company works with, should be linked to the abstract descriptions in the type models. This is done by assigning the roles to domain concepts. Now it should be noted that types represent *interfaces*. By saying that a certain domain concepts plays the role of a certain type, this means that the domain object takes the responsibility of implementing the type's interface. The notation used for this is that of *inheritance*, as inheritance means that the child object takes on the responsibilities of its parent.

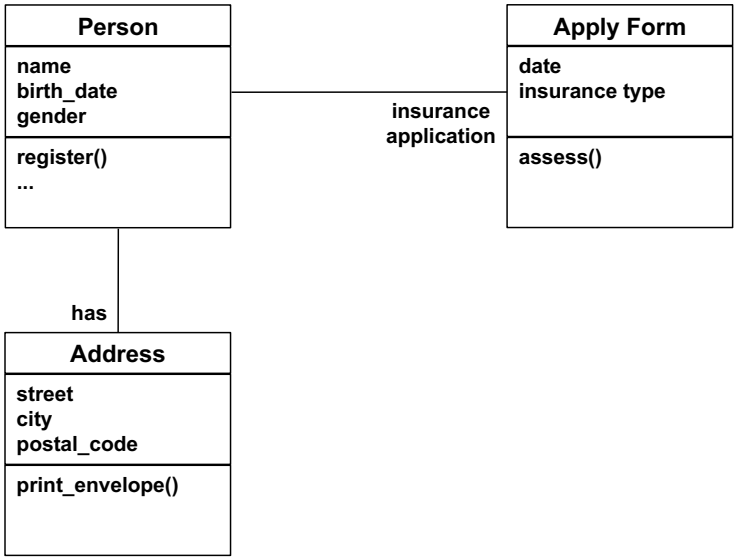


**Figure 20** Class diagram for a concrete insurance application

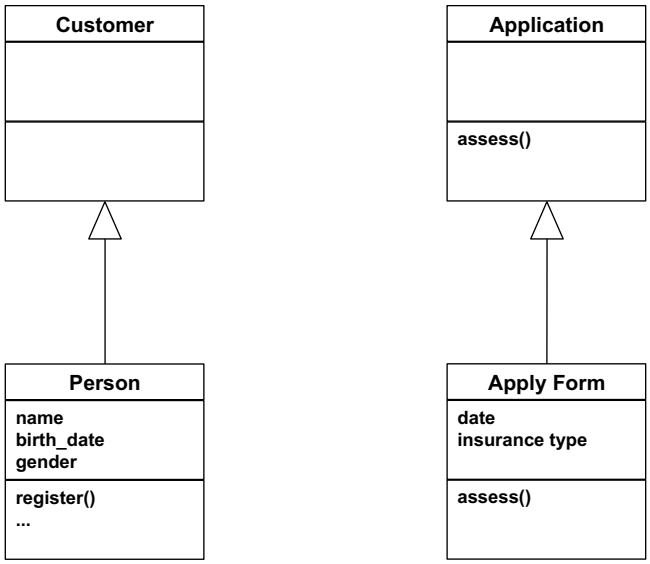


**Figure 21** Mapping domain concepts to types.

 

Figure 21 shows how this is done for two concepts present in Figure 9. Of course, in concrete systems, for every type a corresponding class in the system should implement its interface. One class may implement more than one type, for instance, the class Apply Form may implement both the Application and the Decision types, meaning that the decision is eventually recorded by on the Apply form.

## The CommonKADS template knowledge models

Research that has lead to the CommonKADS methodology has shown that the number of essentially different knowledge intensive tasks is not very large. There exist approximately fifteen different tasks that are used in practice and that really represent a different kind of knowledge. The basic idea is that assessment, that was used as an example in the previous sections, is the essentially the same, in all cases, independent of the actual context of assessment. So be it a physician assessing a patient, an insurance agent assessing an application or a carpenter assessing the quality of a wooden construction, the structure of the task will be essentially the same. Of course, the *content* of the knowledge needed for these tasks will be dramatically different: the rules for assessing a patient will have nothing to do with the rules for assessing woodwork.

CommonKADS expresses this likeliness between different tasks in the presence of a library of *template knowledge models*, also known under the names of reference models or interpretation models. Each entry in this library stands for a prototypical knowledge intensive task that can be instantiated for a specific application. Figure 22 displays the various template models that the CommonKADS book (Schreiber et al., 1999) offers.

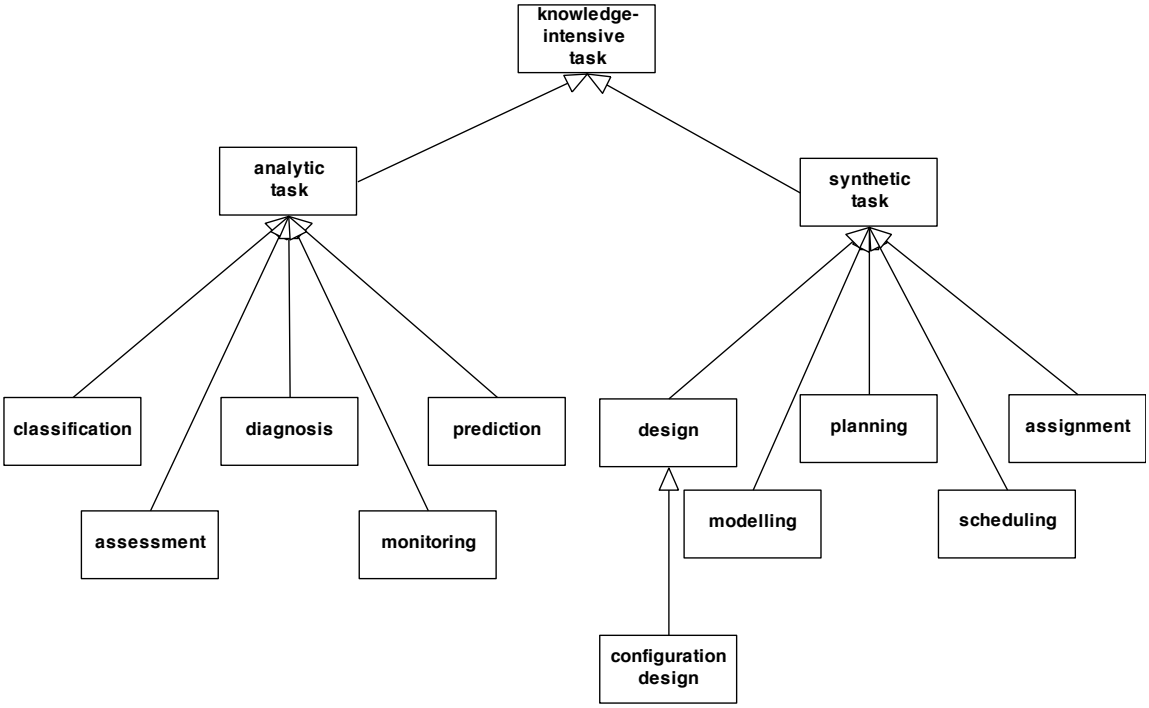


**Figure 22** The collection of template knowledge models offered by CommonKADS (From Schreiber et al., (1999, p. 111).

 

The collection of template models can be “translated” to SDF-II at almost no effort, following the procedure outlined above. However, in reusing the models in a more efficient way, it would be useful to be able to refer to the library of models, instead of having to copy the template model into a specific project. UML offers a facility for doing this, the *framework* construct. A framework is nothing more than a template for a (partial) model, that can be bound to a specific instance in a model. To explain the concept, we will remodel the assessment case, by now first creating a framework and then applying it to the insurance case.

In a framework, the names for the types are usually chosen generally. So we do not speak of an insurance application but of a *case*. Remember that case can also be instantiated by a wooden construction. In building the framework, we start at the level of Figure 11. The inside of the collaboration on the bottom part of this figure can be used as the definition of the framework, as depicted in Figure 23.
The meaning of Figure 23 is that the collaboration inside is isolated in a framework that can be applied in any relevant context. The assess action inside the framework is refined in exactly the same way as depicted by Figure 12 and further.

Having defined this framework, it can be applied to the insurance case. In applying there are two things that must be done. First it has to be stated which framework is used, and second, the relation between types in the framework and the corresponding types in the actual model has to be made. This is displayed in Figure 24. The dashed oval in this figure represents the framework, the arrows to the types of the insurance case indicate the *bindings* of the framework concepts, written next to the arrow, and the types in the model.

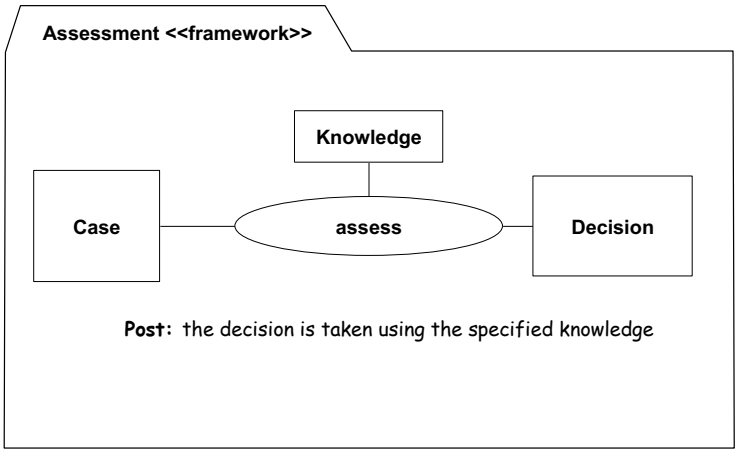


**Figure 23** The definition of the assessment framework

 

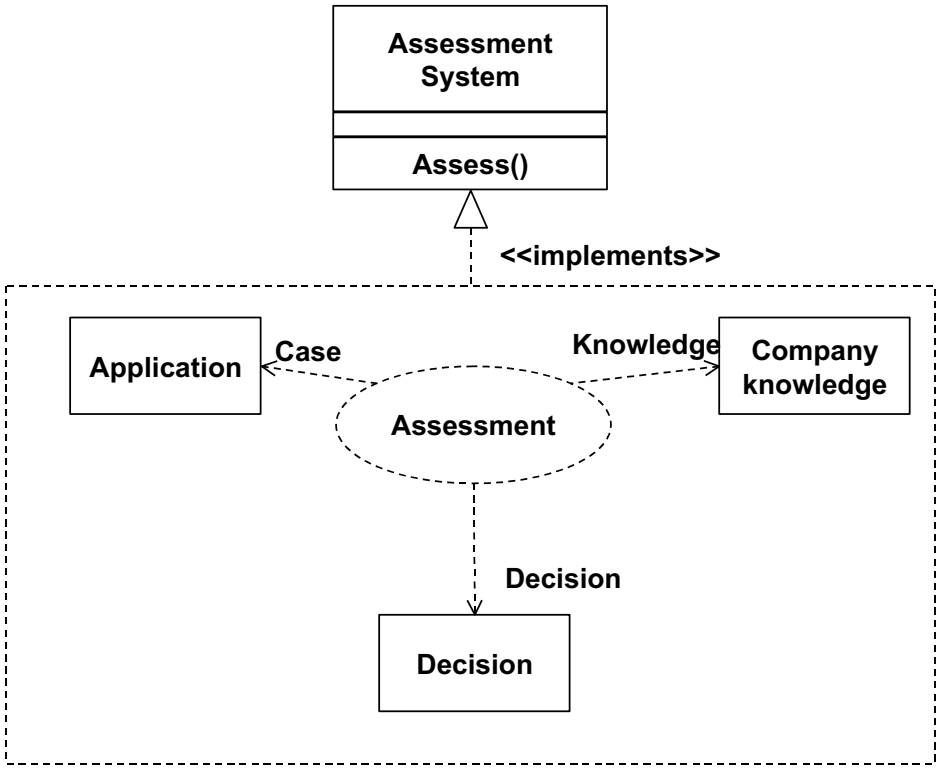


**Figure 24** Application of the assessment framework to the insurance case.

Figure 24 is the exact equivalent of Figure 11. The application of the framework means that the elaboration of the model in terms of refinements is no longer necessary. This is done only once, as part of the description of the framework, and can be reused in every applicable model.

## Summary and guidelines for knowledge modelling

This section on knowledge modelling has outlined the principles of modelling knowledge intensive tasks in UML, using CommonKADS principles. This final part summarises the work and offers some general guidelines for modelling knowledge intensive problems, in the form of a step-by-step plan to take to create a model of a knowledge intensive task. Please note that the following steps are not part of a prescriptive cookbook. The steps should be applied with care and it is possible to use a different approach, to skip, repeat, or change the order of steps. The suggested order serves only as a starting point for planning analysis and design of a knowledge intensive system.

1. *Divide the system into components*

The business model should be subdivided into components, each responsible for a coherent set of functions in the system. This first step is very general and not specific for knowledge intensive systems. The word *system* is to be taken very general here, it can mean the business as a whole, a department or a specific software system. This means that the division into components will take place at various levels.

2. *Represent the relevant tasks as actions and types.*

This step is important to identify the top-level actors in the task and to have a starting point for analysing the various tasks. For each component, the responsibilities should

be expressed in terms of tasks, represented as actions. Types should identify the main actors within the component.

3. *Divide the task into subtasks by decomposing the action and types*
The decomposition should be taken further in order to divide the task into manageable pieces. Each action should be decomposed and refined until a level has been reached that can be considered to be elementary. The criterion for this is that the pre- and postconditions for the resulting actions can be formulated in such a way that the target audience of the document containing the model will know how to take the process of design and implementation further. This is of course a weak criterion, dependent on context and the target audience, but nothing stronger can be given as a general guideline.

4. *Identify knowledge intensive actions*
Actions can be identified as knowledge intensive. The criterion here is that the postcondition for such an action craves expression in terms of knowledge. Again a weak criterion, certainly given that postconditions can be expressed in terms of knowledge rules. Important here is to assess whether there is an independent role of the knowledge needed to realise the action. Try to answer questions like: "should it be possible to maintain the knowledge independent of the system?" or: "is there a need for separate knowledge acquisition for this action?" If the answer is affirmative, the action probably is knowledge intensive.
Knowledge intensive actions can be decomposed just like any other action. The knowledge associated with the action must be decomposed as well. The composing actions may or may not be knowledge intensive, at least one will be. In decomposing knowledge intensive actions, use the CommonKADS template models and the corresponding SDF frameworks wherever possible. These models can either be used as a clear-cut solution or as a starting point for further analysis. When used "off the shelf", the template can be used by incorporating the corresponding framework, when using the template for further analysis, the template itself should be copied into the model and adapted where needed.

5. *Represent knowledge types for the knowledge intensive actions*
For each elementary knowledge intensive action there should be exactly one knowledge type. The goal of this step is to identify the structure of the knowledge sought for, in terms of relations between domain types. It should be stressed again that the structure of the knowledge is independent of its actual representation and implementation. This means that the relations found here do not necessarily lead to rules in a rule-based system. On the contrary, the relations can be implemented in any sensible way. Examples below will make this more clear.

*6. Gather the knowledge to fill the knowledge types*
The knowledge needed for actually realising the knowledge types and thus their corresponding actions should be gathered and represented. This can be in the form of a set of rules, but there are many more possible forms of representing knowledge. Other possibilities are tables like exemplified in Table 4, decision tables, hierarchies, etc. For the eventual implementation of these knowledge types one can use plain programming languages or dedicated tools for knowledge intensive systems, like Prolog, Lisp, or Aion.

 

It is essential that the knowledge is described independently of the structure of the system. This means that the knowledge can be maintained independently of the rest of the system. It even would be possible to provide the knowledge engineer from Figure 9 with a dedicated editor for the knowledge in the system that then can be maintained. The maintenance and updating of knowledge then becomes a separate business process within the organisation in which the knowledge is created, distributed and used.

The examples mentioned have in common that knowledge is seen as static during the life cycle of the system. The analyst gathers and specifies the knowledge and the knowledge is implemented in the system, hopefully in a maintainable form. Currently modern techniques allow for different ways of gathering knowledge and making the knowledge itself more dynamic. Suppose the knowledge intensive task is to select a group of addresses for a direct mail action for a company. The knowledge needed for this selection does not come from human experts but can be generated from the company's databases containing data on income, address, age, etc. of a large number of potential customers. The knowledge needed for this task can be generated from the databases using techniques like data mining and data warehousing.

In such a case, the analysis of the system that should generate the addresses for the mailing can take place following steps 1-5 outlined above. This would yield requirements for the *structure* needed for the knowledge to be generated. The data mining system would then generate the *content* of the knowledge. Of course, the data mining system itself should be represented in the model, as an actor for the action of generating the new knowledge. Figure 25 displays a fragment of such a model. Note the two different roles of the selection knowledge type. In the Create knowledge action, it serves as a normal domain type, in order to change its content. In the Select action, it serves as a knowledge type that uses its content to realise the action yielding the mailing list.

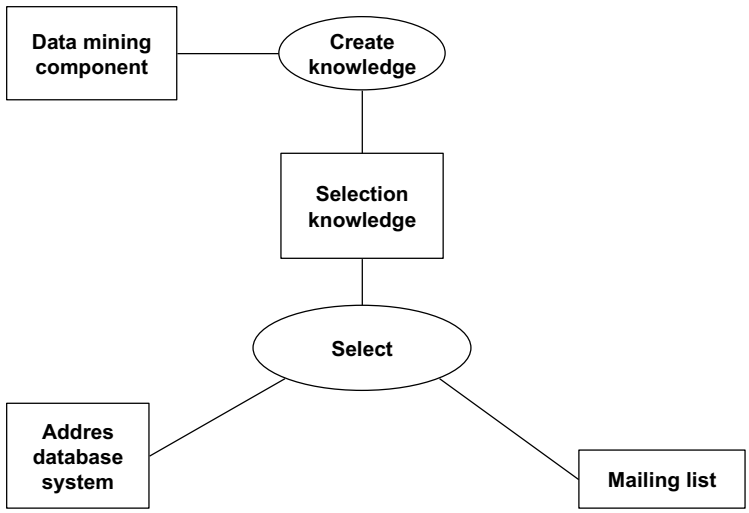


**Figure 25** Collaboration diagram for the case that a data mining system generates the knowledge for a knowledge intensive action.

In a similar way other techniques for creating or applying knowledge can be fitted into a model. In its standard interpretation a knowledge type is the vehicle to *apply* some knowledge in order to realise an action. In order to make sense, it defines a *structure* on the knowledge that is needed. In the end, there is no constraint

 

whatsoever on the way the knowledge is actually represented. Data mining techniques, case based reasoning systems, agent based systems all can play the role of a knowledge type, provided that they can fulfil the responsibilities on the knowledge type in terms of applying the content to instances of the domain types present in the system.

An important feature of the SDF-II that it explicitly leaves undecided where the knowledge should be allocated, i.e. which objects should actually carry and implement the knowledge needed for a knowledge intensive process. Knowledge types represent knowledge and the decision how these types are going to be realised by classes is deferred to the design level. For instance in the assessment example, the knowledge to abstract a case may at the design level be part of the same class that realises the behaviour of the case itself. Other knowledge types may be realised by separate classes or by dividing their responsibility over more than one class.

## *Communication Modelling*

### Approach

A knowledge model specifies the internal structure of an agent carrying out a knowledge-intensive task, such as assessing a loan application or performing diagnosis of a malfunctioning device. This leads to a component specification for this agent. In addition, we also have to model communication between the agent and other agents involved in the task. These other agents are either humans or other software systems. The latter have their own component specifications.

We distinguish two steps in constructing a communication model:

1. Describe a few *typical communication scenarios* with the help of a *UML sequence diagram*. These scenarios are also useful for testing the system later on.
2. Generalize these scenarios by constructing *for each agent a UML state diagram* that describes the events and actions that are exchanged with agents. In this diagram only the external aspects of the agent behavior have to be included, as all other control flow is already specified in the knowledge model.

You have to make sure that the state diagram are consistent with each other by ensuring that each message sent has a corresponding receiving event in another agent.

The approach advocated here is in fact rather standard, both in O-O analysis and in CommonKADS (see Schreiber et al. 1999, Chapter 9). To illustrate the approach we have included a sample communication model for the assessment application

### Communication scenarios

Figure 26 shows a sequence diagram for a scenario in which the loan application of a customer is successfully handled. Each agent has its own “life line”. A box on the line indicates that the agent is carrying out some work. Arrows indicate messages being sent from one agent to another. The vertical lines denote a sequence in time: from top to bottom time passes. In the ca first scenario a customer sends in a loan application. This application is assessed by the assessment system. The application is assessed to be eligible for a loan offer, and a message is sent to the sales department. The sales department sends a formal loan offer to the customer. She thinks a while and then decides to accept the offer by sending a message back to the sales department of the company.

 

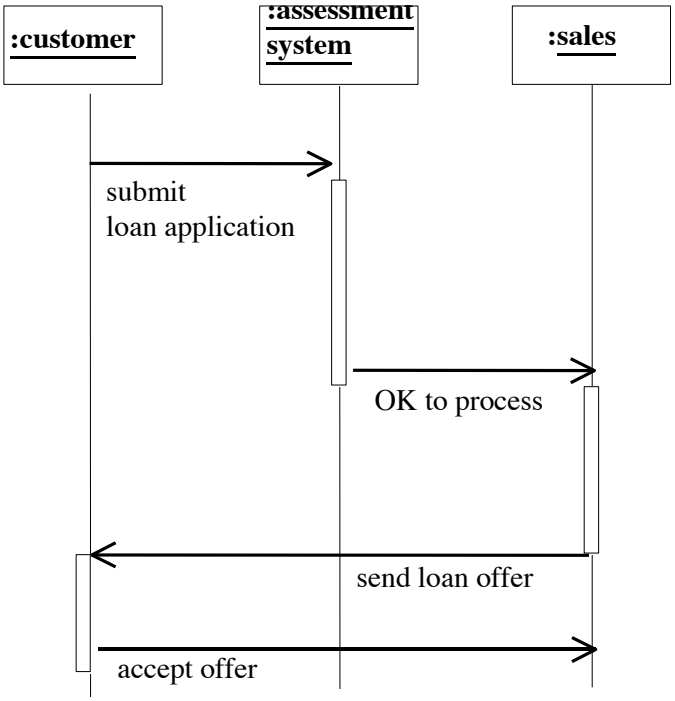


**Figure 26.** Scenario for successful handling of the loan application

In Figure 27 we show a second scenario. Here the application is not considered eligible by the system. As a result a rejection letter is sent to the customer by the system. The sales department is notified at the same time of this turn of events.

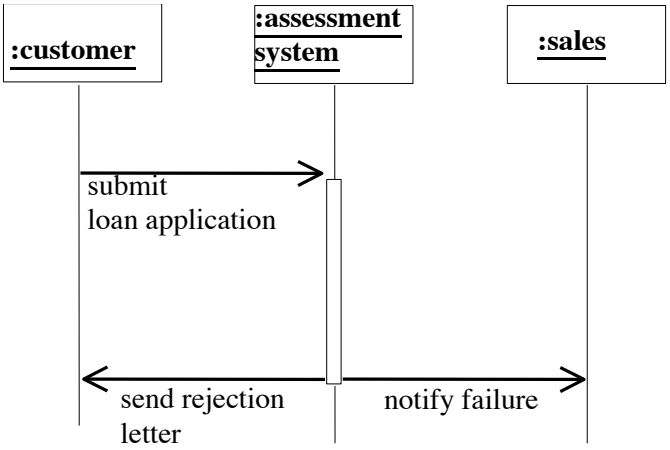


**Figure 27.** Scenario for application rejection by the system

When describing scenarios it is common practice to describe a number of “normal” scenarios, such as the ones described. In addition, one includes scenarios of abnormal situations (e.g., erroneous inputs, and system failures). Scenarios are not only useful for communication modelling, but can also be helpful in defining system tests.

 

## State diagram for an agent

In a sequence diagram we describe one possible sequence of events, also called a “snapshot”. For communication modelling it is necessary to generalize over these scenarios. We can achieve this with the help of a UML state diagram. State diagrams describe the states an object can be in during its “lifetime”. In object-oriented analysis state diagrams are used to model the state behavior of a single object. In the context of communication modelling we are interested in constructing state diagrams for all actor objects.

Figure 28 shows a state diagram for the customer agent. When she has filled in the application form, a message is sent (“application submission”) to another object. The customer then enters a “waiting” state. In case of the vent “offer made” (i.e., the company has made a formal offer) she enters a new state, in which she has to decide what to do with the offer. The customer can either accept or reject the offer.

Figure 29 shows the state diagram for the assessment system. The system becomes active when the event “application submitted” occurs (see the corresponding message sent by the customer). The system then enters the “assessing” state. Depending on the outcome of the assessment (see the guard conditions in Figure 29) message are sent to other agents (the ^ symbol denotes a message in UML state diagrams).

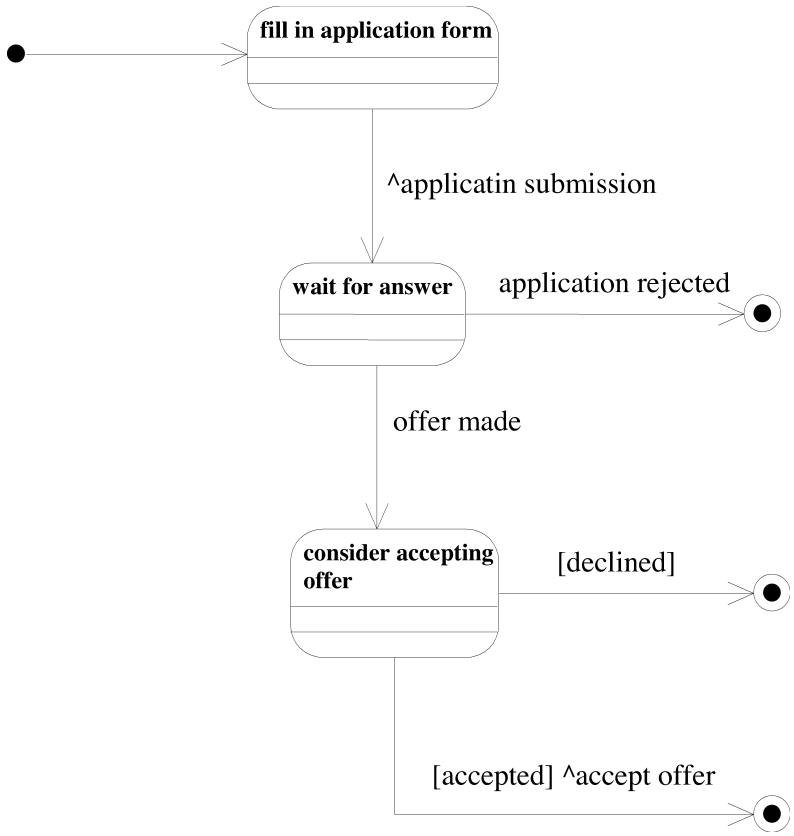


**Figure 28.** State diagram for the customer agent

 

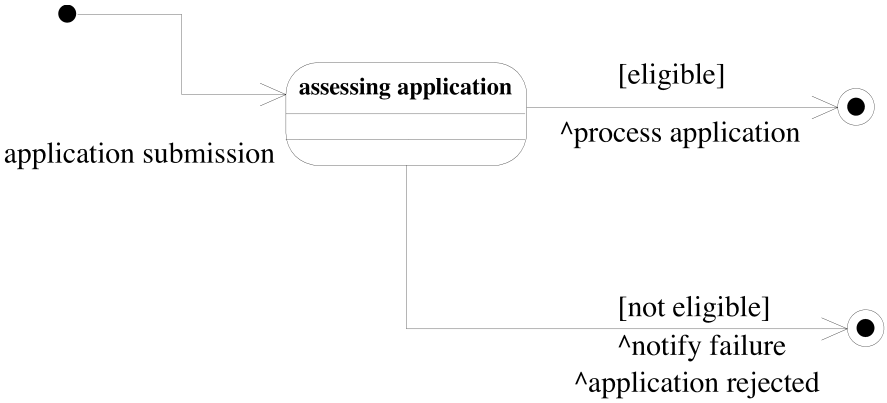


**Figure 29.** State diagram for the assessment agent

To check the consistency of the state diagrams, it is useful to construct a message/event table, in which we list message/event pairs together with their sender and receiver objects. Table 5 shows the table for examples in this section. We usually also included in this table the information items exchanged in the message (if any).

**Table 5 : sender/receiver table for messages**

| Message / event | Sender | Receiver | Information exchanged |
|---|---|---|---|
| application submitted | customer | assessment system | application |
| process application | assessment system | sales | application |
| offer made | sales | customer | offer |
| accept offer | customer | sales | -- |
| application rejected | assessment system | customer | -- |
| notify failure | assessment system | sales | application |

### *Design modelling*

A description of design modelling will be included in Version 2 of this manual (target publication date: October 1, 1999). This section will discuss the main decisions for realizing a knowledge-intensive component. Topics discussed include platform choice, need for reasoning support, and issues related to using a framework approach.
For the moment the reader is referred to the respective chapters in the UML and CommonKADS books.

 

# Project Management

## *Introduction to Project Management Methods*

The approaches taken in project-management methods differ. Some of the methods focus on the deliverables (the products for both the business as for the management of the project) whereas others focus on activities.

The methods focussing on products starts by defining the products that must be delivered at the project end. The product description contains a general description, the relation with other products, the composition of the product, the quality criteria e.g. the criteria mentioned in items 3 and 4 of the step-by-step plan of the guidelines for knowledge modelling (see p. 33). As mentioned in that section the criteria must be made measurable and appropriate quality-check methods should be identified.

Practise has proven that the product-based approach is more successful than the activity based approach. The product-based approach is particularly useful in two respects:

**1. Communication with the stakeholders**

- with the *customer organisation*: what do we get at the end of the project, where do we stand, what is the composition;
- With the *performing organisation*: which products must we deliver, what are the quality expectations.

**2. Management of the project**

The project is divided in manageable parts, namely the products. These parts can easily be delegated to project entities, because the products are clearly defined.

## *Project Management in SDF-II*

The focus of the project management within SDF-II lies on risks. Chapter 15 of the CommonKADS book provides an instantiation of the risk-driven approach for use within knowledge-intensive system development.

In this section we define a work breakdown structure[2] (WBS) which facilitates two standard project approaches which in practise are frequently found:

1. A project with a low technological risk due to the fact that, at an early stage in the project the application task can be modelled. The application task is modelled with the help of an existing and well-understood knowledge-model template.

[2] "The work breakdown structure (WBS) is a tool for defining the hierarchical breakdown and work in a project. It is developed by identifying the highest level of work in the project. These major categories are broken down into smaller components." [http://www.welcom.com/library/glossary/] The WBS consists of technical and management products.

 

2. A project with technological risk related to knowledge modelling. Early in the development process a prototype is built of the reasoning component. This prototype is used to check the feasibility of the system in terms of technical aspects and usability.

Because the two work breakdown structures overlap the total WBS is shown. The brackets [1,2] indicate in which type of project the product or activity is part of the WBS. If absent the product or activity belongs to the subset of both WBSs. SDF users are encouraged to validate and complete the WBS based on your own WBS or project management method.

The work breakdown structure is based upon the following (technical) stages. Below every stage the objective is denoted:

1. Feasibility Study
   - Initial problem understanding
   - Make sure the application problem is suitable for automation
   - Make sure that the system fulfils a real need in the customer organization
   - Verify that no (knowledge-related) risks for the project success exist
2. Systems Analysis
   - Complete / adapt on text analysis [2]
   - Analyze the system in its prospective environment, with special attention for the micro-level interaction between the system and other agents
3. Component Specification
   - Develop a specification of the knowledge component conformant with the requirements set-out in the system context model
4. System Design
   - Make sure that the system architecture can be realized in the customer organization
   - Make sure that as much as possible existing code is used
   - Finalizes knowledge model based on prototype results.

## *Work breakdown structure*

**1 FEASIBILITY STUDY**

**1.1 Business Products**

*B1.1.1* *General Business Model*
for current situation [1]
minimal version [2]

*B1.1.2* *Knowledge Identification* [1]
for all knowledge-intensive tasks involved in a target solution

*B1.1.3* *General Business Model* [1]
for the new situation

*B1.1.4* *Full Knowledge model* [2]
with partial knowledge bases

*B1.1.5* *Prototype reasoner system* [2]
carry out some predefined scenarios

 

### 1.2 Management Products

*B1.2.1* *Risk Identification and Analysis*
especially:
- sufficient organisational support?
- unexpected knowledge-related problems
- Can suitable task template be constructed [2]?
- can types of domain knowledge required be elicited/modelled [2]

*B1.2.2* *Feasibility checklists*
B1.2.2.1 Economic
B1.2.2.2 Technical
B1.2.2.3 Project
see CommonKADS Worksheet OM-5

*B.1.2.3* *Project actions and required organisational changes*

*B.1.2.4* *Feedback of stakeholders*

## 2. SYSTEMS ANALYSIS

### 2.1 Business Products

*B2.1.1* *System Context Model*

*B2.1.2* *Agent <-> Agent Communication specifications*
for all external agents that interact with the system

*B2.1.3* *General Business Model* [2]

### 2.2 Management Products

*B2.2.1* *Risk Identification and Analysis*
especially:
- unexpected technological risks in building the user interface,
- unexpected technological risks with respect to the required connections to other software
- is there still a clear added-value of system?
- development from the perspective of the organisation[3]

*B2.2.2* *Development Assessment Report*
validation by the primary users of the development results

*B2.2.3* *Stage Assessment report*
indication if the feasibility assessment and/or the estimated project effort needs to be adapted

[3] Originally this risk was only identified for the technological risk project, but this risk should be analyzed during every stage of a project, because it justifies the continuation of the project.

 

## 3 COMPONENT SPECIFICATION

### 3.1 Business Products

*B3.1.1* *Knowledge-model Specification*

*B3.1.2* *Filled Knowledge Bases*

*B3.1.3* *Validated Knowledge-Model*

### 3.2 Management Products

*B3.2.1* *Risk Identification and Analysis*

especially:

- possible snags in the application of the chosen template
- availability of domain specialists for domain-knowledge elicitation and component validation

*B3.2.2* *Time Schedule*

*(right type of)* domain specialist

## 4. SYSTEMS DESIGN

### 4.1 Business Products

*B4.1.1* *Design Model: architecture specifications*

*B4.1.2* *Design Model: software/hardware choices*

*B4.1.3* *Design model: application design*

### 4.2 Management Products

*B4.2.1* *Risk Identification and Analysis*

especially:

- unexpected time-overrun because the proper support tools (for reasoner or for user interfaces) are not available in the chosen implementation platform,
- implementation is not consistent with customer standards

*B4.2.2* *Usability and Availability Report*

of the chosen support tools

*B3.2.3* *Project Organisation*

check competence project team with relation to the chosen implementation platform

*B3.2.4* Detailed Implementation and Test Plan

*B3.2.5* Design Validation Report

validated by technical staff of the customer

 

## The SDF-II library

In SDF-II the use of template models is advocated strongly. Template models allow designers, developers and analysts to have a running start with their projects. Template models exist at various levels of analysis and design. A famous collection of template models for object oriented design is the collection of *design patterns* (Gamma, Helm, Johnson, & Vlissides, 1995). This collection provides often used designs for problems that reoccur often in software design, such as recursive structures, model-view-controller models etc. Also the CommonKADS book (Schreiber et al., 1999) contains a set of template models, now models of *knowledge*. One of these models, assessment, was discussed extensively in the chapter on knowledge modelling in this manual. Also, the basic way of applying these models was discussed in this chapter. Here we briefly describe two other examples from the CommonKADS library of template knowledge models. We show how they are converted into the SDF-II OO language, and give some hints on applying them into a real context. For detailed descriptions of the tasks described here, see Schreiber et al. (1999).

### *Diagnosis*

#### Top level description

The goal of a diagnosis task is to find the cause of a malfunctioning system. For instance a car mechanic has to find the cause for a brake that does not work, or a physician has to find the cause of a patient's headaches. The diagnosis task starts with a *complaint*, which is a description of the problem ("the brake does not work"). The result of the task is, in the ideal task, a single hypothesis that describes the cause of the complaint ("there is a leak in the influx pipe of the brake's hydraulic system"). Of course the diagnosis task as a whole is knowledge intensive, so we need an explicit representation of the knowledge needed for the task. Figure 30 displays the basic structure of the diagnosis task.

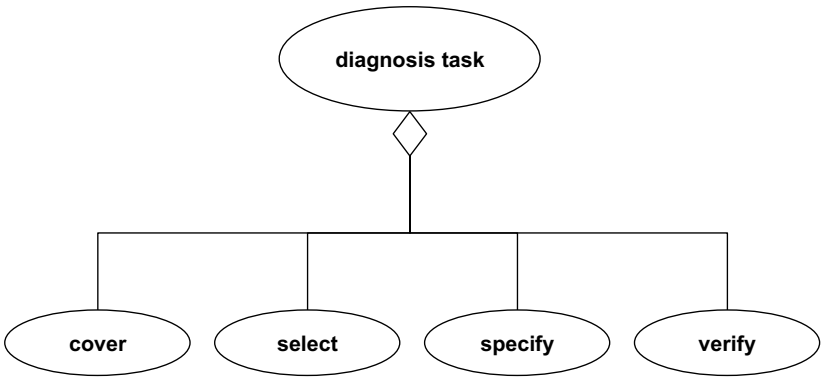


**Figure 30** The basic ingredients for the diagnosis template.

#### Task refinement

This figure serves, like Figure 11 for assessment task, as a definition of the terms that play a role in diagnosis. In order to describe the contents of the task we can now refine this model to yield more basic actions in the task performance. Following the

 

decomposition by Schreiber et al., the task can be divided into four knowledge intensive subtasks, as displayed in Figure 31. The *cover* action generates a new hypothesis, based on the input from the complaints and the knowledge of the system's structure. In order to do this, the action requires knowledge about the structure of the system (e.g. knowledge of the brake system in a car). The select action selects a hypothesis to consider as "current" meaning that this hypothesis will now have the focus of investigation ("first consider the a leak in influx pipe"). This can be done based on heuristic knowledge. Specify means finding indicators on the truth value of the hypothesis. For instance, when there is a leak in the influx pipe the oil pressure in the brake should be zero. This yields an observable (the pressure in the braking system) which can be measured. After measuring, the verify action matches the result to the prediction following from the hypothesis, yielding a truth-value for the hypothesis. By repeating these actions hypotheses can be successively ruled out, until there is only one hypothesis left, no more observables can be specified, or all hypotheses are ruled out. Figure 32 displays the complete collaboration diagram for diagnosis, including the relevant knowledge types. Figure 33 displays the control structure for a data driven strategy: first all possible hypotheses are generated and then they are systematically ruled out until the process succeeds or fails. A possible other strategy would be to generate only one hypothesis and keep it until it is ruled out, triggering the generation of a new hypothesis. This would be based on the same collaboration diagram, but with a different activity diagram specifying the control over the actions.

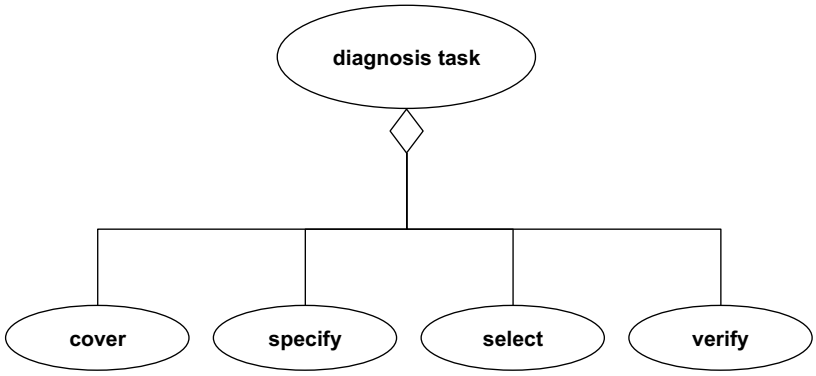


**Figure 31** Decomposition of the diagnosis task, as the refinement model for the transition between Figure 30 and Figure 32.

 

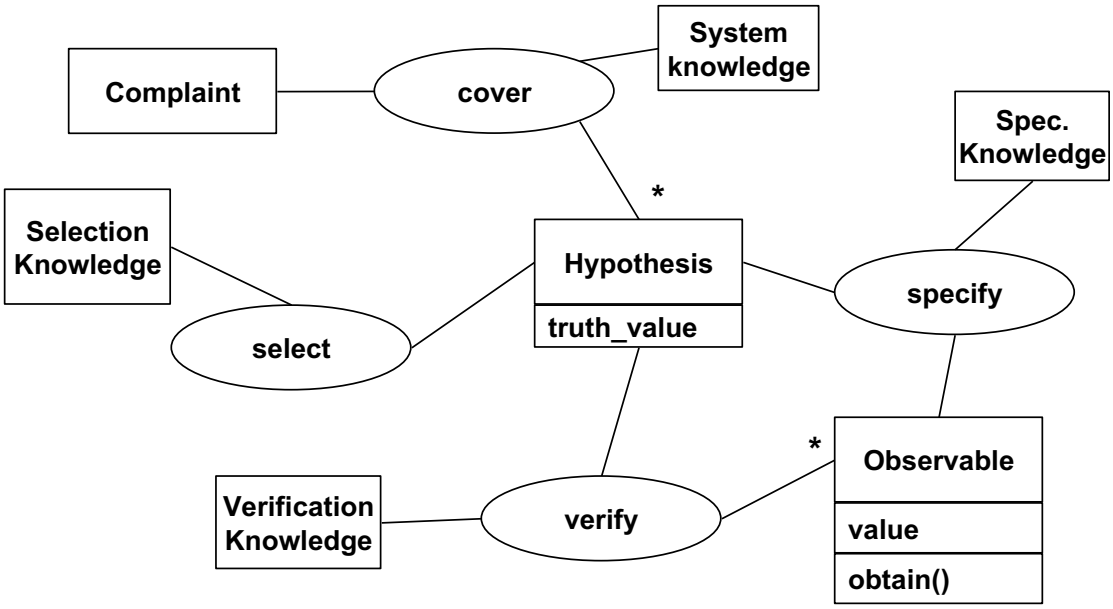


**Figure 32** Collaboration diagram for diagnosis.

## Domain knowledge characterisation

In the diagnosis task four knowledge types are essential:

1. *System knowledge*
   System knowledge is knowledge about the structure of the system: the system's component structure, connections between components, structures of causality. This knowledge is used to generate hypotheses about possible causes for the complaint. Hypotheses take the form of a component of the system that is possibly faulty.

2. *Selection knowledge*
   This knowledge is used to select a current hypothesis. The observables associated with this hypothesis are the ones that will first be measured. This means that attempts will be made to rule out the current hypothesis. Selection knowledge consists of heuristic rules that indicate which hypothesis can best be considered next. It is possible to make this a random selection, yielding empty selection knowledge.

3. *Specification knowledge*
   Specification knowledge links hypotheses to observables. It is related to the system knowledge as it can infer from hypothesis back to observations on measurement points in the system

4. *Verification knowledge*
   Verification knowledge is able to yield conclusions based on observations. The values of observables are interpreted to draw a conclusion on the hypothesis. This can be simple rules, but also more complex knowledge is possible, especially when values of multiple observables contribute to drawing the conclusion.

 

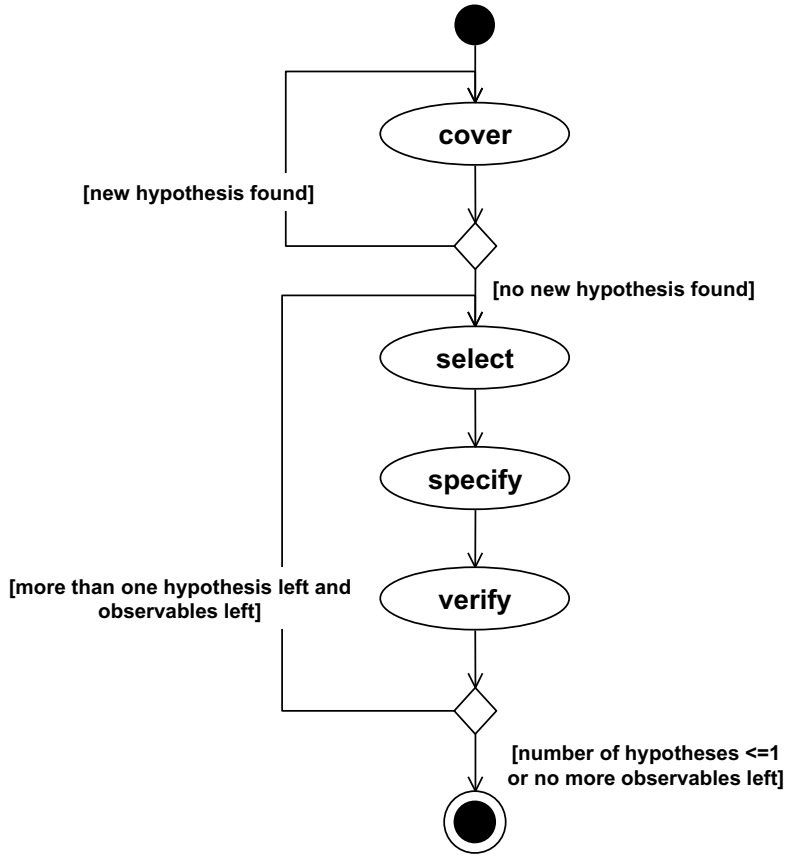


**Figure 33** Activity diagram specifying the control over the diagnosis task.

### *Assignment*

#### Top-level description

Assignment is a task concerned with assigning resources to subjects. A well-known example is office assignment: rooms (the resources) are assigned to employees (the subjects). The top-level diagram is shown in Figure 34. Assignment takes as input a set of subjects and a set of resources. The task output is a set of allocations, which are in fact relations between subjects and resources. This is modelled in Figure 34 through an association class.

#### Task refinement

The template for assignment consists of three steps:

1. *Select subset of subjects*
   From the full set of subjects a subset is selected (this can be just one, of course). This selection process is typically guided by domain heuristics that are used to order the assignment process. Typically, subjects with tight allocation constraints are selected early on in the assignment process. For example, in office assignment the people with coordinating functions (managers, secretaries) are selected first.

 

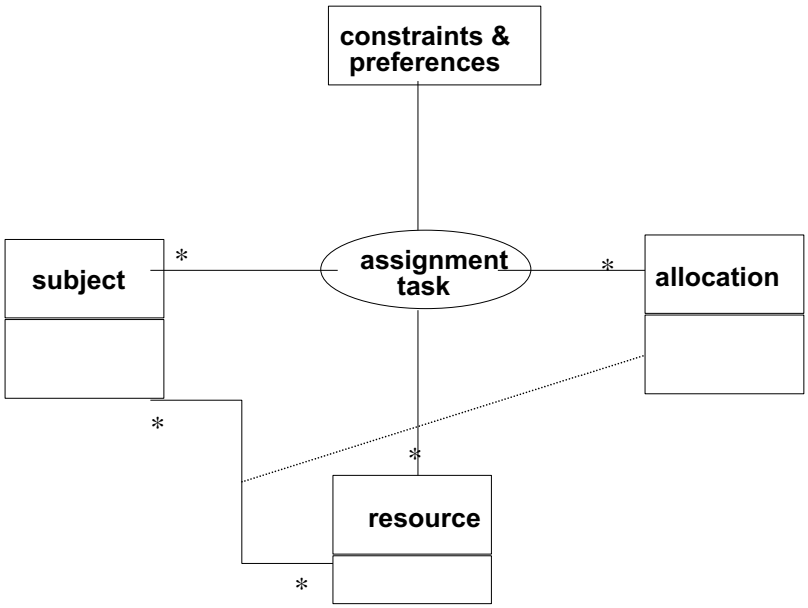


**Figure 34:** assignment template: top level diagram

2. *Group subjects*
   In some domains, subjects can get the same resource; e.g., employees can be assigned to the same office. Forming groups requires a special type of domain knowledge concerned with subject's preferences and constraints (e.g., smoking). In other cases the grouping step just produces groups with a single subject (in office assignment case this is the case when an employee is eligible for a single room),

3. *Assign a subject group to a resource*
   In the final step one or more subjects get a resource assigned to them. For example, two secretaries get a central office. This step requires a different type of constraints and preferences than the grouping step.

Figure 35 shows the refinement of the assignment task as provided by the CommonKADS task template. Figure 36 shows the corresponding specification of control. We see that the control flow takes the form of two nested loops. In the outer loop subsets of subjects are selected. In the inner loop groups are formed from these subsets and subsequently a resource is assigned to each group.

 

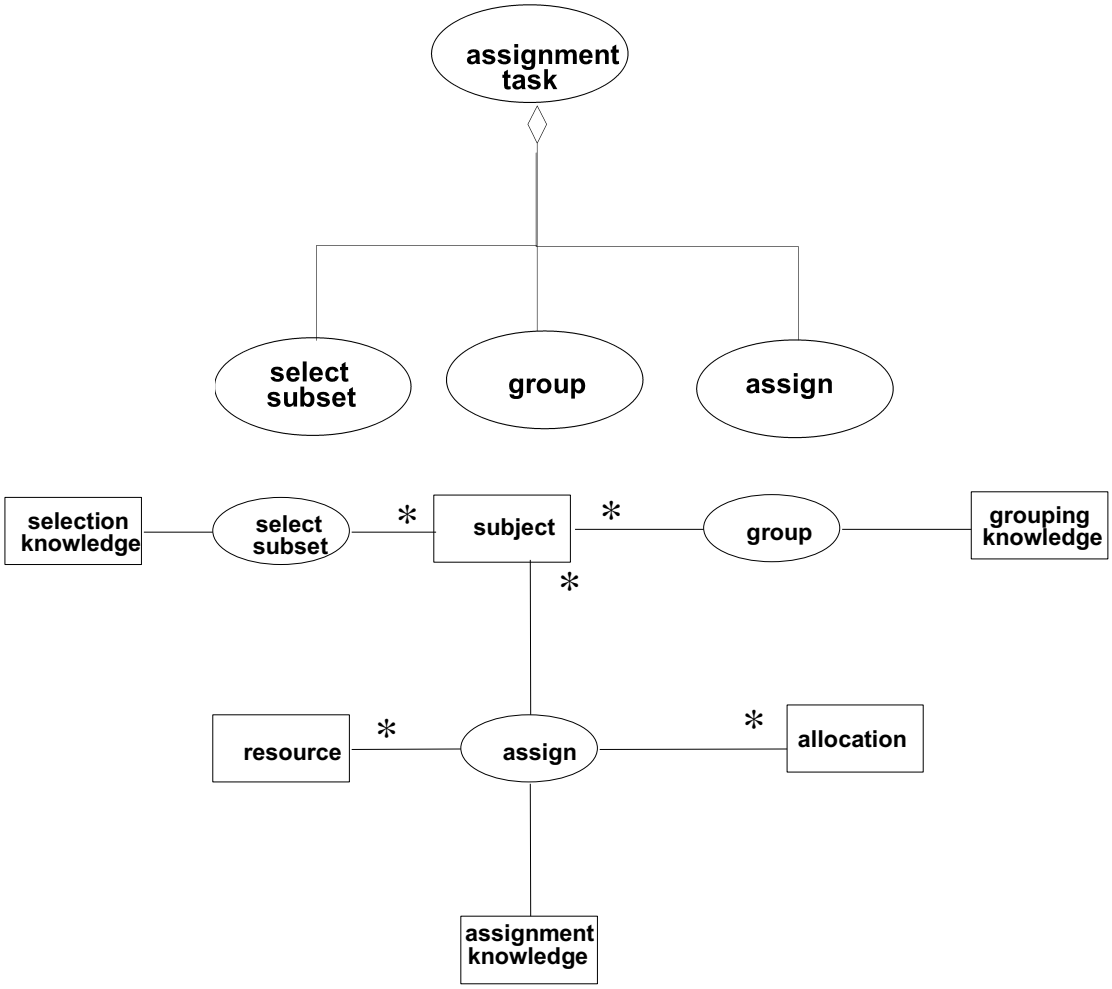


**Figure 35:** refinement of the assignment task

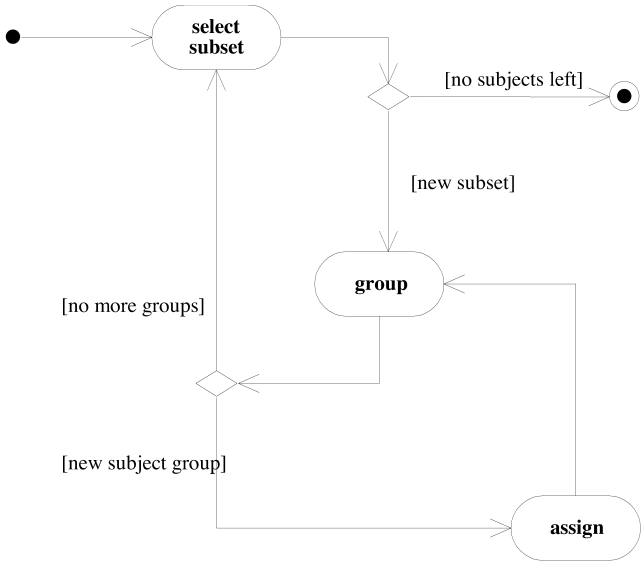


**Figure 36:** activity diagram describing control flow in the assignment task template

### Domain knowledge characterisation

The assignment template typically requires the following types of domain knowledge to be specified:

1. Ordering heuristics for the selection function: e.g, first managers & secretaries, then other staff.
2. Constraints and preferences for grouping subjects, e.g., subject conflicts (smoker and nonsmoker), subject synergy (same type of work).
3. Constraints and preferences for assigning a resource to a subject (group), e.g., managers and secretaries need central offices.

### *How to use the templates*

The templates described in this chapter, as well as those from the much larger set in Schreiber et al. (1999) form a starting point only for further analysis. There are two ways of using them in an actual problem. First the templates can be used as such, meaning that they can be mapped one to one on domain constructs. In this case, the templates can be interpreted as being *frameworks*. See the text on page 31 on how to apply frameworks within SDF-II. Mappings are made from types in the model to types in the framework and the details of the actions are filled in.

Another way of using the templates is taking them as starting point for further analysis. By copying the template, filling in the domain concepts on the place of the abstract types in the template description and then critically reviewing the actions in the template, the template can be moulded for a specific application domain.

### *Converting CommonKADS templates to SDF-II*

The examples in this book were all based on existing templates from CommonKADS. These were converted into the UML notations employed by SDF-II. In converting these templates some steps are taken that are not trivial. Especially roles in CommonKADS inference diagrams do not always map directly to types in UML. In this section a systematic approach is sketched to go through this conversion.

1. *Draw a top level action diagram for the complete task*
   This is useful to see which roles actually are relevant for the task when viewed from the outside. Roles that play only an 'internal' role in the task will not be visible on this diagram. This diagram can then be refined to the inference level.

2. *Identify types based on the CommonKADS roles.*
   CommonKADS inference diagrams contain roles that take the place of domain concepts, just like UML types represent domain concepts. However, there is a main difference. CommonKADS uses a new role for every changed state and for plural and single objects. For instance, in the diagnosis template, there are roles for differential (a collection of hypotheses), hypothesis and result (the truth-value of the hypothesis). In the SDF-II template these are combined into a single role, where the collection aspect is indicated by the multiplicity in the diagrams. The result role is reduced to an attribute of the hypothesis type.
   In general the steps to take are first to look for roles that are collections of other roles in the same diagram, and to look for roles which actually are attributes of other roles when looking at it from an object oriented perspective.

 

3. *Identify basic action*
   Actions correspond to inferences. In most cases, they can simply be mapped on each other. Sometimes it is clear that inferences can be directly mapped onto operations on a type. Transform functions in CommonKADS can usually be mapped on operations on types.

4. *Attach knowledge types*
   All knowledge intensive actions should have a knowledge type attached. In SDF-II it is good practice to explicitly show these knowledge types in collaboration diagrams.

 

## Conclusions

SDF-II is a powerful framework for integrating modelling of knowledge-based systems into UML and the analysis and design of object oriented systems. The major strength of the framework is its framework character. No attempts have been made to reinvent the wheel. The framework combines the strong points of the CommonKADS methodology, UML, inspired in many ways by Catalysis. The framework is open. It should not be very problematic to integrate other approaches, such as agent based technology into the framework. The strategy to use is to be aware of the major features of the approach, use UML notation wherever possible and, where co-operation with other systems is necessary, use a component based approach and specify the interfaces between components.

Using SDF-II will allow to integrate new technology into existing business systems, and make organisations aware of the knowledge they possess and how this knowledge must be created, distributed maintained and discarded. SDF-II provides the vehicle for describing the processes concerned with this management and use of knowledge.

 

## References

Booch, G., Rumbaugh, J. & Jacobson, I. (1998). *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

D'Souza, D. & Wills, A. C. (1999). Objects, Components, and Frameworks with UML: The Catalysis Approach. Reading MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design patterns, elements of reusable object-oriented software.* Reading, MA: Addison Wesley.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). Object-Oriented Modelling and Design. Englewood Cliffs, NJ: Prentice-Hall.

Schreiber, A. Th., Akkermans, J. M., Anjewierden, A. A., de Hoog, R., Shadbolt, N. R., Van de Velde, W. & Wielinga, B. J. (in press). Knowledge Engineering and Management: The CommonKADS Methodology. Boston, MA: The MIT Press.

Vorgers, P. (1999) *From CommonKADS to Catalysis*. Masters thesis. Utrecht: Kenniscentrum CIBIT

Warmer, J., Kleppe, A. (1999). *The Object Constraint Language, Precise Modeling with UML*. Reading, MA: Addison-Wesley

Yourdon, E. (19993). Yourdon Systems Method: Model-Driven System Development. Englewood Cliffs, NJ: Prentice-Hall.

 