Tool Support for Software Development

Progress Report

Michael Thomsen

Department of Computer Science
University of Aarhus
Denmark

May 2000

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

TABLE OF CONTENTS

1. Introduction 1
1.1 Activities during Del A 1
2. Object-Oriented Modelling 3
2.2 Tool support for creating object models....4
2.3 Object models and relational databases....10
2.4 Object models and user interfaces 15
2.5 Section summary 19
3. Software Architecture 20
3.2 Architectural patterns for user interface architectures....22
3.3 Tool integration and software architecture....26
3.4 Section summary 29
4. Work in Progress and Future Work 30
4.1 CSCW & groupware....30
4.2 Goals and activities 30
5. Appendices

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

1. INTRODUCTION

This report is delivered as partial fulfilment of the requirements for qualifying to the second part of the Danish Ph.D. study. The report thus has several purposes: 1) to present an overview of my work during the first part ("Del A") of my Ph.D. study, 2) to present extracts of my results during the first part, and 3) to discuss my directions for further research in the second part ("Del B") of the Ph.D. study.

The report is structured as follows: Section 1.1 presents an overview of my activities during the first part of the study. Section 2 concentrates on my results related to object-oriented modelling, and section 3 presents results related to software architecture. Finally, Section 4 presents my ideas and directions for the second part of the study.

1.1 Activities during Del A

The first part of my Ph.D. study started September 1st, 1998 and it is expected to end in June, 2000. I will, however, in this account of my work in the first part of the study also include the research-related activities that I took part in before I was admitted into the Ph.D. study, as these have influenced my work much.

Figure 1 presents a schematic overview. The first research project I was involved in was the *Dragon Project*, which ran from February 1997 to June 1998, although the University group continued to write papers related to the project until May 1999. The project was a joint project between Maersk Line, a large, globally distributed container shipping company, and University of Aarhus, Denmark and it received financial support from the Centre for Object Technology (`http://www.cit.dk/COT`). The overall goal of the project was to create a series of prototypes of a world-wide customer service system that should support handling of interaction with customers, e.g. in formulating prices for transport of containers (quoting), in booking containers, and in arranging inland transportation of containers.

From a research point of view the main goals where 1) to investigate the use of object-oriented development to create a very large software system, 2) to investigate the use of ethnography and cooperative design in a large project, and 3) to investigate cooperation between a group of developers all with very different competencies. The overall lessons are described in (Christensen *et. al*, 1998b). After the project had officially ended, the university group also worked on the topic of software architecture. The lessons from this are presented in (Christensen *et al.*, 1998a, 1999a) and in extract in section 3.1.1

	1997				1998				1999				2000		
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	…
Dragon		The Dragon project					Dragon Architecture								
OO-RDB				Tech. Report											
Arch.Stud.											SASG		SESG		
Arch.Pat.							DPF course		Pat. article						
OO-UI							Ecoop paper								
Knight									PostWIMP		The Knight Project				…

Figure 1. A schematic overview of my work

www.manaraa.com

Concurrently with our discussions of software architecture in the Dragon Project group, I followed the course on *Design Patterns and Frameworks*. During this course I worked with Klaus M. Hansen on the topic of architectural patterns for interactive systems. This resulted in a paper for TOOLS Asia 1999 (Hansen & Thomsen, 1999), as will be further discussed in section 3.2.

During the Dragon Project, I was also to some extent involved in another case in the Centre for Object Technology, namely Case 4 (`http://www.cit.dk/COT/case4-eng.html`) that focuses on the integration of object technology with non-OO systems. In this project, I primarily worked on an object-oriented framework for accessing relational databases. My work in this project was mostly concentrated on communicating the results of existing research to the project's industrial partners and not so much on new research. The most important result of this activity was an official COT technical report (Thomsen, 1998a) that was published on the COT homepage, and another report presenting implementation specific details (Thomsen, 1998b). An extract of this work is presented in section 2.2.

Another spin-off from the Dragon Project was some work that I did in the fall of 1998 and the spring of 1999 on the relation between object-oriented models and user interfaces. In this study I looked at the concrete problem domain model and user interface produced in the Dragon Project and tried to describe the relation between them. The main results of this work are described in an ECOOP 1999 workshop position paper (Thomsen, 1999). The work is summarised in section 2.4. It should be noted that both this work, is less finished work that my work on the Dragon project and the Knight project (see below) and that it thus constitutes work in progress.

The second large project that I have been involved in is the *Knight Project* (`http://www.daimi.au.dk/~knight`). The Knight Project originally started as a project in the *PostWIMP* course taught by Wendy Mackay and Michel Beaudouin-Lafon, but it has been continued as a regular research project with a substantial amount of work following the course.

The overall topic of the Knight Project is tool support for object-oriented modelling. Within this frame we have investigated several issues: 1) the current support for and practice of object-oriented modelling, 2) the use of gestural input on electronic whiteboards to achieve a 'transparent', non-intrusive user interface, 3) integration with existing CASE (Computer Aided Software Engineering)-tools using component technology and XMI, and 4) the use of the Unified Modeling Language for initial, creative object-oriented modelling. The results so far are documented in a number of papers (Damm *et al.*, 2000a, 2000b, 2000c, In Prep.). Part of the results with respect to issue 1), 2) and 4) will be presented in section 2.2 and the results with respect to issue 3) will be presented in section 3.3.

A list of the publications I have authored or co-authored before or during Del-A can be found in Appendix B. Also, Appendix C presents an overview of my study and teaching activities.

www.manaraa.com

2. OBJECT-ORIENTED MODELLING

Object-oriented languages originate from the Simula language developed in Norway in the sixties (Dahl *et al.*, 1966). Simula was designed to be a language for describing simulations. It is therefore not surprising that one major benefit of object-oriented languages is an underlying conceptual framework providing means for modelling. This conceptual framework provides object-oriented languages with the ability to model the concepts and phenomena in the "real world" that the envisioned computer system is concerned with. It is important though to remember that the world is not object-oriented; object-orientation is rather a perspective on the world, which of course only captures some of the aspects of the world.

Object-oriented development, then, is based on an understanding of the settings that the system will eventually support. This setting we will refer to as the referent system or the problem domain and the process of translating referent system specific concepts to concepts within the computer system we refer to as *modelling* (see Figure 2 below). The result of the modelling process we refer to as the model system, or in the case of object-oriented modelling as the *object model* or the *OO model*. Using the model to refer back to the referent system context we denote *interpretation*. (Madsen *et al.*, 1993, pp. 289; Knudsen *et al.*, 1994, pp. 52).

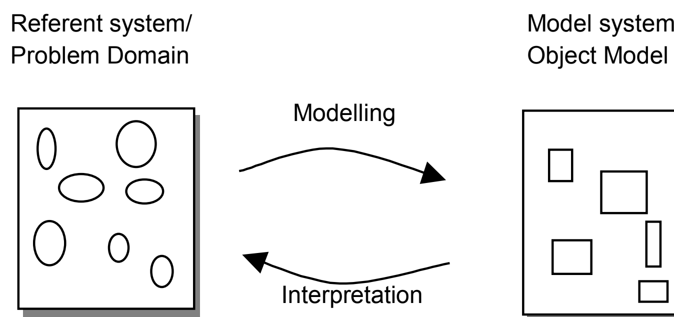


Figure 2. Modelling and interpretation

Besides the direct benefits from modelling, object-orientation also provides a unifying perspective on the whole development process. The conceptual framework provides means for organising knowledge during the analysis phase, object-oriented design notations provide means for design, and object-oriented languages facilitate implementation (Madsen, 1996).

2.1.1 Different model perspectives

When creating or interpreting models it is important to remember that these models often are made from different perspectives. Different perspectives on a problem domain yields different models, often intended for different purposes. If one interprets a model from a different perspective than the one it was created from misunderstandings are inevitable.

Fowler (1997) mentions three perspectives (following Cook & Daniels (1994)): *Conceptual*, *Specification,* and *Implementation*. In the conceptual perspective only the concepts found in the problem domain are described and they are described without regard to the actual programming language used. During specification the concepts are further specified. This leads to the interfaces or types of the concepts. Finally, in the implementation perspective implementation details are added. The Unified Software Development Process (Jacobson et al., 1999) seems to have a similar division where

المنارة للاستشارات
www.manaraa.com

the first model found is the Domain model. This model later influences and is extended by the Analysis, Design, and Implementation models.

In summary, there is not only one object model for a given referent system, and during a project the object model often evolves as development progresses. This is important to notice, as the object model found in later phases most likely will have less direct relation to the problem domain. This can e.g. be the result of non-functional requirements such as time- or space-requirements.

2.1.2 Creating object models

The actual work involved in formulating the object model is far from easy an easy task, as it requires a thorough knowledge of the problem domain. This is especially so in the context of iterative and experimental development. Take as an example the Dragon Project. In this project we experimented with an interdisciplinary approach where a number of different competencies such as an ethnographer and a cooperative designer worked with the object-oriented developers (Christensen *et al.*, 1998b). Although different, they share a common frame of reference – the prototype and the practice it is intended to support. Generally speaking we can say that:

- Ethnography provides a concrete understanding of work's real time accomplishment in contrast to idealisations and formal glosses.
- Cooperative design provides an understanding of the relationship between current and future practice through the experimental formulation of concrete design visions and solutions
- OO provides a concrete relationship between design visions and the application in and through formulating a model utilising concepts derived from practice.

The instances of work and the prototype provide and maintain important common reference points between the three perspectives throughout development. Also, a number of various resources and artefacts were used in producing the model in the Dragon Project. These included sessions with business representatives, descriptions of the database of the current system, Yourdon diagrams of current and future business processes and, of the utmost importance, our own collaborative studies of current practice (Christensen *et al.*, 1998b).

2.2 Tool support for creating object models

Tools for creating the object models are in many cases useful. A variety of CASE tools have been created to support e.g. code generation, documentation, and flexibility in editing and changing the diagrams (Lyvtinen & Tahvanainen, 1992). However, in practice these tools are supplemented with whiteboards, especially in creative phases of development as whiteboards are easy to use, highly flexible, do not hamper the creativity of the user, and can be used for a variety of tasks. These conflicting advantages and disadvantages of whiteboards and CASE tools can lead to frustrating and time-consuming switches between the two technologies. Our goal in the Knight project is to design a tool that offers the best of both worlds.

2.2.1 Current modelling practice

To understand the current practice of modelling further we have conducted three user studies: An informal study of the modelling carried out during the Dragon project, a two-week study of modelling in another COT project and a detailed one-day study with video-taping of a software architecture restructuring session at a commercial software development company. The studies are described in large detail in (Damm *et al.*, 2000a, 2000c).

The two user studies highlighted the effectiveness of ordinary whiteboards as tools for cooperative design. They support a direct interaction that is easy to understand, and they never force the developer to focus on the interaction itself. Whiteboards allow several developers to work simultaneously and thus facilitate cooperation. They do not

المنارة للاستشارات
www.manaraa.com

require a special notation and thus support both formal and informal drawings. Notational conventions can easily be changed and extended.

Whiteboards, however, miss several desirable features of CASE tools. Without the computational power of CASE tools, making changes to the drawings is laborious, the fixed space provided by the board is too limited, and there is no distinction between formal and informal elements. There is also no support for saving and loading drawings.

These observations lead to the following design criteria for a tool to support object-oriented modelling:

- *Provide a direct and fluid interaction.* A low threshold of initial use is needed and the tool should never force the developer to focus on the interaction itself. The whiteboard style of interaction is ideally suited for this.
- *Support cooperative work*. Several developers must be able to work with the tool cooperatively. Informal cooperative work with domain experts as well as software developers must be supported.
- *Integrate formal, informal, and incomplete elements.* Besides support for formal UML elements, there must be support for incomplete UML elements and informal freehand elements. Also, the support for formal UML elements must be extensible, to allow for the introduction of new formal elements.
- *Integrate with development environment*. Integration with traditional CASE and other tools is needed. Diagrams must be saved and restored, and code must be generated and reverse engineered.
- *Support large models.* A large workspace is needed. In addition, there must be support for filtering out information that is not needed at a given time.

2.2.2 Supporting current modelling practice

The user studies and the study of other CASE tools have been used as a basis for implementing a tool supporting collaborative modelling and implementation. This tool, the *Knight tool*, uses a large, touch-sensitive electronic whiteboard (currently a SMART Board (`http://www.smarttech.com`; see Figure 3) as input and output device. This naturally enables collaboration via turn-taking among developers and users. How the interaction with the tool and functionality of the tool enables modelling and interpretation is discussed in the next sections.

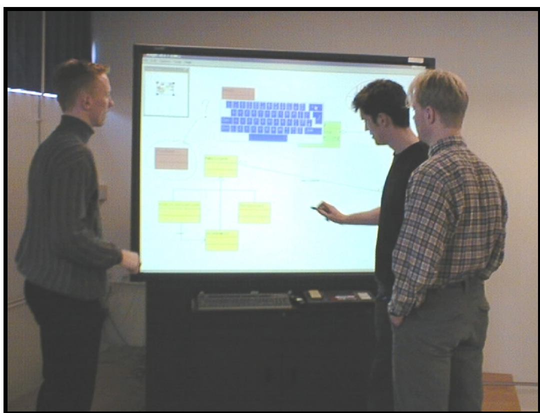


Figure 3. Collaboration around a SMART Board

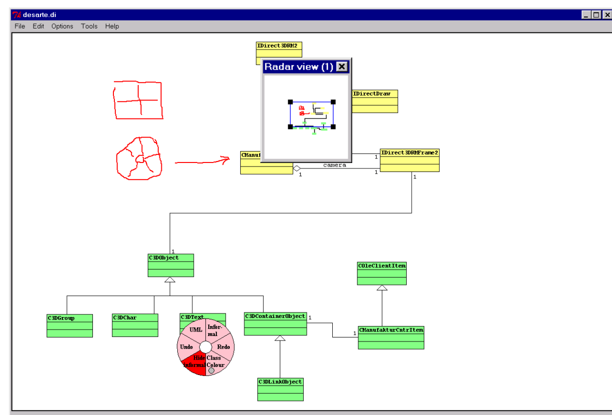


Figure 4. Knight user interface

للاستشارات المنارة
www.manaraa.com

The Knight tool

The Knight tool supports the Unified Modelling Language (UML; Rumbaugh *et al.*, 1999) notation. A major design goal of the Knight tool was to make the interaction with the tool similar to that on an ordinary whiteboard. Therefore, the user interface (Figure 4) is very simple: it is a plain white surface, where users draw UML diagrams using non-marking pens.

The interface is based on gesture input. For example, in order to create a new class, the user simply sketches a rectangle, which the tool then interprets as a class (Figure 5). The gesture recognition is done using Rubine's algorithm (Rubine, 1991).

Figure 5. Recognition of the gesture for a class

Also using gestures, a user may associate two classes by drawing a straight line between them. In general, the gestures for creating UML elements have been chosen to resemble what developers draw on ordinary whiteboards. Furthermore when using gestures the input is done directly on the workspace, and there is no distance in time or space unlike when using e.g. toolbars or dialog-boxes. This directness makes the gestures easier to learn and use.

To support step-wise refinement, we use *compound gestures* (Landay & Myers, 1995), that combine individual gestures that are either close in time or space. This is among other things used to create UML inheritance relationships. Analogous to the interaction on an ordinary whiteboards, where users normally draw a line and then an overlaying triangle, users of the Knight tool also first draw a line gesture resulting in a generic UML relationship, and then a triangle gesture at the appropriate line end to further specify that the relationship is of the type inheritance.

Eager recognition (Rubine, 1991), continuous attempts to classify the current gesture being drawn, is used for the move operation. To move an element, the user draws a squiggle gesture on the item to be moved. When the squiggle gesture is classified with a high confidence, feedback is given in order to show that the gesture was recognised – in this case the item follows the pointer. In this way one single gesture specifies both the command and its parameters resulting in a fast interaction.

Informality vs. formality. A continuum from informality to formality is supported in two ways. First, users may draw incomplete diagram elements, such as a relationship element belonging only to one class (Figure 6).

Figure 6. A relationship with only one class specified

Second, a separate *freehand* mode is provided. In freehand mode, the penstrokes are not interpreted. Instead, they are simply transferred directly to the drawing surface. This allows users to make arbitrary sketches and annotations as on an ordinary whiteboard (see Figure 4 upper-left). Unlike on whiteboards, these can easily be moved around, hidden, or deleted. Each freehand session creates a connected drawing element that can be manipulated as a single whole.

Navigation. The tool provides a potentially infinite workspace. This allows users to draw very large models, but is potentially problematic in terms of navigating. Generally there is a need for an easy way of navigating from one point in the diagram to another point in the diagram, and it is desirable to be able to focus on a smaller part of the diagram while preserving the awareness of the whole context. To achieve this in

المنارة للاستشارات
www.manaraa.com

the Knight tool, any number of floating radar windows may be opened (Figure 7). These radar windows, which may be placed anywhere, show the whole drawing workspace, with a small rectangle indicating the part currently visible. Clicking and dragging the rectangle pans while dragging the handles of the rectangle zooms.

Ordinary pull-down menus are not appropriate for activating the functionality of the tool given the large size of the whiteboard screen. Instead, we use gestures as explained above and pop-up menus that can be opened anywhere on the workspace. The pop-up menus are implemented as *marking menus* (Kurtenbach, 1993) that are opened when the pen is pressed down for a short while (see Figure 8). For faster operation, the commands can also be invoked by drawing a short line in the direction of the pie slice holding the desired command. Furthermore, the menus are context-dependent. The left side of Figure 8 shows the default menu, whereas the right side shows a more specialised menu that is opened when close to the end of a relationship.

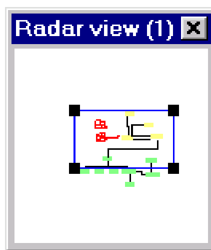


Figure 7. Radar window

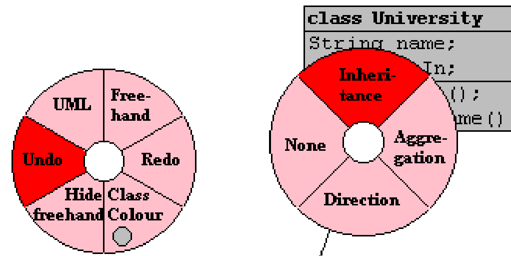


Figure 8. Context-dependent pie menus

2.2.3 The Knight tool and the UML

Our user studies above described how users often used informal, freehand drawing elements, and incomplete elements. Implementing support for these elements in a tool based on the UML is somewhat problematic as the UML metamodel does not contain any freehand elements, and as incomplete elements are not allowed by the constraints in the metamodel. Also we had some problems related to the fact that the UML metamodel does not describe any of the presentational aspects of the UML diagram elements.

During the lifetime of a model, it inevitably becomes gradually more stable and complete. Initially, the developers' knowledge of a domain is limited, and the focus is therefore on understanding the overall structures of the domain. Many issues are unclear at this point, and it is less important to understand the details of the constituent parts, e.g., attributes and methods and their types and parameters. It is thus necessary to do more expressive, but less restrictive, modelling.

We believe that, ideally, the UML should support a continuum of metamodels at different levels of restriction. For each different metamodel, there will be constructs that relate to models ("meta-*model*") and constructs that relate to diagram or presentation ("meta-*presentation*"). An unrestricted meta-model will, e.g., allow attributes with no type information, and an unrestricted meta-presentation will, e.g., allow expressive freehand drawings to be used.

Shifting between levels of restriction

The Knight tool supports *shifting between the levels of expressiveness and restriction*. A user may use the tool in three ways to transform a model element, so that it conforms to a certain restriction level.

- *Automated.* The tool may automatically transform the model element.
- *Guided.* The tool may identify the elements that do not conform to the new level of restriction, and it may provide the user with advice on how each element may be transformed. The user can then decide which transformation is appropriate.

المنارة للاستشارات
www.manaraa.com

- *Manual.* The user has to transform the model element in the usual way, i.e., without any special assistance from the tool.

The Knight tool implements all three kinds of support to varying degree. Strokes are unrestricted elements in a meta-presentation. The default behaviour of Knight is to automatically transform these into UML elements. In this way a restriction is made in the presentation as well as in the model. During integration with other CASE tools, an automated restriction is also necessary. As an example, freehand drawings are converted into comments, and incomplete elements are omitted from the restricted model.

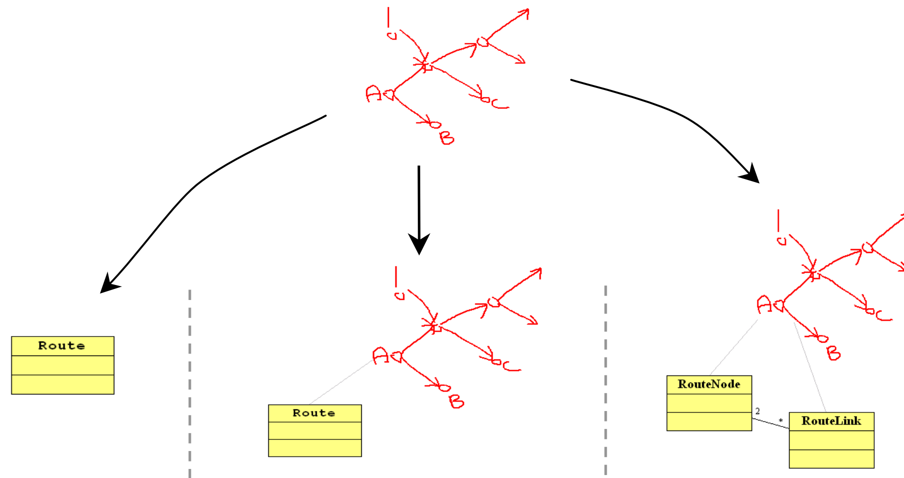


Figure 9. Restrictive transformations of a freehand drawing

The user can also ask Knight to identify the elements that are illegal in the more restricted meta-model. These will be marked graphically, and the user may then request guidance on how to transform the elements. For example, an incomplete association is marked as red, and in order to make it complete, the user can either delete the association or move the dangling end to a class. When the illegal elements are all removed, the whole model has been restricted.

Figure 9 shows examples of manual transformations. In the example, a freehand drawing documenting the concept of a "Route" has been drawn. Depending on the context, the user may choose different transformations. If the drawing is no longer significant, the user may choose to transform the drawing into a class and give it the name "Route". If it is still important to be able to refer to the drawing, the user may transform the drawing to a class and a relation to the drawing, thus using the drawing as an icon for the class. Likewise the user may use the drawing to document a more elaborate model of a route.

Figure 10. From very informal look to formal look

Informal look vs. formal look

The appearance of a diagram influences how people perceive it: A sketchy look of a diagram makes people subconsciously believe that the diagram is not finished, and hence they are more willing to suggest changes to it, and a unfinished look can give developers a cue in remembering which parts they consider unfinished. Knight supports preservation of an informal look of elements, as elements can be displayed in three different ways (Figure 10):

- it may look exactly the way it was drawn, i.e., it is not transformed,
- it may have a semiformal look, which is the same for all elements of the same category, or
- it may have a formal look, i.e., the usual UML notation.

Different parts may also be in different states. In fact, it is often the case that certain parts of a diagram are detailed and finished, while other parts have not yet been

المنارة للاستشارات
www.manaraa.com

elaborated on. Displaying the elements corresponding to whether they are finished or not can show the situation to the users.

2.2.4 Evaluation of the Knight tool

After about half a year into the project, the Knight tool was formally evaluated. This was done by setting up two sessions, with the purpose of testing the Knight tool in a realistic work situation. Both sessions were actual design sessions in which Knight was the primary tool. In the evaluations, a facilitator first introduced the Knight tool to the participating designers and taught them the basic use of the tool. During the session, he also helped if the designers had problems and asked for help. The sessions were videotaped and we took notes with a focus on three aspects of design: cooperation, action, and use. Following the design sessions, we conducted qualitative interviews.

Evaluation 1: Designing a new system using Knight

Research setting. The CPN2000 project (Janecek et al., 1999; `www.daimi.au.dk/CPnets/CPN2000`) is concerned with developing and experimenting with new interaction techniques for a Petri Net editor with a complex graphical user interface. The original user interface is a traditional window-icon-menu-pointer interface, whereas the new interface will use interaction styles such as tool glasses, marking menus, and gestures. As part of the design, three object-oriented models for the handling of events and for the implementation of certain interface elements had been constructed. These three models were integrated into one model using the Knight tool.

Participants. Three designers participated in the meeting. One of these had modelled the event handling and was knowledgeable of the UML and traditional CASE tools. The other two modelled the interaction styles and had little knowledge of the UML.

Results. The resulting diagram is shown in Figure 11. This rather large model was constructed with few problems and mishaps. The criteria for a direct and fluid interaction was met, as the developers were able to use the tool for long periods without help after the short introduction. Also the use of gestures was mostly unproblematic. However, some participants had trouble drawing certain gestures. These gesture have since been improved. With respect to collaboration, the electronic whiteboard worked well. The only problem participants reported was that only one person could draw at a time. Nevertheless, each developer was able to hold his or her own pen, and they all coordinated their actions when necessary. The freehand drawings were widely used and appreciated, although the low resolution of the electronic whiteboard meant that the freehand drawings were relatively coarse-grained. In addition, response from the electronic whiteboards was delayed when a user drew quickly. This meant that freehand text was hard to do both legibly and fast.

Evaluation 2: Restructuring a system using Knight

Research setting. The DESARTE project (`http://desarte.tuwien.ac.at`) is concerned with designing an electronic support environment for architects. As part of this environment, a 3D replacement of the workstation desktop is being implemented. A conceptual model had previously been designed and was to be restructured during this meeting using the Knight tool.

Participants. Two designers attended the meeting: The designer responsible for implementing the 3D desktop and a user involvement expert with an understanding of architectural work practice. Both had a good knowledge of object-oriented modelling.

Results. The second session showed a few more breakdowns and problems than evaluation 1. The developers were nevertheless able to complete the session and their work. When an error did occur, such as the system interpreting a gesture differently than expected, the participants sometimes got confused about what was happening: The feedback of the tool was not sufficient in the event of misinterpretations. This has since been improved. Furthermore, one of the participants initially had many problems

www.manaraa.com

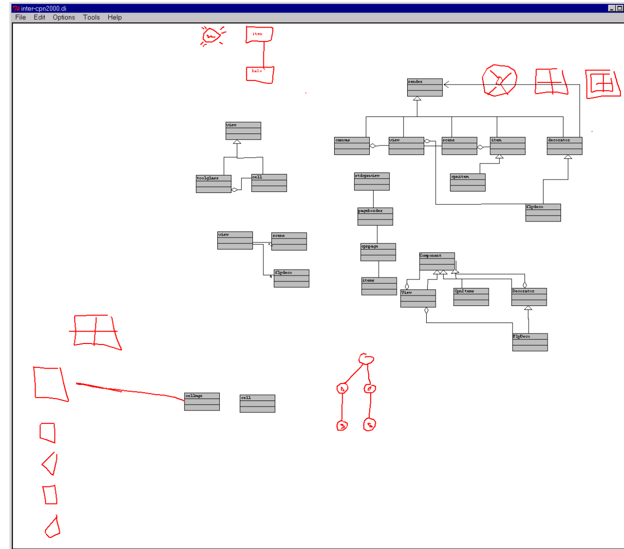


Figure 11. Diagram produced in the first session

operating the marking menu: Often he invoked commands by accident when drawing. This was partly due the fact that he had no previous knowledge of gestural input and marking menus, and partly due to a programming error that has now been corrected. Collaboration was as in the first evaluation facilitated by the large screen and the design for the tool. Also, the evaluators often made freehand drawings to illustrate the user interface of the designed environment. A minor problem in this case, was that that informal and formal elements could only be rudimentary connected, and there was little support for advanced grouping.

Evaluation conclusion

As a conclusion on the evaluation, the observations and subsequent interviews showed that the Knight tool is a valuable tool for modelling in practice. A number of improvements have been made since the evaluations, and more are planned. Specifically, we plan to:

- Further expand the support for informal and incomplete elements
- Fine-tune the gesture recognition
- Couple the two different modes to distinct parts of the screen as the concept of *segments* in Flatland (Mynatt *et al.*, 1999)
- Investigate the use of alternative input devices, e.g. Mimio (http://`www.virtualink.com`)
- Investigate the notion of *filtering* of elements in the diagram

2.3 Object models and relational databases

In a running system, objects representing entities in the referent system will be instantiated from the object model. These instances comprise the data being manipulated by the user of the running system. Much of that data often needs to be saved, or persisted, before the running system terminates. There are several ways of doing this. The simplest is to use a language supplied persistence mechanism, a persistent store or an object-oriented database, as such storage mechanisms offer functionality for persisting the objects directly. Often though, the system will be running in an environment where there already are a number of existing databases, and often there is a need to persist the data in, or to read existing data from, these databases.

المنارة للاستشارات
www.manaraa.com

This situation is often complicated by the fact that most existing databases are *relational databases*. These databases are based on different abstraction mechanisms than the object-oriented paradigm, which results in an so-called *impedance mismatch* problem: To read data from the database the data in the relational model needs to be mapped to the object-oriented model, and vice versa to save data. Implementing support for this is not trivial. This motivates the creation of a *generic framework* that can be used by many applications. Some ideas for such a framework are presented in this section.

2.3.1 Mapping object-models to relational schema

The first problem to be tackled is the mapping of data between the two models. This topic is discussed in this subsection, before we discuss the actual framework in the next subsection. The presentation is based on (Thomsen, 1998a, 1998b).

In discussing the mapping we will use the example of a simple model of a university (Figure 12). In the following, we will discuss how to map classes, generalization hierarchies, and association and aggregation relationships.

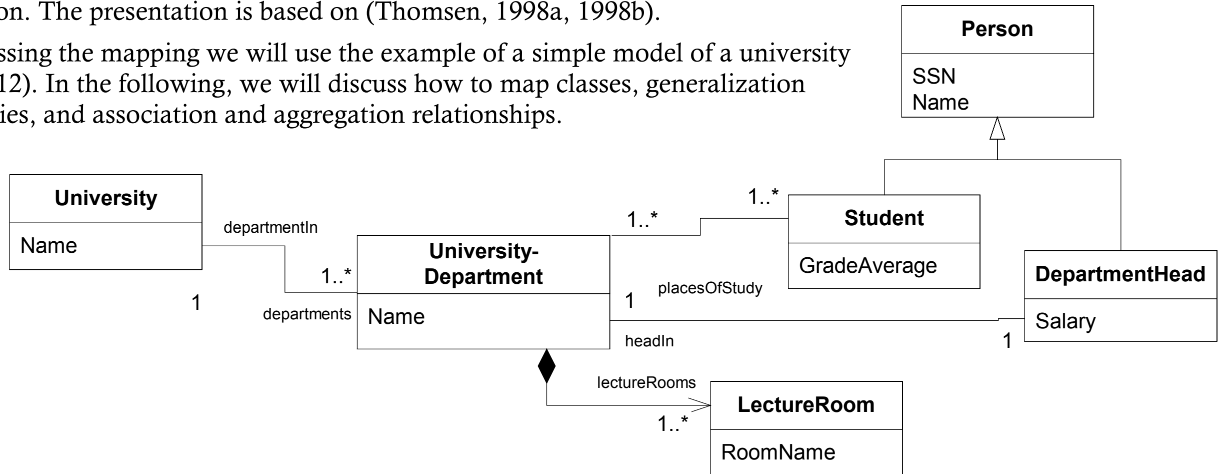


Figure 12. The university model

The class: The main choice when mapping a class is whether there should be one table per class or several tables per class. As a general rule Blaha *et al.* (1991) write that one should construct one table per class, but this is not always the case. If the mapping to one table violates third normal form (Codd, 1970), then the table will have to be split up vertically into two or more tables. If performance aspects are critical and if the number of records is large, then the table can by split up horizontally into a number of tables representing groups of tuples. The resulting University table – which conforms to third normal form – is:

Entity Name	Required?	Type
UniversityId	Yes	Integer
Name	Yes	Text 100
Primary key: UniversityId		

Generalisation hierarchies: Just as with classes there are several possibilities when mapping superclasses and subclasses. Blaha *et al.* (*ibid*) present three possibilities, which all have advantages and disadvantages depending on the actual hierarchy. The mappings we describe here only take single inheritance into consideration.

Separate tables mapping. This mapping maps each superclass or subclass to a separate table. This mapping is performed using the standard mapping for classes, with two modifications: The primary key in the topmost superclass is also used as a primary key in the tables representing the subclasses, and the table representing the topmost class gets an additional attribute denoting which subclass is being represented. Using this mapping on the generalisation in our example model, we get the following tables for `Person` and `Student` respectively:

Entity Name	Required?	Type
SSN	Yes	Text 10
Name	Yes	Text 100
PersonType	Yes	Text 20
Primary key: SSN		

Entity Name	Required?	Type
SSN	Yes	Text 10
GradeAverage	No	Real
Primary key: SSN		

المنارة للاستشارات
www.manaraa.com

This mapping is useful when there are many subclasses on each level, but it can result in a too large number of tables if the super/sub-class hierarchy is tall. Furthermore this mapping can be expensive when the attributes of subclasses have to be fetched, as several tables have to be consulted/joined.

Leaf-classes only mapping: Where the previous mapping had one table per class, this mapping results in fewer tables. This is achieved by only creating tables for the subclasses that are at the very bottom of the hierarchy. These tables receive all the attributes of the bottom class, but besides this also receive all the attributes from any class met going up the tree from the bottom class to the topmost superclass. The result for the `Student` and `DepartmentHead` tables becomes:

This mapping is useful when the superclass has few attributes and the subclass has many attributes, as the impact of putting the superclass attributes in every subclass table will not be very large. It can introduce inconsistencies though, as uniqueness of fields cannot be checked across tables. An example of this could be a database having both a student and a professor with the same SSN, in each of their tables. Furthermore any class not being the bottom subclass will be mapped to a table that is 'too large'. This can result in a waste of space in some databases, and it might reduce the efficiency.

Single table mapping: Where the philosophy of the second mapping was to bring attributes from superclasses "down" into the subclass level, the third mapping does the opposite. Instead of a number of tables for each subclass, we construct only one table having all the attributes of any class in the super/sub-class tree. This results in the following table:

Entity Name	Required?	Type
SSN	Yes	Text 10
PersonType	Yes	Text 20
Name	Yes	Text 100
GradeAverage	No	Real
Salary	No	Integer
		Primary key: SSN

This mapping is useful when the hierarchy is not too wide. If the hierarchy furthermore is high, the lookup of attributes in subclasses low in the hierarchy is much faster than with the first mapping. The disadvantage is of course that the table can become quite large. This can result in wasted space, in databases that do not optimise the space used by "half-empty" records.

Association relationship: When mapping associations between classes, we can choose between introducing a new table mapping the association, or just introduce foreign key attributes that refer to the classes that are associated. Which strategy to choose depends on the multiplicity of the association.

Many-to-many associations: When mapping many-to-many associations, we have to introduce a new table to conform to the rules of third normal form. Take as an example the association in the above example model between the classes `UniversityDepartment` and `Student`. If we mapped this association by adding an attribute in the `UniversityDepartment` and Student tables, which would refer to the primary key of the associated table, then our primary key would no longer be valid. This is easily seen as we could now have several records having only different values in the new association field, and thereby having the same value in the `DeptId` attribute, which is our primary key.

Making the association attribute part of the primary key could of course solve this problem, but at the same time produces a new problem. We will now violate second normal form, as some of the non-primary attributes no longer will depend on the full primary key. We therefore always use a new table when mapping many-to-many associations. This table simply contains the attributes forming the two primary keys of

المنارة للاستشارات
www.manaraa.com

the two associated tables. These are all required attributes, as associations only can exist between existing objects. For the primary key we use the joint key of both attributes, as we can have records with the same values in either one of the two attributes. The result of mapping the association between the `Univerisity-Department` and `Student` tables results in the following association table:

Entity Name	Required?	Type
DeptId	Yes	Integer
SSN	Yes	Text 10
		Primary key: (DeptId, SSN)

Many-to-one associations: The mapping of many-to-one associations can of course be done using the technique for many-to-many tables discussed above, but this is no longer the only alternative. Since one of the association ends has a one multiplicity, we can represent the reference to this class by adding an attribute containing its primary key to the table with the many multiplicity. This respects third normal form, and does not destroy the primary key as we never add multiple entries in the many table with only different values in the association field – this is easily seen as each distinct record in the many table is associated to only one other class. In our example model the association between `University` and `University Department` results in the following table:

Entity Name	Required?	Type
DeptId	Yes	Integer
Name	Yes	Text 100
UniversityId	Yes	Integer
		Primary key: DeptId

The advantage of this mapping compared to the mapping of many-to-many with an extra table is that access to the data becomes faster as only two tables need to be consulted and that the overall number of tables is lower. There are also disadvantages though. The introduction of the association attribute in one table moves the association into one of the classes where it conceptually belongs in neither but as a separate entity. Furthermore practice has shown that it is difficult to get association multiplicities correct on the first model iterations which speaks in favour of introducing a separate association table so changes to many-to-many are trivial. We therefore recommend adding the association table when very fast access is not critical.

One-to-one associations: One-to-one associations can of course be mapped in the same manner as with many-to-many and one-to-many associations, but besides this they can be mapped in a third way. Since there will only be one object on both sides of the association, we can collapse not only the association table into one of the object tables, but collapse both of the associated classes and the association table into one table. This mapping used on the association between university department and department head results in the table shown below.

Entity Name	Required?	Type
DeptId	Yes	Integer
Name	Yes	Text 100
SSN	Yes	Text 10
Salary	No	Integer
		Primary key: DeptId

One has to careful though, as this transformation will often violate third normal form. The advantages in this further collapse of tables are the same as before: Faster lookup and lower overall number of tables, and the disadvantages are also the same: It is "conceptually wrong" and a change of multiplicity becomes difficult. Furthermore this

المنارة للاستشارات
www.manaraa.com

mapping cannot handle cyclic associations of the form where A associates B, B associates C and C associates A. We therefore again recommend that one-to-one associations are transformed with the extra associations table as with many-to-many associations.

Aggregation: In an object model there is typically a distinction between association and aggregation, but when mapping these two constructs the same methods apply. In Blaha et al, all that is said about aggregations is that they can be mapped in the same ways as associations. This is the case as aggregation can be viewed as a special case of one-to-many association. It should probably be investigated further is this really is the case in reality, or if it only holds in theory, and special care has to be employed when mapping these.

2.3.2 A framework for coupling object models and relational databases

The previous section described the possibilities that exist for mapping between relational database schema and object models. At an operational level, the framework offers the following operations:

Update: Takes as input a single object currently in the database. The data in the database corresponding to the object will be updated according to the values in the object.

Create: Takes as input a single object currently not in the database The object is added to the database.

Delete: Takes as input a single object that has previously been fetched from the database, and deletes the data in the database corresponding to the object.

Fetch: Takes as input a class and returns a list of all those instances of that class that are currently found in the database. To restrict the number of returned objects search criteria can be added.

ReFetch: Takes as input a single object that has previously been fetched from the database. It then compares the values in the object with the values in the database, and if the values in the database have been changed they will be fetched and the values in the object changed.

Furthermore, the design of the framework has been based on the criteria:

Orthogonality. The framework operations should be orthogonal to the type of the objects operated upon, i.e. it should be able to handle all types of objects, and it should not make any requirement on the objects types, eg. that they should be a subtypes of rdbObject.

High transparency. The framework should be designed with a high level of transparency, in the sense that only the most important database operations on the objects should be explicit, whereas other less important operations should be made transparent to the programmer.

Ease of use. The framework should offer abstractions as strong as possible, in the sense that as little code as possible should have to be provided by the user of the framework.

Flexibility. The framework should only enforce a specific architecture regarding persistence aspects. The user should be able to choose how to organise other parts of the architecture, e.g. user interface – object model communication.

To ensure that the first criterion is meet is has been chosen to create a *shadow class* for each persistent class. This shadow class then handles the persistence of that class. Other choices, such as requiring that a class to persist should inherit from a certain

المنارة للاستشارات
www.manaraa.com

persistence class would clearly violate the first criterion, and lead to a framework that would be less general. Figure 13 (Left) shows the generic interface for the shadow classes. To create a concrete shadow class for any given class, the user of the framework must create a subclass of the generic shadow class that implement three abstract methods: `init`, `obj2rdb`, and `rdb2obj`.

The `init` method should be specialised to provide the framework with the name of the table and the names of the columns that instances of the class are to be persisted in. The two methods `obj2rdb` and `rdb2obj` should be specialised to save the state of the class using a number of provided `setInt`, `setChar`, `setText`, … and `getInt`, `getChar`, `getText`, … methods. As the user thus has to add very little code to the framework, the third criterion is also meet.

Once the `obj2rdb` and `rdb2obj` methods have been specialised, the uer of the framework can use the shadow classes to persist his classes as illustrated in Figure 13 (right). The class of the name `customerRdbClass` is the shadow class for the customer class, and in the example an instance of the customer class is persisted simply by handing the instance to the save method on the shadow class.

The second criterion – transparency – is met by the fact that the user only has to implement the two persistence methods using the `setX` and `getX` methods. Other database specific functionality is taken case of by the framework. Finally the fact that the framework is based on shadow classes that can handle the persistence of any type class and the fact that it imposes no restrictions on the architecture of applications using the framework ensures that the fourth criterion is also met.

```
rdbClass:
(#
  classType:< object;
  init:< (# … #);
  obj2rdb:< (# … #);
  rdb2obj:< (# … #);
  create:
  (# theObject: ^ classType
     enter theObject[]
     do …
  #)
  save: (# … do … #)
  delete: (# … do … #)
  refetch: (# … do … #)
  fetchAll:
  (# currentObject: ^classType
     do … inner #)
  #)
#)
```

```
(#
   customer: (# … #);
   aCustomer: ^customer;

   customerRdbClass: rdbClass
   (#
      classType:: customer;
      …
   #);
   custRdb: @customerRdbClass;

do
   &customer[]->aCustomer[];
   …
   aCustomer[]->custRdb.save;
   …
#)
```

Figure 13. (Left) Simplified interface class – (Right) Simple use

2.4 Object models and user interfaces

In object-oriented development of *interactive applications* one major task, besides the creation of the object model, is the design and implementation of the *user interface*. Traditionally development methods offer much advice on how to construct each of these major parts, but there seems to be less advice offered with respect to the benefits of a joint production of the two parts.

Modelling is native to object-oriented development. During development, models are continually created, modified, and extended. An important such model is the object model that describes the most important concepts in the *problem domain*. Clearly, a lot of knowledge of the problem domain lies behind the production of these models, so it seems natural to investigate how much of this knowledge can be used in producing the user interface.

المنارة للاستشارات
www.manaraa.com

As the user interface is what provides the *end-user* with a means of interacting with the application it is crucial that it is well designed. An important criterion in achieving this is understandability – the user interface must be understandable to the user in terms of the work context she/he works in and it must support the praxis of hers/his use domain. As the models constructed in object-oriented development are descriptions of aspects of the problem domain it also seems natural so investigate if knowledge about the user interface can be used as input in constructing these models.

2.4.1 A systematic look: The Dragon case

We now present and discuss the findings in comparing the user interface and the object model of the Dragon Project. The Dragon project provides an interesting case for investigating the relationship between object models and user interfaces for a number of reasons:

- The object model found even in the running system is very close to a "clean" domain model as the project was a prototyping project where collection of knowledge of the problem domain and ways of supporting its work processes was more important than performance issues,
- the problem domain of shipping is quite complicated or at least not trivial, and
- both the domain model and the user interface were discussed and produced in cooperation with users from the problem domain.

We first look at simple and complex attributes and widgets, then at grouping of widgets in the user interface and how this relates to grouping in the object model, and finally, at some of the mismatches found and their possible explanations and at what matches could be imagined other than the ones found here.

Simple attributes and simple widgets: Simple attributes such as text, integer, and date are in many cases represented by a simple textfield widget (see Figure 14 below). This widget both indicates the current state, and allows the user to change the state. Often the textfield widget is accompanied by a text label, placed next to the textfield, which provides the user with a description of which attribute the textfield belongs to. In some cases – particular when representing dates – the textfield performs a syntax check when the user inputs a new text. This serves both as a validation of the input, and allows the user to input dates in various ways which are then formatted uniformly by the system. In some cases where the user was not allowed to change the contents of the widget one of two alternatives was found: Either a special type of 'read-only' textfield widget or a plain text-label.

Name: Heinz Thompson
Phone: 127 3723 233

Booking No:
BIO001889

Figure 14. Representing simple attributes

Simple attributes that can only hold one of a number of predefined values are represented in four different ways. One is an editable textfield (Figure 15a below) and a second is a option button (Figure 15b). The option button is generally used in the cases in which there are only a few alternatives, whereas the textfield is used when there are many possible valid values. To validate whether a correct input has been made, the field performs a syntax check on the input as in the case of dates above. In a few cases, a combination of the two input options was used (Figure 15c): An editable text field with a popup menu from which the legal options can be chosen. Finally in those cases in which there are only very few alternatives, and where all these need to be visible at the same time a group of radiobuttons has been chosen (Figure 15d).

المنارة للاستشارات
www.manaraa.com

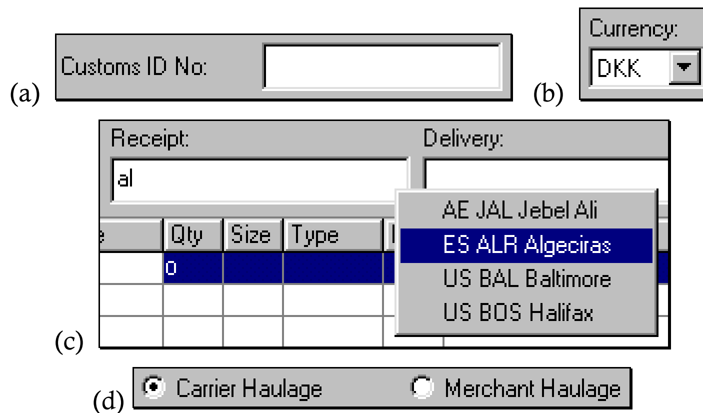


Figure 15 (a, b, c, d). Representing type attributes

Complex attributes and complex widgets: In many cases in the domain model, a class has an attribute that is a reference to another class. As examples, the class `Quote` holds a reference to the class `Customer`, and a reference to the class `Standard Product`. In the user interface of the Dragon system these relationships between classes are represented by a number of widgets that present a few of the most important attributes of the related concept, e.g., the `Quote` tab has one text field holding the `customer name` of the related `Customer`, and two text fields showing `receipt` and `delivery` of the related `Standard Product`. These widgets do not always show *all* information about the related concept as this often would be too space consuming. In the two example cases, if the user wishes more information than what is shown in the few widgets, he will have to go to another tab.

Besides holding single references to other classes, domain classes also often hold collections/lists of references. An example is the `Sea Corridor` class which holds a list of references to related `Sea links`. In almost all cases in the prototype these many related elements were represented by a list-/table-type widget (see Figure 16 below). Furthermore, the columns in this list view closely corresponded to each of the attributes in the related class.

Base port to base port

From	To	Transport	Days	Service
HK HKG Hong Kon	JP NGO Nagoya	MVS	5	AE1
JP NGO Nagoya	JP TYO Tokyo	FEO	2	

Figure 16. Representing many related elements

Grouping of widgets: Another important feature in user interfaces is grouping. If the user interface is to be understandable it is important that the elements in the user interface are grouped in a natural way. This section looks at how groupings found in the user interface correspond to groupings in the object model.

The highest level of grouping in the user interface it the overall division of the user interface into 13 major tabs. Seven of the 13 major tabs (Customer, Quote, Booking/Transport, Documentation, Products, Schedule and Vessel) each correspond to one important class in the domain model. For all seven tabs almost all data in the tab is found in this one corresponding class. Also each of these tabs/classes corresponds to the seven probably most important concepts in the problem domain. The other six tabs (Stuffing, Doc. status, Allocation, Rerouting, Report and Route

المنارة للاستشارات
www.manaraa.com

Map) correspond to neither one class nor one concept. They rather correspond to one specific, important task that operates on a number of classes.

Inside each tab elements are further grouped in subtabs and inside canvases. Figure 17 below illustrates an example from the Booking tab: A separate canvas labelled `Contact` holds three widgets labelled `Party`, `Name`, and `Phone`. In the `Booking` tab this `Contact` canvas displays the contact person that has been assigned to the current booking. Interestingly this Contact canvas corresponds directly to the `the_Contact_Person` attribute on the `Booking` class. This attribute is a reference to the class `contactPerson`, which has three attributes that correspond to the three widgets `Party`, `Name`, and `Phone` in the canvas.

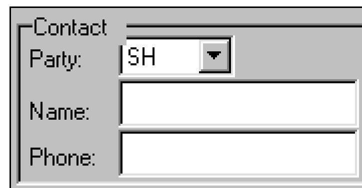


Figure 17. Grouping using a canvas

Mismatches: A small amount of classes (around 10 %) are not shown anywhere in the user interface at all. These classes hold e.g. configuration properties or they hold duplicates of data found elsewhere that has been computed to increase performance. In general, one could question whether such classes should be in the domain model at all.

Also, as demonstrated by the introductory example in the previous section, there are many cases where one of the attributes in a class is not represented anywhere in the user interface and many cases where a widget which has no corresponding attribute where its state is stored. These omissions were often not deliberate but were rather either due to an error or due to different insights gained by the different people working on the user interface and object model.

Other matches: This section has presented a number of matches or relations between widgets in the user interface and classes and attributes in the object model of a concrete computer application. Without resorting to naive inductivism, we do believe that many of these are valid in a broader context. It would be interesting to see whether these observations also hold for other applications and other domains and to investigate whether some more concrete statements can be formulated on the relationship. It would also be interesting to see how much more information about the user interface could be found if the combination of a domain model and a task/use-case model was used.

In addition, there are of course many other possible and likely matches other than those exposed by this one case. We especially expect this to be true in applications with more advanced user interfaces such as drawing tools or programs with graphical visualization in the forms of e.g. charts or graphs. However, even applications such as these will most likely have parts that are more traditional where most of the observations will be valid.

2.4.2 Implications for development and future work

The case study suggests that domain models are useful input in designing the user interface and that user interfaces can provide input in the development of the domain model. We doubt however that we should pursue ways of fully specifying (or generating) user interfaces from domain models. Many important choices in user interface development are based on other factors than those captured by these models, e.g. factors such as ergonomics, understandability, performance, social conventions, user expertise etc. Rather we should consider the production of the domain model, the task/use-case models, and the user interface to be separate but parallel activities. Each activity can provide very useful input to each of the other activities and the

المنارة للاستشارات
www.manaraa.com

development can in this way achieve synergy – a synergy that we indeed often experienced in the Dragon project.

There are several ways of achieving this synergy. One way would be to perform simple manual comparisons of the user interface and the object model at regular intervals in the development. It would be interesting, however, to investigate if tool support for this comparison could be usable. The tool could help both in creating the object model starting from the user interface and the reverse. A complete generation would probably not be desirable – rather the user could select parts of the user interface or the object model and then ask the tool to produce the corresponding part. To make the tool really usable these generations should respect the software architecture within the application, and would then also potentially help in keeping the code uniform. Also the tool could assist the developer in respecting user interface guidelines. We are currently looking at ways of designing such a tool, but have not yet had the time to implement a prototype.

2.5 Section summary

This section has outlined my results with respect to object-oriented development. After a brief description of the notion of modelling, and a few comments on the actual creation of models as an collaborative effort, three areas were dealt with: the Knight tool that supports the construction of object models, persistence of object models in relational databases, and the relationship between object models and user interfaces.

www.manaraa.com

3. Software Architecture

In the recent years the topic of software architecture has attracted much attention. When discussing or designing software at the architectural level, the focus is on higher level abstractions as opposed to lower level abstractions such as code. Also software architecture considers non-functional aspects such as time and space usage. This higher abstraction level should enable software designers to handle the large complexity of complex software applications, and a focus on software architecture can thus be seen as a means of leveraging software quality.

A commonly agreed upon formal definition of software architecture does not exist[1] – we have chosen to use the often-used definition by Bass et al. (1998):

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*

In designing software, developers often lack a full understanding of the solution to a specific problem. This can e.g. be due to a lack of knowledge of a particular solution or a particular technology, but it can also be due to a lack of understanding of the problem itself. As the design of software architecture is the highest level of design in software development, architectural uncertainty causes significant risks. This has caused us to define the concept of *architectural uncertainty* (Hansen & Thomsen, 1999):

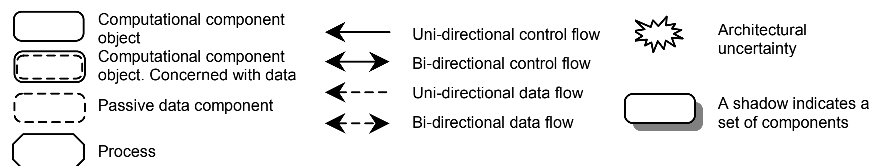
> *An architectural uncertainty represents a lack of full understanding of an architectural structure.*

To avoid failure in creating successful software, special attention should be paid to these architectural uncertainties: They should be identified, and should either be solved or somehow isolated. This will be discussed in more depth in section 3.2 with a concrete example of an architectural uncertainty.

Another interesting area is the integration of software architecture with the development process. In the context of the Dragon Project, we have tried to gain an understanding of this, which will be discussed in section 3.1.1. Finally, section 3.3 will discuss a concrete software architecture, namely the software architecture that we designed in the Knight project to facilitate the integration of the Knight tool with existing CASE-tools.

In the presentations and discussions of concrete software architectures in the following sections, we will try to present the architectures using well-defined software architecture notations, such as the UML or in most cases a variation of the architecture specific notation introduced in (Bass *et al.*, 1998). In the Bass *et al.* notation (Figure 18), solid lines denote control and processing, whereas dashed lines denote data. We furthermore add the starred symbol to denote an architectural uncertainty, and add a shadow to a component to indicate one or more components of that kind.

Figure 18. Software architecture notation



[1] Although there a commonly agreed upon definition does not exist, a large number of definitions do exist; the SEI Software Architecture homepage at `www.sei.cmu.edu/architecture/definitions.html` lists more than 50!

www.manaraa.com

3.1.1 Software architecture and the development process

Iterative development is becoming the norm in most object-oriented development processes. However, development using an iterative approaches, lead to many new problems, as there is no firm foundation to base the initial development on. This results in a special challenge with respect to software architecture: the software architecture needs to be designed early in the process so that it can provide a frame for the development, but often the concrete requirements, and the needed technical insights, are not available early in the process.

In the Dragon project we tried to meet these challenges by designing an initial software architecture as early as possible to support the initial prototyping phase. This architecture then later evolved though a number of *architectural refactorings* into the final architecture. Our experiences are presented briefly below –more details and concrete examples are available in (Christensen *et al.*, 1999a).

Initial prototyping

We believe that an *explicit architecture* is essential – even in initial prototyping cycles. In the Dragon project our initial architecture was designed to meet the following criteria:

- the architecture had to offer a fairly stable structure, in which the prototype could evolve during the first phase,
- the structure had to be flexible enough to allow for a high degree of experimentation within rapid development cycles, and
- it should support an efficient work organisation, allowing all developers to work intensively on the prototype in parallel.

The desire for a stable structure, in which the prototype can evolve, can be seen as a way of reducing the complexity of the system. Also, an agreed upon software architecture provides an overview of a quickly growing prototype, it constitutes a consensus about "how to do things", and it serves as a vehicle for communication and explanation. This was e.g. experienced when a new developer was introduced to the prototype late in the experimental prototyping phase: as a result of the well-defined architecture, he obtained an understanding of the relatively large prototype quite easily.

The architecture must be flexible though: it should allow evolutionary development of and experimentation with all parts of the prototype to facilitate the experimental work performed within participatory design sessions. Finally, the criteria for efficient work organisation, is important in order not to slow down the development.

Architectural evolution

While an explicit architecture is essential, we also believe that it rarely is possible to design the final architecture of the system during the initial phase of the development. We will, instead, argue that *architectural evolution* is necessary. This can be due to changing requirements over time, due to increasing understanding of the problem domain, and due to further understanding of the technical ways of realising the system. In this way, not only the system itself but also the software architecture can be said to be prototyped.

Such architectural evolution can take place though a number of architectural refactorings: function-preserving transformations of the architectural structures, akin to code refactoring that are considered with semantic-preserving transformations of objects and classes (Opdyke, 1992).

المنارة للاستشارات
www.manaraa.com

3.2 Architectural patterns for user interface architectures

The introduction to this section motivated and introduced the concept of *architectural uncertainty*. This subsection will present one example of such an uncertainty and discuss how this uncertainty was handled. The example is taken from a joint paper with Klaus Marius Hansen (Hansen & Thomsen, 1999). For more details and for another example the reader is referred to the paper.

3.2.1 The Dragon case

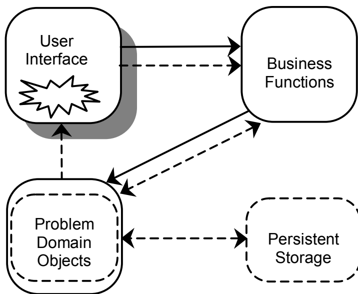


Figure 19. Conceptual view of the initial architecture in the Dragon project

This example of an architectural uncertainty is also taken from the Dragon Project. One consequence of the large degree of cooperative design carried out in the project was that the user interface was constantly extended, re-designed and changed. This was somewhat in opposition to the other parts of the Dragon System, as e.g. the Problem Domain Model and Persistent Storage components, which were at least structurally stable after the initial phase. From a software architecture point of view the radically changing User Interface component was thus an architectural uncertainty (illustrated in Figure 19).

3.2.2 Architectural patterns

The concrete software architecture 'challenge' in the Dragon Project, where the user interface is much more unstable than the rest of the application is a common problem faced in constructing interactive systems. Recognising this, and to allow the designed architecture to be used in other contexts, our paper presented both the problem and the solution in the form of an *architectural pattern*. Architectural patterns, like design patterns (Gamma *et al.*, 1995), present a common solution to a common problem within a given context, which matches well with the notion of architectural uncertainty. Architectural patterns are not new. Well-known architectural patterns include those presented by Buschmann et al. (1996), and Shaw (1996).

The pattern format we use varies somewhat from the format of Buschmann et al. (1996) and Shaw (1996). The former can be too verbose to efficiently communicate the essence of the patterns, and the latter can be too short to give sufficient understanding to actually apply the patterns. Figure 20 presents the pattern format, which may be viewed as a condensed version of the format in Gamma et al. (1995) or as a slight variation on Brown et al.'s (1996) 'deductive mini-pattern' template.

> **Name and thumbnail**: What should the pattern be commonly known as? Followed by a short description.
>
> **Problem**: What is the architectural problem that the pattern faces, and what are the main forces behind this problem?
>
> **Solution**: What is the effective solution of the stated problem? This section includes a description of the pattern's high-level static structure in the form of a Unified Modeling Language class diagram using packages.
>
> **Sample implementation**: How could this pattern be applied? This section includes UML class diagrams.
>
> **Consequences**: What are the benefits and liabilities of applying this pattern?

Figure 20. Pattern format

Using this format, we now present the Application Moderator pattern, which in a generalised manner describes the problem faced in the Dragon Project, and its general solution. Following this we return to the concrete application of this general solution in the Dragon case.

www.manaraa.com

3.2.3 The Application Moderator pattern

The Application Moderator architectural pattern divides an interactive application into a problem domain related part (the Problem Domain Model), a user interface component (User Interface), an abstract interface to the user interface (User Interface Mirror), and an Application Moderator component that couples the User Interface Mirror with the Problem Domain Model and other functionality.

Problem: How does one design the architecture of an interactive system such that the user interface functionality is separated from the problem domain related functionality and such that changes to the user interface require minimal change to the rest of the system?

The following forces should be balanced:

- User interface functionality should be separated from problem domain related functionality, so that each part is easier to understand and maintain.
- Changes to the user interface should have as little impact on the rest of the application as possible.

Solution: The Application Moderator pattern divides the application into five components, Problem Domain Model, Application Moderator, User Interface Mirror, and User Interface. Optionally a Testing component may be implemented. The overall structure is shown in Figure 21.

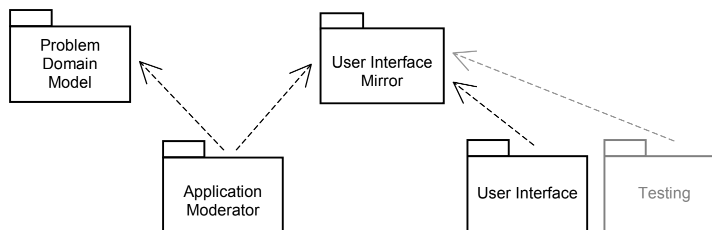


Figure 21. Overall structure of the Application Moderator pattern

The Problem Domain Model contains problem domain related data and functionality. The User Interface Mirror contains an abstract interface to the User Interface component. It consists of data members that reflect the state of the widgets in the user interface and event members that represent events that occur in the user interface, such as a button press. Furthermore, it contains two abstract methods, `setState` and `getState`, which should be refined to write the state of the concrete widgets into the data members (`getState`) and conversely (`setState`). The User Interface contains the concrete widgets, implements the `getState` and `setState` methods as described above, and calls event members as appropriate.

The Application Moderator connects the Problem Domain Model with the User Interface by subscribing to the events in the User Interface Mirror and thus moderates their communication. Upon invocation of the events, it can access the current state of the user interface by calling `getState` and then read the data members or it can set the state of the user interface by setting the data members and then call `setState` or it can do a combination of both.

Finally, the architecture allows for effective testing of most of the application. This is done be creating a Testing component that systematically sets the state of the User Interface Mirror, calls one or more event methods and then tests if the data members are in a correct state.

Sample implementation: Consider a small, and perhaps somewhat artificial, *Financial History* application for tracking financial expenses, with user interface as shown in Figure 22 (Left) and problem domain model as shown in Figure 22 (Right). An application of the Application Moderator pattern towards structuring this application may proceed as follows; the steps should not necessarily be taken sequentially. Part of the class structure of a resulting implementation is illustrated in Figure 23.

www.manaraa.com

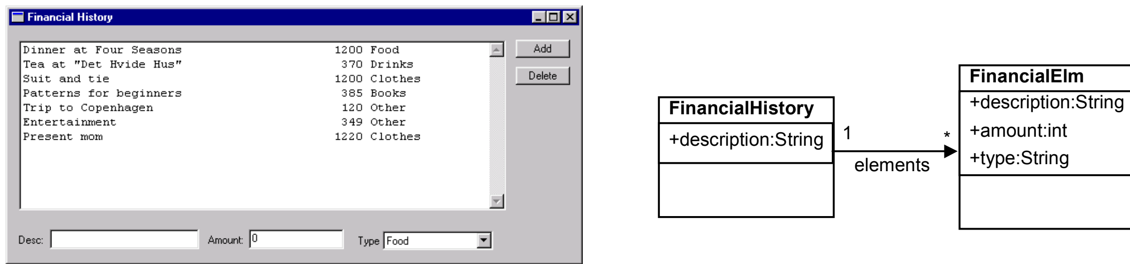


Figure 22. (Left) Financial History user interface
(Right) Financial History problem domain model

1. Implement the Problem Domain Model

The Problem Domain Model is as described above.

2. Implement the User Interface

The User Interface component (`InputUI`) can be implemented using a user interface toolkit and corresponds to what may be generated by a user interface builder.

3. Implement the User Interface Mirror

The User Interface Mirror reflects the User Interface and acts as an interface to it. The User Interface Mirror is in this sample implementation divided into two classes: `UIDataMirror` and `UIEventMirror`. The `UIDataMirror` provides a data interface (Figure 23, left) that reflects the data displayed in the user interface. The description and amount attributes correspond to the two text fields in the user interface of the Financial History application. The expenses attribute is a list of `FinanancialExpense` objects (Figure 23, right). Each element of this list corresponds to an element in the list view of the application. `CurrentExpense` models the current selection of the list view. Moreover, the `getState` and `setState` operations are defined in the class. The event members of the User Interface Mirror are implemented as a number of abstract methods on the `UIEventMirror`, with one function for each event of interest in the user interface. The `onAdd` method, e.g., corresponds to the event that the `addButton` was pressed.

4. Refine the User Interface component

The User Interface component, implemented in step 2, is refined to implement the `setState` and `getState` operations and to call the event methods in the User

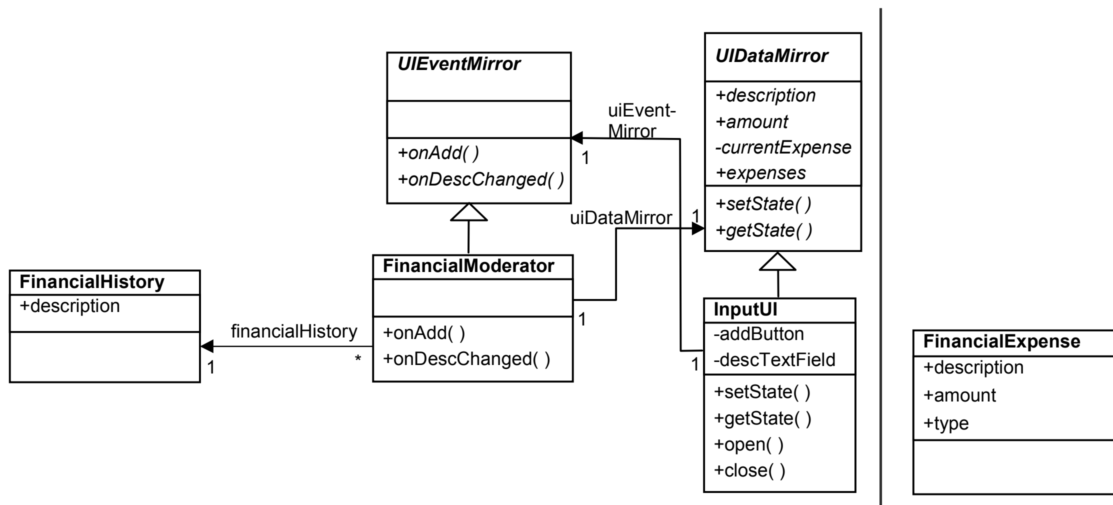


Figure 23. (Left) Class diagram for the Financial History application
(Right) `FinancialExpence` data interface

المنارة للاستشارات
www.manaraa.com

Interface Mirror. In this case the User Interface component binds the button-press events of the buttons to call the corresponding event members in the `UIEventMirror` associated through its `uiEventMirror` association.

5. Implement the Application Moderator

The Application Moderator connects the User Interface Mirror and the Problem Domain Model. In our example, the `FinancialModerator` binds the events in the user interface by implementing the event members in the `UIEventMirror` interface and reading and writing to the data members in the `UIDataMirror`. Generally, the implementation of the Application Moderator should:

- Add methods, which map data between the Problem Domain Model and the data members in the User Interface Mirror,
- Refine the abstract methods of the User Interface Mirror that represent events of interest,
- Implement the general application functionality inside the appropriate event methods, and
- Implement multiple view consistency if needed.

6. Optionally implement the Testing component

An optional Testing component can be implemented by creating another specialisation of the User Interface Mirror that systematically calls the event members (e.g. in `UIEventMirror`) and then tests the state of the data members (e.g. in `UIDataMirror`).

Consequences: This architectural pattern has both benefits and liabilities.

Benefits:

- The User Interface component and the User Interface Mirror components are independent of the Problem Domain Model component.
- The Problem Domain Model component is independent of the User Interface component.
- The architecture supports effective testing. Regression testing of large parts of the application, e.g., can be done efficiently via a Testing component.

Liabilities:

- The interface to the User Interface represented by the User Interface Mirror component takes some time to develop and maintain.
- The architecture results in a minor overhead in terms of function calls and data conversion.

3.2.4 Applying the Application Moderator pattern

As described earlier, the frequent changes to the user interface throughout the Dragon project constituted a major architectural uncertainty. Figure 24 shows how the Application Moderator pattern addresses this by isolating user interface changes. The

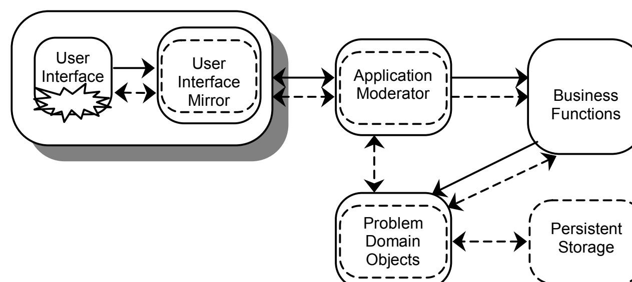


Figure 24. Isolating user interface changes in Dragon

المنارة للاستشارات
www.manaraa.com

Business Function, Problem Domain Objects, and User Interface components are now no longer communicating directly. Instead, all communication with the user interface now goes through the User Interface Mirror component. This means that if changes to the user interface do not require changes to User Interface Mirror, the rest of the application will not be affected by the changes.

In our case, changes to the user interface often required none or only small changes to the rest of the application. For example, several versions of a set of allocation user interfaces resulting from regional differences existed over time using the same User Interface Mirror. The choice of architecture was helpful: Although the extra interfaces were somewhat problematic to introduce in the context of rapid development, it more than paid for the extra work in terms of increased stability.

3.3 Tool integration and software architecture

The previous section presented one concrete architecture from the Dragon Project and discussed architectural uncertainty in that case. In this section we look at another concrete architecture – namely the architecture used in the Knight tool to integrate with existing case tools.

3.3.1 Motivation

A skilled craftsman at work uses a variety of tools, each one tailored to the concrete work situation, and he effortlessly changes between these with quick alternations. Software developers, on the other hand, often find themselves constrained to few tools, as software tools in general do not integrate well. These tools are often closed with respect to extension, and they often use idiosyncratic file formats and interfaces making the alternation between tools hard.

Even though software tools do not often integrate well, we believe that *one* tool will never be appropriate for *every* activity in software development. Thus, integration of tools is important. The Knight tool, as described in section 2.2, is one example of a tool that complements existing tools for object-oriented modelling. The tool, however, does not itself offer functionality found in traditional CASE tools such as semantic checks and code-generation. Also, users of the Knight tool are likely to already be users of a traditional CASE-tool that they are comfortable with. There is thus a high motivation for integrating the Knight tool with existing CASE tools.

3.3.2 Types of integration

Integration may be viewed on an architectural level, i.e., as cooperation between high-level components interacting via high-level connectors. A distinction can be made between *data* and *processing* components, with processing components operating on the data components. The data in the data components may either be shared by several processing components or be separate. In either case, the processing components need to communicate in order to cooperate. If they are running simultaneously, they may do this by changing the shared data concurrently or by communicating changes to the replicated data. If they are running asynchronously they typically cooperate by changing the shared data, or communicating their changes to a shared third party. Table 1 illustrates this taxonomy for tools working on a common, logical core of data, and shows some typical applications. The taxonomy is inspired by Ellis et al.'s taxonomy of Computer Supported Cooperative Work (1991).

Time \ Data	**Shared**	**Separate**
Asynchronous	Import/export	Merging configuration management systems
Synchronous	Components in same process	Component technology interaction between applications

Table 1. Tool collaboration taxonomy and typical applications

www.manaraa.com

In the Knight project, we have worked with two types of integration:

- An asynchronous integration with shared data based on OMG XMI
- A synchronous integration with separate data based on Microsoft COM

Each of these types of integrations support different work scenarios. The asynchronous integration is light-weight, and it supports situations where the data needs to be moved from one tool to another, and were both tools are not available at the same time. It does, however, lead to a turn-around time in moving the data from one tool to the other. The synchronous integration, on the other hand, requires both tools to be present concurrently, both then support a tight, real-time integration. This is useful in situations were changes in one tool need to be reflected upon in the other tool immediately.

This section will present some of our experiences in implementing the asynchronous support using XMI – for our experiences on the synchronous integration the reader is referred to (Damm *et al.*, 2000b)

3.3.3 XMI integration

The asynchronous integration in Knight using shared data is based on XML Metadata Interchange (XMI, 1998, 1999). XMI is an accepted Object Management Group (OMG, `http://www.omg.org`) specification that provides the basis for an interchange format for UML models. The specification is in fact more general, as it specifies a way of creating an interchange format for any data that can be described by a metamodel.

Figure 25. Description of a part of a bank

Consider the simple diagram in Figure 25, which describes a part of a bank using a UML Class diagram. The diagram is a set of data. Since it describes the structure of a set of concrete account and customer objects, it is also metadata. If a set of metadata conforms to a specific semantics and syntax, it is called a 'model'. Since the diagram actually uses the UML class diagram notation, the diagram is a model. These two levels of abstraction comprise the two lowest levels in the OMG's Meta Object Facility (MOF, 1999). Two higher levels are also present (see Table 2): first, the UML notation itself can be described. This leads to a so-called metamodel: a set of data that describes a set of models, one of which is our model of a bank. Second, as there are many other metamodels than the UML metamodel, the MOF introduces a meta-metamodel level that can be used to describe all metamodels.

Based on a MOF-compliant metamodel such as the UML metamodel, the XMI standard describes a way to produce a grammar corresponding to that metamodel. This grammar can then be used to save and load models (e.g., a model of a bank), resulting in an interchange format for all models conforming to the metamodel. Figure 26 shows an extract of the XMI code exported by Knight for the Bank model. For each element in the model, an XMI element with the same name is present, and inside these elements follow further elements corresponding to the attributes of the element.

www.manaraa.com

Meta-level	MOF term(s)	Examples	Sample XMI artifacts
M3	meta-metamodel	The "MOF Model"	MOF DTD
M2	metamodel, or meta-metadata	The UML Metamodel	UML DTD, MOF XML file
M1	model, or metadata	A UML model of a bank	UML XML file
M0	data	Concrete bank account objects and customer objects	

Table 2. OMG MOF metadata architecture[2]

```
<Model_Management.Model xmi.id="_1">
   <Foundation.Core.ModelElement.name>New diagram</Foundation.Core.ModelElement.name>
   <Foundation.Core.Namespace.ownedElement>
     <Foundation.Core.Class xmi.id="::53umlClass0">
       <Foundation.Core.ModelElement.name>BankAccount</Foundation.Core.ModelElement.name>
       <Foundation.Core.Classifier.feature>
         <Foundation.Core.Operation xmi.id="::63umlOperation1">
           <Foundation.Core.ModelElement.name>Withdraw()</Foundation.Core.ModelElement.name>
 ...
```

Figure 26. Part of the bank model encoded in XML conforming to the UML DTD

To specify a grammar, the XMI standard uses XML (eXtensible Markup Language; XML, 1998) DTD's (Document Type Definitions) and the actual exchange files are then XML files conforming to this DTD. In other words, XMI specifies a set of rules for mapping a MOF compliant metamodel to a DTD, and a way of mapping a model to an XML file conforming to this DTD. The rules are not described here, as they are quite elaborate in their full detail. In this context it suffices to say that for each class in the metamodel, the rules create a grammar rule that can describe both the attributes of the class and references to elements associated to the class.

XMI implementation. Based on the UML metamodel standardised by the OMG, several companies have produced a UML DTD using the rules in the specification. We use the UML DTD provided as part of the IBM XMI Toolkit (`www.alphaworks.ibm.com/tech/xmitoolkit`), as this DTD is based on the *relaxed transformation rules* allowing for the exchange of models having elements that are not fully specified.

The basic import and export is quite simple. During the import the XML file is parsed using the Expat open source XML parser (`www.jclark.com/xml/expat.html`). Using the callbacks from the parser, an XML parse tree is built with the XML elements as nodes and their contents as subnodes. A traversal of this tree then creates the diagram. Saving is performed analogously: the diagram is traversed and from this an XML tree is built. This tree is then streamed to a text file.

XMI experiences. Even though the above implementation description sounds simple, we encountered a number of problems pertaining to the XMI standard. First of all, since the UML DTD is based directly on the UML metamodel it can only express what is in the UML metamodel. This is a problem since the UML metamodel is only concerned with UML *models* and not UML *diagrams* that have appearance. While this issue will most likely be solved in a future version of the UML in which the metamodel will describe diagrams (Kobryn, 1999), there is currently no standardised way of encoding presentational information in a UML XMI file.

[2] It should be noted that described four-level architecture is only the typical architecture. The number of levels is not fixed by the specification.

المنارة للاستشارات
www.manaraa.com

XMI allows for extension elements to be added to each element, which can be used to encode information, that is not part of the metamodel. These extensions are thus well suited as a place to store the presentational information, and this was the approach that we chose. However, since the concrete structure of the extensions is not described in the standard, different tools in practice encode this information in different ways. The consequence of this is that different tools can only exchange models and not diagrams. This is a problem, not only for the simple reason that it can be very annoying to have to re-layout a diagram, but also because positional information in a diagram often has semantics: two classes positioned close to each other will, e.g., most likely be closer related than two classes far away from each other.

The extension elements are also problematic for other reasons, especially in combination with round-trip engineering in which a number of tools import, change and then re-export the XMI file. While a tool may add any number of extensions to any element at export, another tool may also ignore their contents at import. However, the standard does not allow tools to discard extensions made by other tools when re-exporting a file. A tool must thus make sure that it stores all extensions during import even though it has no interest in their contents. We solved this in a simple way: at the import, the XML parse tree built during the parsing is scanned, and each node is then either used to create a diagram element, or it is stored in a list of ignored nodes. During the export, new nodes are then created from the current diagram contents and these are then merged with the nodes that were "ignored". Once this merged tree has been built it can be streamed to a file. This technique is also used for UML elements that our tool has no interest in such as elements from diagram types not supported by the tool.

The following scenario illustrates another complication with extensions in combination with round-trip engineering: a tool creates a UML element and an extension containing further information about the element. Another tool then imports this file, changes the state of the UML element and re-exports the file. This might result in the extension exported by the first tool being inconsistent with the new state of the UML element. While this is a general problem in integration, there is no general and simple solution to it. One possibility would be to add a timestamp attribute to each element. This would allow a tool to detect that an element, for which it has made an extension, had been changed since the extension was created.

As a more practical complication, it was difficult to find tools that could validate our exported XMI files. We are only aware of three tools that support the XMI specification and work on UML diagrams: the IBM XMI Toolkit, which can convert XMI files to and from Rational Rose files (`http://www.rational.com`), Rational Rose itself with an extra XMI plug-in (`http://www.rational.com/products/rose/support/patches`), and the open source CASE tool Argo UML (Robbind *et al.*, 2000; `http://www.argouml.org`). The IBM Toolkit and the Rose plug-in produce XMI files that are compatible, but neither of them is compatible with ArgoUML which uses an earlier version of the XMI specification. The new XMI 1.1 specification (XMI, 1999) will add another format, and so will future versions of the UML metamodel. If the XMI standard is to be used extensively as an interchange format, both the XMI standard and the UML metamodel must become more stable. Otherwise, what was supposed to be a unifying format will turn into a plethora of formats that tools must struggle to support.

3.4 Section summary

We described our experience with software architecture and the development process based on the Dragon project, and introduced the concept of architectural uncertainty. Also, we described how a concrete architectural uncertainty was handled using the Application Moderator pattern, and we looked at the concrete software architecture in the Knight project that was used to integrate tools.

المنارة للاستشارات
www.manaraa.com

4. Work in Progress and Future Work

As the previous sections have outlined, much of my work in the first part of the study is related to the construction of tools for software developers. In the second part of my Ph.D. study, I plan to continue my work in this direction, but with a focus on tools that support *cooperation* in software development. Cooperation is important at many levels: teams of developers needs to cooperate in implementing the software, programmers need to cooperate with other competencies; e.g. participatory designers, and developers need to cooperate with the end users that are eventually to use the system. (Dewan & Riedl, 1993) describes a number of concrete situations and some early support.

4.1 CSCW & groupware

Tools that support cooperative work are often called groupware, and the research field that looks at cooperative work from a computer perspective is often referred to as CSCW (Computer Supported Cooperative Work). Ellis *et al.* (1991) define groupware as:

> *Computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment*

They furthermore describe three key areas that must be attended when supporting group interaction: *communication, collaboration,* and *coordination*. Also, they organise groupware applications using the simple taxonomy illustrated in Figure 27.

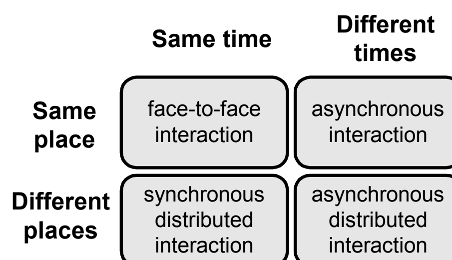


Figure 27. Groupware taxonomy

Using these terms and this taxonomy, the next section describes the concrete activities that I plan to undertake in the second part of the study.

4.2 Goals and activities

My long-term goals for the second part of the study are:

- to obtain a general understanding of cooperation in software development,
- to understand in what ways this cooperation can be supported by tools,
- to experiment with concrete tools, and to evaluate their subsequent use, and
- to develop general technologies that allow others to create their own collaboration tools more easily.

I hope to meet these goals though a number of concrete activities described below.

4.2.1 Mapping object models and user interfaces

My work on the relation between object models and user interfaces also touches upon cooperation issues, namely cooperation between members of the development team with different competencies – in this case user interface designers and object-oriented developers creating their respective artefact. I am still considering implementing tool support for mapping between these two artefacts.

المنارة للاستشارات
www.manaraa.com

4.2.2 Communicative support

I believe that the communicative support currently offered to teams of developers needs to be improved. The Ph.D. thesis of Henrik B. Christensen (1999) described one approach where developers could see changes to a shared code base on a number of so-called *software maps.* These maps were based on information from a custom configuration management system also implemented by Christensen. It would be interesting to have such support in the context of more commonly used configuration management systems such as Concurrent Versions System (`CVS;` Berliner, 1990; `http://www.sourcegear.com/CVS`). I have recently worked on a number of scripts that provide basic notifications on changes in a `CVS` repository. Using these, change information could be routed to a standardised notification service, e.g. the Elvin notification server (Fitzpatrick *et al.*, 1999). By subscribing to the events from this server, clients could visualise the changes in different ways

4.2.3 Shared editing

Group editors – groupware editors that allow users to collaboratively edit a shared document in *same time* but *different place* – constitute an important class of collaboration tools. Well-known examples include Dolphin (Streitz *et al.*, 1994), which supports collaborative brainstorming in a hypermedia environment, and GROVE (Ellis *et al.*, 1988), which supports collaboration in creating document outlines. We believe, though, that the topic is far from exhausted. Most, if not all, of the work in this area is relatively old, and given the technological improvements in the last 5-10 years we now possess new possibilities (and with these new challenges). Secondly, much of the existing work on shared editing seems to be centred on supporting less well-defined work practices than e.g. typical activities in software development. Given this I plan to work on two activities: Support for distributed diagramming and support for shared coding (see below).

Distributed diagramming in Knight

The Knight tool currently supports same-time, same-place collaboration in creating object models. However, as developers often are separated geographically, e.g. in different offices or in different company locations, it would be interesting to extend Knight to cover same-time, different-place diagram editing. We believe that there are not only interesting technical issues to investigate, but also many possibilities for interesting user studies and real work evaluations.

Distributed programming and debugging

Although a number of group text-editors exist, we know of only one application of group text editing to the task of programming, viz. `MDebug` (Dewan & Riedl, 1993). This system was implemented quite a long time ago, so it would be interesting to see whether it could be refined.

4.2.4 Merging different-time work

In some occasions different-time work is unavoidable, or it is not desirable for several parties to work on a shared artefact at the same time. This can lead to merging problems when several developers change the shared artefact in conflicting ways. As described in section 3.3, I have recently worked with the XMI standard. XMI offers a standardised way of representing structured data in a XML tree structure, and as tools for calculating differences between such trees are available (e.g. `XML Treediff` from `IBM Alphaworks`), it would be interesting to see if these could be used to merge XMI based data. Concretely, I would like to investigate if UML diagrams represented in XMI format can be gracefully merged using these tools in combination with rule-based merging (Munson & Dewan, 1994; Asklund, 1994). If this is successful one could imagine building a generic framework that could merge any XMI describable data.

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

APPENDICES

Appendix A. Bibliography

(Asklund, 1994). Asklund, A. Identifying Conflicts During Structural Merge. In *Proceedings of NWPER'94*, Lund, Sweden.

(Bass *et al.*, 1998). Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice*. Addison Wesley Longman

(Berliner, 1990). Berliner, B. CVS II: parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference.*

(Blaha *et al.*, 1991). Blaha, M., Eddy, F., Lorensen, W., Premerlani, W., Rumbaugh, J., *Object-Oriented modeling and design*. Prentice-Hall, New Jersey.

(Brown *et al.*, 1998). Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J. *Anti Patterns. Refactoring Software, Architectures, and Projects in Crisis*. Prentice Hall.

(Buschmann *et al.*, 1996). Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

(Christensen, 1999). Christensen, H.B. *Ragnarok: An Architecture Based Software Development Environment*, Ph.D. thesis, Department of Computer Science, University of Aarhus, DAIMI PB-540.

(Codd, 1970). Codd, E.F. A relational model for large shared data banks. In *Communications of the ACM* 13(6), pp. 377-387.

(Cook & Daniels, 1994). Cook, S., Daniels, J. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice Hall.

(Dahl *et al.*, 1966). Dahl O.-J., Nygaard K. "SIMULA an ALGOL-Based Simulation Language", *Communications of the ACM*, vol. 9, no. 9, September 1966, pp. 671-678.

(Dewan & Riedl, 1993). Dewan, P., Riedl, J. Toward computer-supported concurrent software engineering. In *IEEE Computer*, 26(1), pp. 17-27.

(Ellis *et al.*, 1988). Ellis, C.A., Gibbs, S.J., Rein, G.L. Design and use of a group editor. In *Proceedings of the IFIP TC 2/WG 2.7 Working Conference.*.

(Ellis *et al.*, 1991). Ellis, C. A., Gibbs, S. J., & Rein, G. L. Groupware: Some Issues and Experiences. *Communications of the ACM*, pp. 38-58, 34(1).

(Fitzpatrick *et al.*, 1999). Fitzpartick, G., Mansfield, T., Kaplan, S., Arnold, D., Phelps, T., Segall, B. Augmenting the Workaday World with Elvin. In *Proceedings of ECSCW'99*, Copenhagen, Denmark.

(Fowler, 1997). Fowler M. *UML Distilled*. Addison-Wesley.

(Gamma *et al.*, 1995). Gamma, E., Helm., R., Johnson, R., Vlissides, J. *Design Patterns. Elements of Reusable Software*. Addison-Wesley.

(Greenbaum *et al.*, 1991). Greenbaum J., & Kyng M. *Design at Work: Cooperative Design of Computer Systems*, Hillsdale New Jersey: Lawrence Erlbaum Associates.

(Hughes *et al.*, 1994). Hughes J., King V., Rodden T., Andersen H. "Moving Out of the Control Room: Ethnography in System Design", *Proceedings of CSCW '94*, Chapel Hill: ACM Press , 1994, pp 429 – 439.

(Jacobson *et al.*, 1999). Jacobson I., Booch G., Rumbaugh J. *The Unified Software Development Process*. Addison-Wesley.

المنارة للاستشارات
www.manaraa.com

(Janecek et al., 1999). Janecek, P., Ratzer, A.V., Mackay, W.E. Redesigning Design/CPN: Integrating Interaction and Petri Nets in Use. *Proceedings of the Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN,* 1999, 119-133.

(Kobryn, 1999). Kobryn, C. UML 2001: A Standardization Odyssey. In *Communications of the ACM,* pp. 29-37, 42(10).

(Kurtenbach, 1993). Kurtenbach, G. *The Design and Evaluation of Marking Menus.* Ph.D. Thesis, University of Toronto.

(Landay & Meyers, 1995). Landay, J.A., and Myers, B.A. Interactive Sketching for the Early Stages of User Interface Design. *Proceedings of CHI'95*, 45-50.

(Lyvtinen & Tahvanainen, 1992). Lyytinen, K., Tahvanainen; V.-P. *Next Generation CASE Tools*. IOS Press.

(Madsen *et al.*, 1993). Madsen O. L., Møller-Pedersen B., Nygaard K. *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley.

(Madsen, 1996). Madsen O. L. Open Issues in Object-Oriented Programming – a Scandinavian perspective. *Software Practice and Experience*, vol. 25, no. S4, December.

(MOF, 1999). MOF Revision Task Force. *Meta Object Facility Specification v. 1.3.* Document ad/99-06-05, Object Management Group.

(Munson & Dewan, 1994). Munson, J.P., Dewan, P. A Flexible Object Merging Framework. In *Proceedings of CSCW'94.*

(Mynatt *et al.*, 1999). Mynatt, E.D., Igarashi, T., Edwards, W.K., and LaMarca, A. Flatland: New Dimensions in Office Whiteboards. *Proceedings of CHI'99*, 346-353

(Opdyke, 1992). Opdyke, W. *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois at Urbana-Champaign.

(Robbind *et al.*, 2000). Robbins, J.E. & Redmiles, D.F. Cognitive support, UML adherence, and XMI interchange in Argo/UML. In *Information and Software Technology*, 42(2), 79-89, 2000

(Rubine, 1991). Rubine, D. Specifying gestures by example. *Proceedings of SIGGRAPH'91*, 329-337.

(Rumbaugh *et al.*, 1999). Jacobson I., Booch G., Rumbaugh J. *The Unified Modeling Language Reference Guide*. Addison-Wesley.

(Shaw, 1996). Shaw, M. Some Patterns for Software Architectures. In *Pattern Languages of Program Design 2*. Addison-Wesley.

(Streitz *et al.*, 1994). Streitz, N.A.; Geissler, J.; Haake, J.M.; Hol, J. DOLPHIN: integrated meeting support across local and remote desktop environments and liveboards. In *Proceedings of CSCW'94.*

(XMI, 1998). XMI Partners. *XML Metadata Interchange (XMI),* OMG Document ad/98-10-05, October 20. Available online at `http://www.omg.org/cgi-bin/doc?ad/98-10-05`.

(XMI, 1999). XMI Partners. *XML Metadata Interchange (XMI) 1.1 RTF Final Report.* OMG Document ad/99-10-04, October 20. Available online at `http://www.omg.org/cgi-bin/doc?ad/99-10-04`.

(XML, 1998). W3C. *Extensible Markup Language (XML) 1.0.* W3C Recommendation REC-xml-19980210, 10-Feb-98. Available online at `http://www.w3.org/TR/1998/REC-xml-19980210`

المنارة للاستشارات
www.manaraa.com

Appendix B. Author's Bibliography

Fully refereed conference papers

(Andersen *et al.*, 2000). Andersen, C.J., Hansen, K.M., Sandvad, E.S., Thomsen, M., Tyrsted, M.: Tool Support for Iterative System Development Activities: Issues and Experiences. To appear in *Proceedings of NWPER'2000*, The Ninth Nordic Workshop on Programming and Software Development Environment Research, Lillehammer/Norway, May 28-30.

(Christensen *et al.*, 1998a). Christensen M., Damm C.H., Hansen K.M., Sandvad E., Thomsen M. (1998): Architectures of Prototypes and Architectural Prototyping. In Mughal, K.A. & Opdahl, A.L. (Eds.) *Proceedings of NWPER'98*, Bergen, Norway, June, pp.247-267.

(Christensen *et al.*, 1998b). Christensen M., Crabtree A., Damm C.H., Hansen K.M., Madsen O.L., Marqvardsen P., Mogensen P., Sandvad E., Sloth L., Thomsen M. (1998): The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. In Jul, E.(Ed.) *Proceedings of ECOOP'98*, Brussels, Belgium, July, pp. 13-40.

(Christensen *et al.*, 1999a). Christensen M., Damm C.H., Hansen K.M., Sandvad E., Thomsen M.: Design and Evolution of Software Architecture in Practice. In Mingins, C., Meyer, B. (Eds.) *Proceedings of TOOLS Pacific 1999*, Melbourne, Australia, November, pp. 2-15.

(Damm *et al.*, 2000a). Damm C.H., Hansen K.M., Thomsen M. (2000): Tool Support for Cooperative Design: Gesture Based Modeling on an Electronic Whiteboard. In Turner, T., Szwillus, G., Czerwinski, M., Paterno, F. (Eds.) *Proceedings of CHI'2000*, The Hague, The Netherlands, April. New York: ACM Press

(Damm *et al.*, 2000b). Damm, C.H., Hansen, K.M., Thomsen, M., Tyrsted, M. (2000): CASE Tool Integration: Experiences and Issues in Using XMI and Component Technology. To appear in *Proceedings of TOOLS Europe 2000*, Mont St Michel & St Malo, France, June 5-8.

(Damm *et al.*, 2000c). Damm C.H., Hansen K.M., Thomsen M., Tyrsted M. (2000): Creative Object-Oriented Modelling: Support for Creativity, Flexibility, and Collaboration in CASE Tools. To appear in *Proceedings of ECOOP'2000*, Sophia Antipolis and Cannes, France, June 12-16.

(Hansen & Thomsen, 1999). Hansen K.M., Thomsen M.: The 'Domain Model Concealer' and 'Application Moderator' Patterns: Addressing Architectural Uncertainty in Interactive Systems. In Chen, J., Li, J., Meyer, B. (Eds.) Proceedings of TOOLS Asia 1999, Nanjing, China, September, pp. 177-190.

Position Papers

(Thomsen, 1999). Thomsen M.: Domain Object Models and User-interfaces. Presented at *Workshop on Interactive System Design and Object Models* at ECOOP'99, Lisbon, Portugal.

(Christensen *et al.*, 1999b). Christensen M., Damm C.H., Hansen C.M., Sandvad E., Thomsen M.: Software Architectural Evolution in the Dragon Project. Presented at *Workshop on Object-Oriented Architectural Evolution* at ECOOP'99, Lisbon, Portugal.

www.manaraa.com

Technical reports

(Thomsen, 1998a). Thomsen M.: *Persistent Storage of OO-models in Relational Databases*. COT report, Centre for IT research, document number COT/4-02-V1.5, 1998.

(Thomsen, 1998b). Thomsen M.: *Implementation of Rdbmap – a framework for object-relational mapping layers*.

In preparation

(Damm *et al.*, In Prep.). Damm C.H., Hansen K.M., Thomsen M., Tyrsted M.: On the UML's Support for Modelling Practice. *Submitted to <<UML'2000>>*

www.manaraa.com

Appendix C. Study and teaching activities during Del-A

Courses taken

Course	Type	Points	Semester
Aspects of Object-Oriented Programming (Bd.2)	B-course	2 points	Fall, 1998
Design Patterns and Frameworks	C-course	2 points	Fall, 1998
Cooperative Design in New Contexts	C-course	1 points	Fall, 1998
Post-WIMP Interaction	C-course	2 points	Spring, 1999
Cryptology (Bd.7)	B-course	2 points	Fall, 1999
Scientific Computing (Bd.5)	B-course	2 points	Spring, 2000
Advanced Interaction Techniques	C-course	2 points	Spring, 2000

Teaching

Course	Type	Semester
Computer Architecture (dArk)	Teaching assistant	Fall, 1998
Human-Computer Interaction (HCI)	Teaching assistant	Spring, 1999
Aspects of Object-Oriented Programming (AOOP)	Lecturer	Fall, 1999
Human-Computer Interaction (HCI)	Teaching assistant	Spring, 2000

المنارة للاستشارات
www.manaraa.com