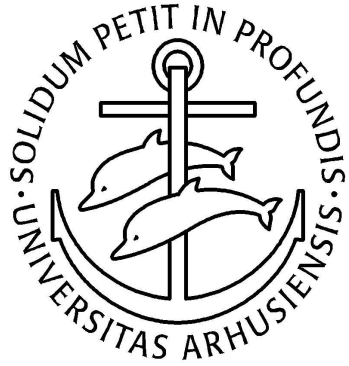
Experimental Object-Oriented Modelling

Klaus Marius Hansen

PhD Thesis



Department of Computer Science
University of Aarhus
Denmark

Experimental Object-Oriented Modelling

A Thesis
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Klaus Marius Hansen
March 27, 2002

Abstract

This thesis examines object-oriented modelling in experimental system development. Object-oriented modelling aims at representing concepts and phenomena of a problem domain in terms of classes and objects. Experimental system development seeks active experimentation in a system development project through, e.g., technical prototyping and active user involvement. We introduce and examine "experimental object-oriented modelling" as the intersection of these practices.

The contributions of this thesis are expected to be within three perspectives on models and modelling in experimental system development:

Grounding We develop an empirically based conceptualization of modelling and use of models in system development projects characterized by a high degree of uncertainty in requirements and point to implications for tools and techniques for modelling in such a setting.

Techniques We introduce and discuss techniques for handling and representing uncertainty when modelling in experimental system development. These techniques are centred on patterns and styles for handling uncertainty in object-oriented software architectures.

Tools We present the Knight tool designed for collaborative modelling in experimental system development and discuss its design, implementation, and evaluation. The tool has subsequently been successfully commercialized.

In summary, this thesis presents techniques and tools that advance the effectiveness and efficiency of experimental modelling.

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction and Motivation

Object-oriented modelling and experimental system development approaches are successful in terms of prevalence: object technology, and object-oriented modelling, is *the* technological paradigm for building large software systems. Its success has been achieved in particular through object-oriented languages such as Java and C++. Experimental system development is winning recognition and becoming widespread, particularly through the advent of "agile methods" .

http://www.agilealliance.org

However, their combination is often considered problematic: traditional object-oriented modelling is specification-oriented, and experimental system development is action-oriented through, e.g., prototyping with active user involvement. This thesis tries to identify use patterns, techniques, and tools which resolve this apparent contradiction through a focus on "experimental object-oriented modelling".

1.1 Experimental Object-Oriented Modelling

Object-oriented modelling is concerned with representing concepts and phenomena of a problem domain as classes and objects in a solution domain for a software system. Some of the effectiveness of such representations is traditionally claimed to be caused by the modelling process's similarities to natural cognitive processes of learning (Madsen et al., 1993). This thesis does not address extending the theoretical discourse of *why* object-oriented modelling works effectively but, rather, takes as starting point that object-oriented modelling helps in building robust, scalable, and malleable systems and tries to point to *how* its use can be more effective. Section 2.2 discusses object-oriented modelling in more detail.

Experimental system development emphasises concrete experiments throughout a system development project. Thus, the context in which object-oriented modelling is performed becomes important and warrants a focus on its combination with experimental system development techniques (Grønbæk et al., 1997). Experimental system development is typically used in projects that have a high degree of uncertainty with respect to requirements from a complex problem domain. Prototyping and participatory design techniques are ways of uncovering the

needed functionality of a system by active experimentation. Section 2.3 discusses experimental system development in more detail.

The problems resulting from the combination of object-oriented modelling and experimental system development include:

- *End user participation* involves sessions with users in which current problems and future ideas are formulated. These need to be constantly reconciled with an object-oriented model. Also, the understanding of problems and ideas needs to be formulated in an understandable way: problem domain experts seldom understand or appreciate formal notations used for modelling and object-oriented models need to be formal eventually.

- *Collaboration* using current modelling tools is problematic: most are run on ordinary desktop computers, giving little physical space for co-located collaboration. Moreover, their usual implementation of formal modelling notations only enables developers who understand this notation to use the tool.

- The tools that best support actual modelling practice are *physical tools* such as whiteboards and papers whereas object-oriented models need to be eventually represented in *virtual tools*. Iterative and incremental development requires that models are reworked and thus warrants shifts between physical and virtual representations.

- A *prototype* is often a central part of an experimental system development project and is constantly changed and extended. If object-oriented models are to be used in prototyping they need to evolve with prototypes.

A major part of this thesis consists of such identifications of what may make modelling in experimental system development problematic, and, based on these, the thesis identifies possible techniques and tools for experimental object-oriented modelling.

1.2 Structure

This thesis is divided into two parts: part one is a summary and discussion of our research into object-oriented modelling in experimental system development. Part two includes, in chronological order, co-authored papers related to this thesis. The summary part can be read independently of the papers.

1.2.1 Part I: Overview

Introduction and Motivation This chapter.

Background The empirical background — in the form of research projects — for this thesis is presented. We discuss object-oriented modelling and experimental system development in detail and introduce "experimental modelling".

Experimental Modelling Based on empirical studies of modelling, the chapter formulates a number of challenges for supporting experimental object-oriented modelling with techniques and tools. The following chapters will present solutions to some of these challenges.

Some Techniques for Experimental Modelling With focus on support for representing and handling uncertainty, the chapter presents techniques for experimental modelling that meet a number of the challenges presented in the previous chapter.

A Tool for Experimental Modelling We discuss current tool support for experimental modelling and present the design, implementation, and evaluation of the Knight tool for experimental modelling which meets most of the challenges presented in the "Experimental Modelling" chapter.

Conclusions and Future Work Beyond summarizing the contributions of this thesis, the chapter points to the ways in which the tools and techniques presented can be extended with respect to support for modelling as well as other work situtations.

1.2.2 Part II: Papers

The second part of this thesis consists of the following included papers:

- (Christensen et al., 1998a): Christensen, M., Crabtree, A., Damm, C., Hansen, K., Madsen, O., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., and Thomsen, M. (1998). The M.A.D. experience: Multiperspective Application Development in evolutionary prototyping. In *Proceedings of the ECOOP'1998*, pages 13–40.

  This paper reports how ethnography, participatory design, and object-orientation were combined in the "Dragon Project" — a large, experimental system development project. Its focus is on how, and why, multiple perspectives or practices in experimental system development collaborate.

- (Hansen and Thomsen, 1999): Hansen, K. M. and Thomsen, M. (1999). The "Domain Model Concealer" and "Application Moderator" patterns: Addressing architectural uncertainty in interactive systems. In *Proceedings of TOOLS Asia'99*, pages 177–190.

  Based on the experience from the Dragon Project and experience with open hypermedia system development (Hansen et al., 1999), we present architectures that help identify and encapsulate areas of an interactive system that

are likely to change due to uncertainties about the requirements. The architectures are presented as architectural patterns.

- (Christensen et al., 1999): Christensen, M., Damm, C., Hansen, K., Sandvad, E., and Thomsen, M. (1999). Creation and evolution of software architecture in practice. In *Proceedings of TOOLS Pacific'99*, pages 2–15.

  Also based on experiences from the Dragon Project, we discuss appropriate architectures and ways of creating architectures in experimental system development. In particular, we introduce a set of architectural practices as an "architectural strategy" for handling uncertainty.

- (Damm et al., 2000a): Damm, C., Hansen, K., and Thomsen, M. (2000a). Tool support for object-oriented cooperative design: Gesture-based modeling on an electronic whiteboard. In *Proceedings of CHI 2000, ACM Conference on Human Factors in Computing Systems*, pages 518–525.

  We introduce the Knight tool for collaborative, object-oriented modelling. This is the first of five included papers related to the Knight tool and even though they focus on different aspects of the Knight tool, they overlap considerably in their presentation of the tool and the interaction techniques used. This paper discusses empirical studies for and design and evaluation of the tool.

- (Damm et al., 2000d): Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000d). Tool integration: Experiences and issues in using XMI and component technology. In *Proceedings of TOOLS Europe'2000*, pages 94–107.

  This paper on the Knight tool focuses on tool integration and discusses its implementation. We present a concrete architecture for and an architectural conceptualization of tool integration.

- (Damm et al., 2000b): Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000b). Creative object-oriented modelling: Support for creativity, flexibility, and collaboration in CASE tools. In *Proceedings of ECOOP'2000*, pages 27–43.

  We discuss the nature of object-oriented modelling and suggest creativity, flexibility, and collaboration as design goals for tools that support object-oriented modelling. It is then discussed how the Knight tool fulfills these goals.

- (Damm et al., 2000c): Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000c). Supporting several levels of restriction in the UML. In *Proceedings of UML'2000*, pages 396–409.

  We argue that a particular area in which the Unified Modeling Language (UML; (OMG, 2001b)) has shortcomings in supporting modelling practice

is that it has only one fixed, formal view of what models are and how they look. We discuss extensions of the UML that varies along those two dimensions, and we discuss how the Knight tool has been amended to support these extensions.

- (Andersen et al., 2000): Andersen, C., Hansen, K., Sandvad, E., Thomsen, M., and Tyrsted, M. (2000). Tool support for iterative system development activities: Issues and experiences. In *Proceedings of NWPER'2000*, pages 1–21.

  With background in a number of projects in the Centre for Object Technoloy (COT), this paper argues that tools for experimental system development should focus on supporting concrete activities in contrast to, e.g., "analysis", "design", or "implementation". A set of cases from the COT projects are presented and discussed. `http://www.cit.dk/COT`

- (Hansen and Ratzer, 2002): Hansen, K. and Ratzer, A. (2002). Tool support for collaborative teaching and learning of object-oriented modelling. To appear in Proceedings of ACM ITiCSE'2002.

  This fifth paper on Knight reports an extensive use evaluation of the commercial version of the Knight tool. In doing so, it discusses how the collaborative aspects of the Knight tool may enhance the learning and teaching of object-oriented modelling.

1.3 Acknowledgements

Thanks in particular to the participants of the Dragon and Knight projects (and to a lesser degree the participants of the DESARTE and WorkSPACE projects) with whom I have performed most of the work presented in this thesis. Also thanks to Ole Lehrmann Madsen and Preben Holst Mogensen for giving the right advice. And, finally, thanks to Ellen and Liv for showing what is really important.

Klaus Marius Hansen
klaus@mariushansen.dk
Aarhus, Denmark
March 27, 2002

Chapter 2

Background

This chapter presents the background for this thesis in the form of an introduction to our empirical background and a brief survey and discussion of the literature on object-oriented modelling and experimental system development. Related work is discussed in the relevant chapters.

2.1 Empirical Background

During our PhD study, our research activities have centred around concrete development activities through participation in research projects. Figure 2.1 renders our project participation timewise.

http://www.intermedia.au.dk/dmm

http://www.cc.gatech.edu/projects/ael/research.html

http://as15.iguw.tuwien.ac.at/desarte/

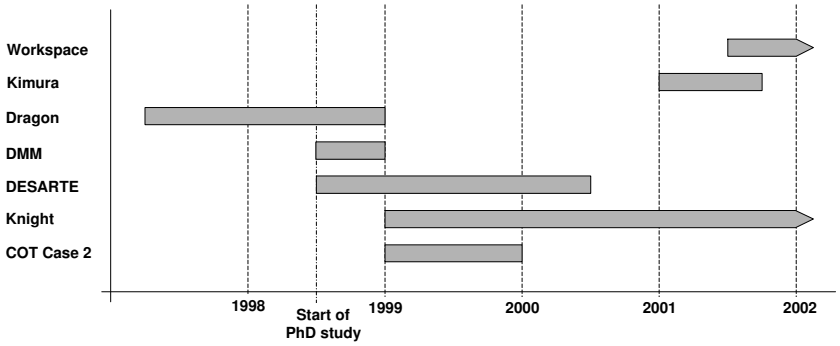


Figure 2.1: Research Project Participation

The rest of this section presents, in chronological order, the projects related to this thesis in sufficient detail to understand and motivate the following discussions. Our results in connetion to the "DMM", "Kimura", and "DESARTE" projects will not be discussed in this thesis. (Hansen et al., 1999) and (MacIntyre et al., 2001) are our co-authored papers related to these projects.

2.1.1 The Dragon Project

http://www.cit.dk/cot/case5-eng.html

The Dragon Project was initiated in March 1997 as a joint project between the University of Aarhus and a globally distributed shipping company with the purpose of developing a series of prototypes of a Global Customer Service System (GCSS). The concept of GCSS had evolved within the company based on a global Business Process Improvement (BPI) process aiming at the coordination of globally distributed sites and supporting an effective and efficient customer service. This improved customer service was, at the same time, meant to support local needs. In this context, shipping can be characterized as the transport of cargo in containers, and customer service in this area involves interacting with customers in, e.g., formulating prices, booking containers, and documenting ownership of cargo.
Participants in the Dragon Project from the University included: one project coordinator, one participatory designer, three full-time and three part-time object-oriented developers (of which the author of this thesis was one), one ethnographer, and, for part of the project, one usability expert. Participants from the company included senior management, business representatives from the major continental regions, administrative staff, customer service personnel, and external consultants.

Our co-authored papers related to the Dragon Project are (Christensen et al., 1998b), (Hansen, 1998), (Christensen et al., 1998a), (Christensen et al., 1999), and (Hansen and Thomsen, 1999).

2.1.2 Centre for Object Technology (COT) Case 2

http://www.cit.dk/COT/

http://www.uk.teknologisk.dk/

http://www.cit.dk/COT/case2-eng.html

Centre for Object Technology (COT) was a Danish collaboration project with participants from academia, industry, and the Danish Technological Institute. The Dragon Project was part of COT as "Case 5". Another subproject, in which we participated, was a technology transfer project: university researchers and developers from a private company designed an object-oriented control system for a flow meter. The control system was designed through an iterative prototyping process in which models and prototypes were gradually refined.

Our main reason for participating in this project was to gain empirical insight into object-oriented system development. This experience was later, together with the Dragon Project, used as the main empirical input to the Knight Project (Section 2.1.3).

Our co-authored papers related to this project are (Andersen et al., 2000) and the Knight Project papers referenced in Section 2.1.3.

2.1.3 The Knight Project

http://www.daimi.au.dk/~knight

The Knight Project started in February 1999 with the purpose of investigating and

prototyping innovative interaction techniques for supporting object-oriented system development. Our participation in the Dragon Project had pointed to a number of areas, in particular object-oriented modelling, in which enhanced tool support was needed.

The participants in the Knight Project were three PhD students (of which the author is one) and a Master's student. Initially, the project focussed on interaction techniques, but the scope broadened to include software architecture, modelling notations, and integration of modelling tools. The main result of the Knight Project, the "Knight" tool, is presented in Chapter 5.

Our co-authored papers related to the Knight Project are (Damm et al., 2000a), (Damm et al., 2000d), (Damm et al., 2000b), (Damm et al., 2000c), and (Hansen and Ratzer, 2002).

2.1.4 The WorkSPACE Project

The objective of the WorkSPACE project is to develop physical and virtual components and integrate these into augmented reality work places, environments, and fields. The work situations of design professions are the base for experiments with computer-supported individual work, local collaboration, and distributed collaboration.

`http://www.daimi.au.dk/workspace/`

A main virtual component of WorkSPACE is the "Topos" 3D environment for organizing heterogeneous information (Grønbæk et al., 2001) shown in use in Figure 2.2. Our current efforts in the WorkSPACE project are aimed towards integrat-

Figure 2.2: Topos in Use

ing Topos and Knight in order to provide modellers with a rich and heterogeneous environment for creating object-oriented models.

2.2 Object-Oriented Modelling

http://www.acm.org/dl/ A simple search on the ACM Digital Library on the word "modelling" shows the diversity of meanings assigned to the term: "mathematical", "system", "cognitive-user", "interaction", "object-oriented", "3D", "process", "entity-relationship", "declarative-geometry", "performance", "world", "business process", "task-oriented", "financial", "multi-level", "meta", "anomaly-detection" modelling and more turn up.

Even though there is a common core to all these kinds of modelling, this thesis limits itself to the discussion of types of modelling pertaining to object-oriented system development and in particular to experimental system development.

2.2.1 Modelling and Programming

In (Madsen, 1995), a distinction between the "Scandinavian" (*conceptual*) and the "reuse" (*software engineering*) perspectives on object-oriented programming and modelling is made. From a conceptual perspective, the main goal of object-oriented programming is to represent phenomena and concepts of a problem domain using abstractions of the used object-oriented programming language. From a software engineering perspective, object-oriented programming supports reuse, information hiding, and more which make it technically superior to, e.g., procedural or functional language alternatives for the development of large software systems.

2.2.1.1 The Conceptual Perspective

The *BETA* programming language (Madsen et al., 1993) is representative of the conceptual perspective. Object-oriented programming is seen as mapping a set of real-world phenomena and concepts to corresponding objects and classes in object-oriented programming languages. The set of real-world phenomena and concepts is called the *referent system* and the corresponding objects and classes are called the *model system*. The process of mapping phenomena to objects is then called *modelling* and the reverse process may be called *interpreting* (Figure 2.3).

In (Kristensen and Østerbye, 1994), modelling is considered in more detail, and *conceptual modelling* is defined as

> an understanding and construction process with the purpose of forming a model of some part of the world expressed by concepts and phenomena (Kristensen and Østerbye, 1994, page 2).

The view of (Kristensen and Østerbye, 1994) is a programming language view: programming is seen as a modelling process and conceptual modelling (and conceptual programming) is seen as a potentially successful candidate for common understanding between developers and others involved in system development. From this view, a framework for analysing and designing programming languages is developed. Moreover, a vocabulary for reasoning about programming language abstractions in relation to conceptual understanding is developed.

Even though sound software engineering is important from this conceptual perspective on object-orientation as well, it is seen as a side effect rather than as the most important goal of object-oriented programming.

2.2.1.2 The Software Engineering Perspective

Meyer (and through him the programming language "Eiffel") exemplifies the software engineering perspective on object-orientation (Meyer, 1997): he acknowledges the use of objects to model the system that we construct, but Meyers' models are, primarily, based on software engineering principles. Introducing object composition ("expanded types" in Eiffel), e.g., he justifies it as "improving efficiency", "providing better modeling", and "supporting basic types in a uniform object-oriented type system" (Meyer, 1997, page 256).

Meyer acknowledges the modelling power of object-oriented programming language constructs but warns that modelling with reference to a real or imagined part of the world may lead to a problematic design. Rather, object-oriented designs should be based on "clear, general, manageable, change-ready, and reusable" abstractions (Meyer, 1997, page 694).

Early object-oriented methods such as (Booch, 1991) also support this software engineering perspective: object technology's primary force is its sound, engineering foundation.

In some aspects there may be a tension between such a perspective and the conceptual perspective: when modelling a representative object, it is important that it represents a phenomenon and the properties of that phenomenon whereas information hiding, e.g., is concerned precisely with hiding the structure of the

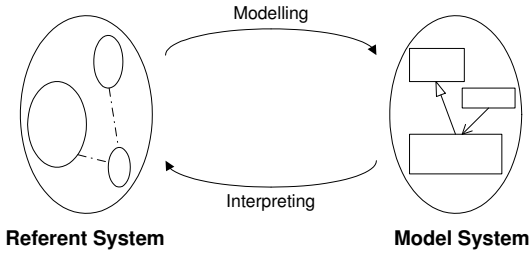


Figure 2.3: Modelling and Interpretation

object (Madsen et al., 1993). Nevertheless, both perspectives stresses modelling as important but disagree on what and how to model.

2.2.2 Modelling in Object-Oriented Methods

In the late 1980s the focus of many "methodologists", starting with Shlaer and Mellor (Shlaer and Mellor, 1989), moved from formulating methods for structured programming to working with object-oriented programming languages. The most influential of these were probably those of Coad and Yordon, Booch, Rumbaugh, and Jacobson (Coad and Yordon, 1990; Coad and Yordon, 1991; Booch, 1991; Rumbaugh et al., 1991; Jacobson et al., 1992). All of these are prescriptive: they mandate the use of a process (Booch, e.g., has "analysis", "design", "evolution", and "maintenance" phases) and particular artefacts that should be created during development. Moreover, they stress the use of modelling: Booch, e.g., explains that

> Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design (Booch, 1991, page 39)

Rumbaugh et al. state that a model is:

> an abstraction of something for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity (Rumbaugh et al., 1991, page 15)

and they stress the significance of modelling by calling their method the "Object Modeling Technique".

Coad and Yordon go further and present one of the first examples of a "metamodel" (see section 2.2.2.1) for defining their analysis modelling notation (Coad and Yordon, 1990).

There has been a number of reactions to these traditional, specification-oriented object-oriented methods, most recently in the form of so-called "agile methods" of which Extreme Programming (Beck, 1999) is the most prominent. Agile methods focus on "adaption instead of prediction" and on "people instead of processes" (Fowler and Highsmith, 2001): they try to adapt to complex and changing development situations by focussing on the interaction between and performance of people involved in development. Agile methods typically suggest the use of object-oriented programming languages and in this way implicitly require the use of object-oriented modelling even though they do not formally mandate its use.

http://agilealliance.org

Common to the traditional and more current methods seems to be the general view that a "method" consists of two aspects: a "notation" describing how the work products of the method should be and a "process" describing how and when these work products should be created (Booch, 1991; Fowler and Scott, 2000). The latest development in the object-oriented methods community has been to standardize, or rather create a framework for, these two aspects:

2.2.2.1 The Unified Modeling Language

The Unified Modeling Language (UML; (OMG, 2001b)) is an Object Management Group (OMG) standard for representing object-oriented models graphically in the form of diagrams. It has in practice replaced the modelling notations from methods such as Booch, Rumbaugh, and Jacobson.

http://www.omg.org

Currently, the UML consists of two levels:

- A *diagram level* that defines the visual appearance of UML diagrams. Figure 2.4 shows a simplistic example of a UML *class diagram*: The rectangles represent classes and the lines between elements are relationships such as inheritance and association. This diagram level is only defined in natural language in the UML specification.

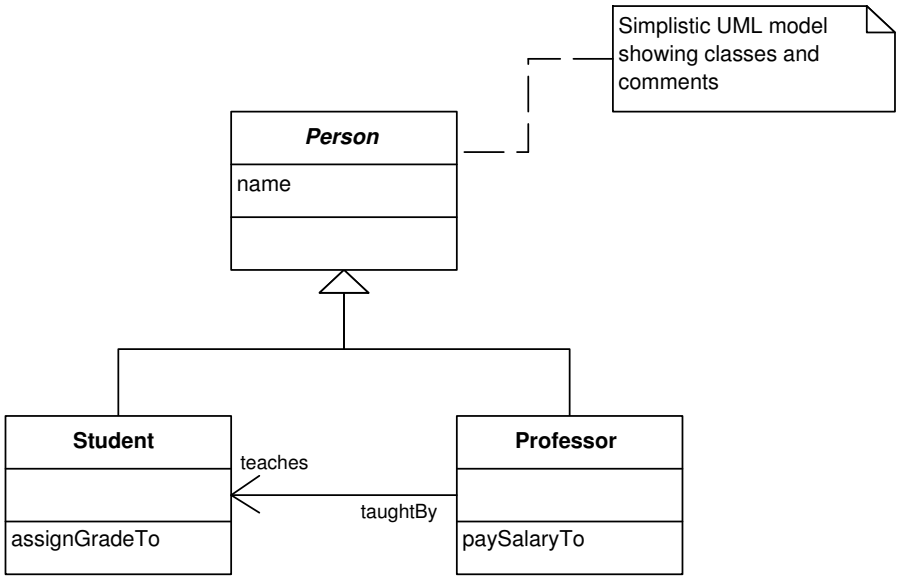


Figure 2.4: A Simple UML Class Diagram

- A *model level* that defines the semantics of the elements on UML diagrams. It is written in the form of a *metamodel* that contains classes defining UML elements such as "Class", "Generalization", and "Association". The meta-model is expressed as a UML model combined with well-formedness rules

in the form of constraints and general static and dynamic semantics in the form of natural language.

The fact that the UML is the result of a standardization effort, and the fact that it should replace notations from several object-oriented methods complicates the language: it consists of nine diagram types (class, object, use case, sequence, collaboration, statechart, activity, deployment, and component diagrams) defined by well over a hundred metaclasses in the metamodel.

Chapter 4, which is concerned with techniques for modelling, will present the UML in more detail as needed.

2.2.2.2 The Unified Software Development Process

The original version 0.8 of the UML was actually conceived by its inventors as a "Unified Method" with the intent of replacing the methods of that time. However, from version 0.9 and onwards, the focus with respect to the UML, was put on standardizing the modelling notations, which, through five iterations, has proved to be a complex task in its own (Kobryn, 1999).

Recently, the efforts regarding a standard modelling language have been extended to cover a "unified process". Jacobson et al. (Jacobson et al., 1999) present a generic framework for defining processes which is sold commercially as the "Rational Unified Process" (Kruchten, 1999). The Unified Process has an underlying object-oriented model presented in UML and new development processes are modelled using the basic elements of this model. An example of a process defined using the Unified Process is shown in Figure 2.5.

The horizontal axis shows "dynamic aspects" of the process such as iterations, increments, milestones, and phases. The vertical axis shows "static aspects" of the process such as disciplines, artefacts, and roles. In the "inception" phase, the idea for a product is formulated, in the "elaboration" phase these ideas are fleshed out, e.g., in the form of prototypes or requirement documents, in the "construction" phase, the product is iteratively built and tested, and, finally, in the "transition" phase, the product is finalized and put into production use. Each of the workflows on the vertical axis contribute differently in the different phases of development as indicated by the height of the shaded figures.

As evident from the above description, the Unified Process is a very generic framework for defining (iterative) processes: even though it was originally intented to cover traditional methods as mentioned above, it is so generic that it has even been used to instantiate the process *dX*, mimicking the Extreme Programming process.

http://www.objectmentor.com/publications/RUPvsXP.pdf

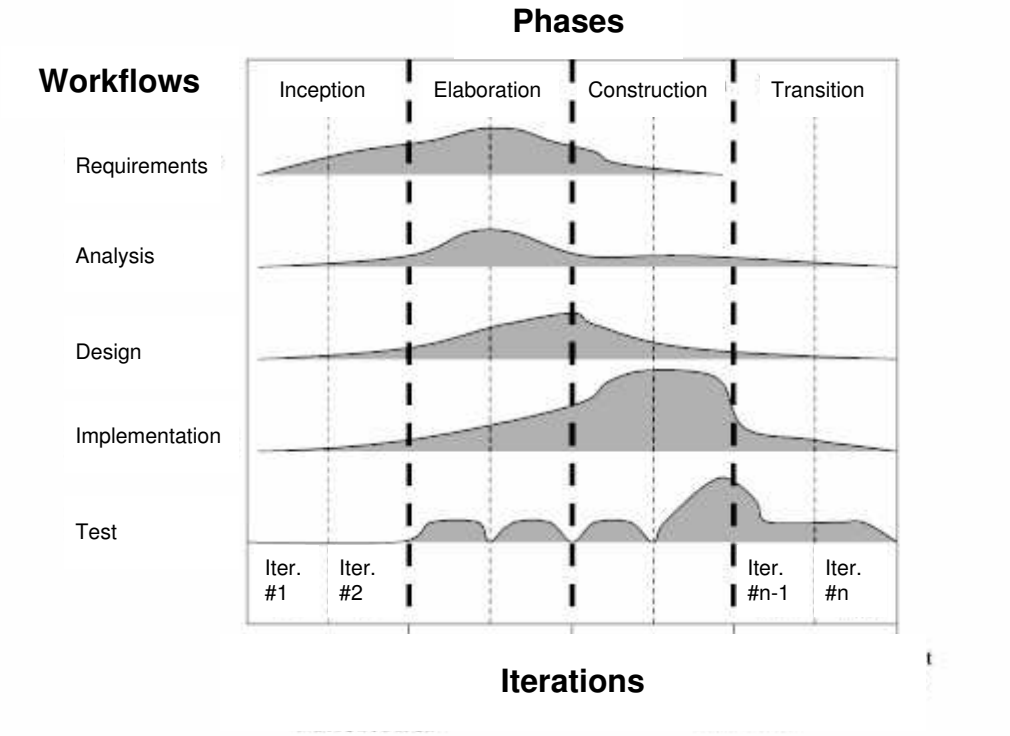


Figure 2.5: Two Dimensions of the Rational Unified Process (adapted from (Kruchten, 1999))

2.3 Experimental System Development

Traditional specification-oriented methods are severely limited since they are based on a detached, observation-based feed-forward approach to system development which seldom matches the complex and continually evolving work practices that many systems have to support (Grudin, 1989; Hughes et al., 1994). *Prototyping* approaches try to overcome this by focusing on the construction of a set of prototypes that can be evaluated (or even co-constructed) by potential users. *Participatory design* techniques try to involve potential users in the development process for moral and practical reasons. However, its research incarnations seldom consider fully implementing the system under construction.

(Cooperative) Experimental System Development (CESD) (Grønbæk et al., 1997) tries to combine prototyping and participatory design techniques by focusing on active user involvement throughout the entire development process. In their analysis of development projects, Grønbæk et al. introduce four conceptual dimensions of system development:

Concerns capture what a system development project is all about. *Project management* focuses on establishing and running projects. *Analysis* supports a cooperative learning process through investigation of user practices in relation to potential organizational changes. *Design* concerns visions of technology in use and the creation of a common understanding of possibilities. *Realization* is concerned with the technological implementation of design visions and with organizational change implied by the developed system. *Computer-supported cooperative work* focuses on the main concern of the developed system, i.e., its use.

Activities are the concrete actions and situations in system development. Examples of activities include project establishment, ethnographical investigations, design workshops, programming, and evaluation of a developed system.

Involved Domains are what the activities are directed towards understanding or changing. The starting point for cooperative design is most often the user organization practices. The developers' or designers' practice is also involved and may be changed. The software that is developed, the technology, and visions of technology in use are other domains that connect the users' and developers' practices.

Project Assignment is the system development task as understood by involved parties. Grudin discusses three traditional types of project assignments: *in-house development*, *custom/contract development*, and *packaged software/product development* (Grudin, 1991).

The two distinguishing characteristics of CESD as a system development approach framework are the distinction between concerns and activities and the relation of activities to the involved domains. Traditional waterfall development (Royse, 1970) equates activities and concerns as do many attempts to bring iteration to the development process (Boehm, 1988). CESD acknowledges that concrete activities may contribute to several abstract concerns and that they do so embedded in a related domain context.

Viewed on the activity level, CESD is in many ways similar to a number of system development approaches ranging from the original prototyping approach (Bally et al., 1977) to the more current such as STEPS (Floyd et al., 1989), Budde et al.'s prototyping approaches (Budde et al., 1992), Rapid Application Development (RAD) as exemplified by the Dynamic System Development Method (DSDM; (Stapleton, 1997)), M.A.D. (Christensen et al., 1998a), Adaptive Software Development (Highsmith, 2000), Rational Unified Process (Kruchten, 1999), Extreme Programming (XP; (Beck, 1999)), and the Crystal Family (Cockburn, 2001).

These approaches are all "experimental" in the sense that they try to overcome traditional specification approaches by experimenting with visions, designs, and solutions. CESD, with its historical background in particatory design, focuses on experiments such as dilemma games (Mogensen, 1994), future workshops (Kensing and Madsen, 1991), or prototyping whereas most of the mentioned approaches focus on experiments in the technical domain primarily through (evolutionary) prototyping.

In this sense, experimental system development is a reaction to characteristics of a large number of development projects: they, e.g., have a high degree of uncertainty regarding what to build and how to build it, need to involve collaboratively a large number of domains or perspectives, and need to create a system for and in a complex, heterogeneous environment. This orientation towards experimentation also requires a certain kind of activities (or techniques) to handle, e.g., iteration, incremental development, or parallelism.

This thesis is primarily interested in how these characteristics of experimental system development interplay with (object-oriented) modelling techniques:

2.4 "Experimental Modelling": A Working Definition

An implication of experimental system development for tools and techniques (abstracted activities) is that they should aim at supporting concrete *activities* instead of abstract *concerns*. This thesis tries to do just that for object-oriented modelling in experimental system development.

In system development, a diverse set of activities are concerned with developing models of a future system: user interface designers may create user interface mock-ups, consultants may create workflow specifications for future users of a system, and software developers may create UML models of software architectures. Common to all these activities is the translation process of modelling:

> Modelling is the process of representing the elements of a problem domain as elements in a solution domain with the intent of informing the development of a software system.

"Experimental modelling" is modelling in experimental system development.

The work products of modelling are models:

> Models are the artefacts constructed by modelling with the intent of informing the development of a software system.

2.5 Summary

This chapter presented background material needed for understanding this thesis. In particular object-oriented methods and experimental system development were presented and discussed.

It was argued that modelling is a central part of traditional and current object-oriented development methods. Chapter 3 will investigate the use of object-oriented modelling in the context of experimental system development and point to a number of challenges to experimental object-oriented modelling.

Chapter 3

Experimental Modelling

This chapter presents empirical studies of software development in three projects. We focus on modelling in particular, and based on the studies, we present and discuss challenges for tool support and techniques for experimental modelling.

The material in this chapter is based in part on (Christensen et al., 1998a) and (Damm et al., 2000a).

3.1 Empirical Studies

Our experiences from the Dragon Project provoked initial thoughts on what modelling is and how it should be supported. To elaborate on and ground these anecdotal experiences, we conducted two empirical studies of object-oriented modelling.

3.1.1 Work Instances

Below, the studies and distilled observations from the studies are presented. Experience from the Dragon Project is included as well. The distilled observations are "work instances" in the sense of (Christensen et al., 1998a): based on observation of work, they identify practical problems and present a situated method of resolution. All descriptions are kept in terms of the problem domain, and the descriptions of the problem and solution take the form of abstract scenarios. In doing so, they serve two purposes: to present a "problem space" for design against which to create solutions and a "solution space" for design against which to evaluate solutions. The problem and solution spaces are in our context connected to experimental modelling.

Instances compare to use cases and patterns: use cases describe how an

> instance of an actor carries out a number of different operations on the system. When a user uses the system, she or he will perform a behaviorally related sequence of transactions in a dialogue with the

system. We call such a special sequence a *use case* (Jacobson et al., 1992, page 127).

A pattern

> describes a problem which occurs over and over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice (Alexander et al., 1977)

Both concepts relate to problem solving and both are applicable to system development. To examine their differences, consider Cockburn's classification of use case definition and usage in four dimensions (Cockburn, 1997):

Purpose Use cases have either as purpose to gather user *stories* or *requirements*.

Contents The description of a use case can be *contradicting* or *consistent*. If it is consistent, it can be in *consistent prose* or have *formal content*.

Plurality Typically, a use case corresponds to either *one* or *multiple* scenarios.

Structure Sets of use cases are either required to be *unstructured*, *semi-formal*, or to have *formal structure*

Table 3.1 shows Cockburn's own use case concept as well as patterns and instances in these four dimensions. Seen from the four dimensions, patterns, instances, and Cockburn's use cases can all be seen as a variations of the general use case concept although patterns and instances noticeably have as "purpose" to gather stories: to function as accounts of events typically including a context to problems and proposed solutions.

In order to distinguish them further, we can look at the *domain* in which their description is made. This can be either the "use" (how is the system to be used?), the "solution" (how do we create the system?), or the "problem" domain (which work practices will the system be implemented in?). The results of this are shown in Table 3.1.

	Use Cases	*Patterns*	*Instances*
Purpose	requirements	stories	stories
Contents	consistent prose	consistent prose	contradicting
Plurality	multiple scenarios	multiple scenarios	one "scenario"
Structure	semi-formal	semi-formal	unstructured
Domain	use domain	solution domain	problem domain

Table 3.1: Dimensions of Use Cases, Patterns, and Instances

In Cockburn's (Cockburn, 1997) classification of use case definitions, "instances" would have as "purpose" to gather user "stories" to be used in re-construction and for ping-pong between different practices, and its "contents" is "contradicting" in the sense that it does not need to be logically consistent, its "plurality" is "one scenario" since it presents an abstract example of the resolution of a problem, and its "structure" lies somewhere in between "unstructured" and "semi-formal" in that it prescribes the presence of a problem and a solution but allows free prose in the descriptions of those. In contrast to use cases and patterns, its domain is the problem domain of the system under development.

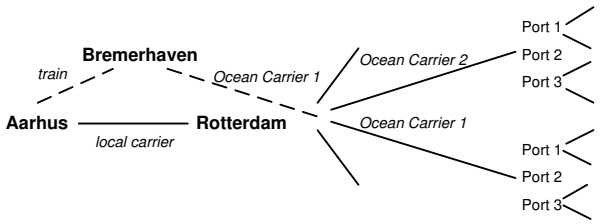


Figure 3.1: Pictorial Representation of the Bremerhaven Instance

3.1.1.1 An Instance Example

A prototypical example from the Dragon Project is the "Bremerhaven" instance. In Figure 3.1, the *problem* of rerouting containers in the event of carrier problems is illustrated. 200 containers cannot be shipped by the *local carrier* from Aarhus to Rotterdam since it does not call Aarhus this week. 150 of the containers are supposed to go on *ocean carrier 1* to Asia and 50 of the containers are supposed to go on *ocean carrier 2* to the Americas. The *solution* is to put all containers on a train to Bremerhaven, ship all containers with *ocean carrier 1*, and in Rotterdam transfer the containers for the Americas to *ocean carrier 2* that is calling Rotterdam.

The instance provided the starting point and context for assessing the prototype: it was used as an initial, fuzzy specification of a solution for the rerouting problem in the developed system and its refinements were used for evaluation purposes. This example also shows the relationship between the prototype and the instances of work: the instances were used as a foundation for prototyping but they also evolved during – and as a result of – cooperative prototyping. Moreover, prototyping pointed to areas in which observations and design were needed in such a way that the prototyping process influenced and coordinated other development efforts.

The type of instances presented in this chapter differs slightly from the type of instances employed in the Dragon Project: instead of distilling a large number of concrete situations into one very abstract problem-solution instance, our instances

focus on *how* the problem was actually resolved in the given context without the support of particular technologies. The intention is to inform the development of a solution by providing concrete details of a problem's typical resolution.

3.1.2 Dragon: Prototyping a new System

Settings The Dragon Project was presented in Section 2.1.1.

Participants The primary participants in the Dragon Project were university researchers and business representatives. The roles of the business representatives ranged from customer service agents to managers. The researchers acted primarily as designers and developers.

Procedure We participated in the project as modellers and developers. Our participation lasted for more than a year.

3.1.2.1 Dragon Instance: Working with a Problem Domain

Problem The researchers are trying to model the booking process of shipping customer service in the form of UML diagrams. Their knowledge of shipping is limited. Also, the customer service agent has no knowledge of object-oriented modelling.

Solution While the researchers model, a customer service agent tries to explain booking. Moreover, the customer service agent often interrupts the researchers in order to understand the semantics of the drawings. After some modelling sessions and encouragement from the developers, the customer service agent actually picks up some of the modelling notation and is able to participate more actively in the modelling sessions.

3.1.2.2 Dragon Instance: Combining Formal Diagrams and Informal Drawings

Problem During modelling of the booking problem domain, the researchers occasionally run into problems with lack of understanding. In such cases, the problem is often handled by the customer service agent by drawing small, informal drawings in connection to the UML diagrams on a blackboard. Most of the drawings are transient in nature, but some add essential knowledge to the UML diagrams.

Solution After the modelling sessions — or when a new area is about to be covered — the researchers transfer the contents of the blackboard to a computerised UML modelling tool. When they do this, they also make sure to transfer the most important drawings (Figure 3.2), even though this is not permitted by the UML.

3.1.3 COT Case 2: Building a New System

Settings The COT Case 2 project that was studied was presented in Section 2.1.2.

Participants The developers from the private company acted as domain experts in initial phases of development whereas the university researchers played the role as mentors, teaching object-oriented development techniques. The developers from the private company had no previous knowledge of object-orientation whereas the university researchers were experienced object-oriented developers.

Procedure We studied design sessions in the project during a two-week period. The sessions took place in meeting rooms equipped with whiteboards, overhead projectors, and computers. Typically, 2-4 people from the private company and 2-4 people from the research group participated in each session. We observed three sessions in detail and switched between being observers and active participants. In every session an observer took notes.

3.1.3.1 COT Instance: Alternating Between Tools

Problem A researcher and two developers are about to discuss a new area of the problem domain of flow meters. To brainstorm an initial design, the researcher and the developers draw initial models on a whiteboard. Just before lunch they run out of whiteboard space.

Solution The researcher captures the work so far using a digital camera (Figure 3.3). After lunch the researcher and the developers continue on a freshly-

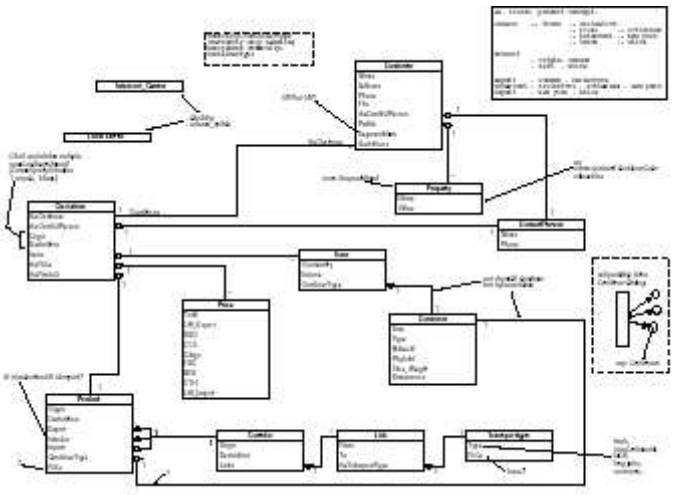


Figure 3.2: Combining UML and non-UML Elements

wiped whiteboard, while a developer redraws the diagrams from before lunch in a CASE tool using the photos as a reference.

Next morning, one of the researchers uses the CASE tool to print out diagrams that illustrate the details which were discussed the previous day. He places printouts of the diagrams on the overhead projector (Figure 3.4) and the developers use whiteboard markers to make changes.

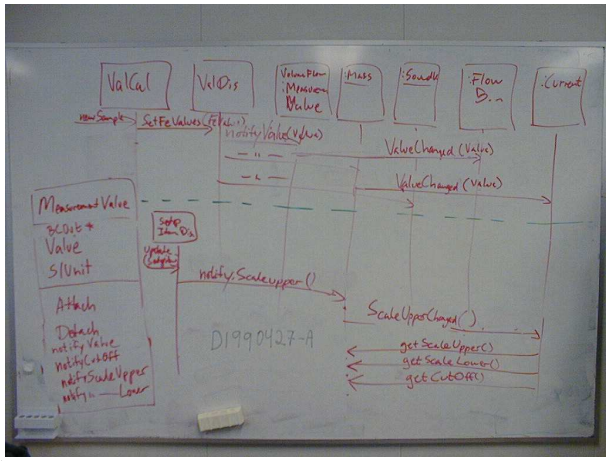


Figure 3.3: A Whiteboard Snapshot

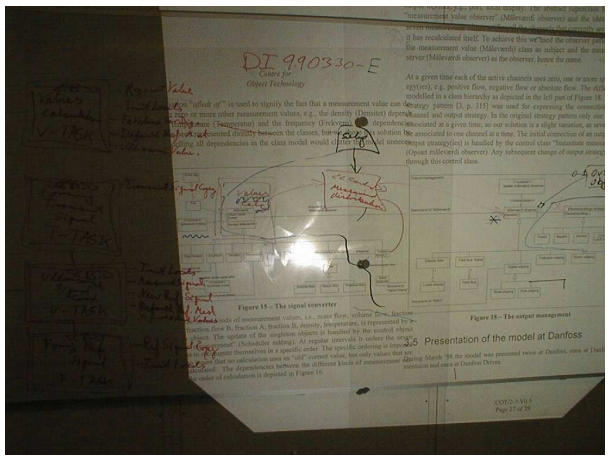


Figure 3.4: Projecting a Diagram on a Whiteboard

3.1.3.2 COT Instance: Modelling and Programming

Problem A researcher and a developer discuss how to include distribution in the design of their system. While using the whiteboard for discussion, another developer is coding up their previous design in order to keep their prototype up-to-date. At certain points he has to make conceptual changes in order to convert the design to code. Both kinds of changes need to be synchronized.

Solution The researchers and developers manually update the design documents, making sure that changes made during coding as well as during the design session are incorporated into the documents. Also, the code is manually updated so that the documentation and code are synchronized.

3.1.4 Mjølner Informatics: Redesigning an Existing System

Settings Mjølner Informatics is a medium-sized Danish software development company specializing in the development of complex, technically advanced software systems and object technology. One of their products is the Mjølner System (Knudsen et al., 1994), an object-oriented software development environment for the BETA programming language (Madsen et al., 1993). As part of its ongoing development, a restructuring of the environment was scheduled. `http://www.mjolner.com`

Participants Six developers from Mjølner Informatics attended the meeting. They had varying experience with object technology and the programming environment. Previously, four senior developers had each been responsible for their own part of the programming environment. Two junior developers were taking over the maintenance of these parts of the programming environment.

Procedure We observed a design session that involved restructuring of the development environment. We videotaped the design session and took notes especially emphasizing how the design was created and changed.

3.1.4.1 Mjølner Instance: Tweaking the Formal Notation on the Fly

Problem One of the senior developers is modelling the conceptual structure of part of the existing architecture on a blackboard. He wants to show how classes representing concepts are included in other concepts but is incapable of doing so using the UML.

Solution Another senior developer suggests a new notational element that shows conceptually nested classes and the other developers subsequently use this.

3.1.4.2 Mjølner Instance: Modelling Collaboratively

Problem A senior developer starts to draw a model of a part of the development environment on the blackboard. The other developers sit at a table and ask questions. Many of the questions are about the relations to another part of the programming environment.

Solution Following this, another senior developer starts to draw the part of the environment that he originally developed. Soon he discovers relations between the two parts of the development environment and asks the first developer to elaborate on his drawing. After the second developer finishes, the first developer continues on his drawing (Figure 3.5).

3.1.4.3 Mjølner Instance: Sketching Solutions

Problem Two senior developers and a junior developer are discussing the debugger part of the development environment. Soon it turns out that the solution they are discussing requires drastic changes to the integrated user interface. They are unable to use the UML to illustrate this and just creating a new notational element makes no sense.

Solution The junior developer draws the current user interface and connects its parts with UML classes on the blackboard. They try out various new suggestions for the user interface while discussing it in relation to the design already sketched out on the blackboard.

Figure 3.5: Collaboration at the Blackboard

3.2 Challenges

Based on the instances above, the rest of this chapter abstracts key characteristics of modelling in experimental system development with the intent of discussing implications for tools and techniques for such modelling processes.

3.2.1 Multiple Practices

"Practice: **3 a**: the continuous exercise of a profession" http://www.m-w.com

Models are created in collaboration between many people with different perspectives coming from different practices: developers have to collaborate with other developers and users and vice versa. In the COT case, engineers and object-orientation experts worked together in concert in order to create a prototype. In the Mjølner case, developers of separate modules of an application collaborated with each other and other developers, and in the Dragon Project, the perspectives of ethnography, participatory design, and object-orientation collaborated. This approach of Multiperspective Application Development provides a typical and representative example of collaboration between multiple practices. The next sections provide an overview of these perspectives and discuss their coordination efforts.

3.2.1.1 The Ethnography Perspective

A major source of system development failures is the unsuitability of the developed systems to the activities they are meant to support and in particular their unsuitability to the change of activities that a new system implies. The ethnography perspective aims at preventing this by creating an understanding of the work practices in which the system will be used (Blomberg et al., 1994). This understanding provides a concrete rendering of the working division of labour in contrast to idealized renderings such as workflow models (Rouncefield et al., 1994).

Ethnography achieves this through direct observation and naturalistic descriptions of work practice from the perspective of practitioners (Crabtree et al., 1997). Such descriptions allow for an understanding of practical problems and their solutions, in addition to an understanding of work practice. Thus, instances of work outline a problem space as well as a solution space for design. Ethnography's strength is understanding the problem space.

3.2.1.2 The Participatory Design Perspective

Participatory design (Greenbaum and Kyng, 1991) seeks to involve stakeholders in creative design processes for moral as well as practical reasons: stakeholders, potential end users in particular, will have their work influenced by the future system, and stakeholders are competent practitioners within their field, which enables them to cooperate effectively during design. Such involvement is often achieved in workshop-like settings through the focus on a common and concrete issue, e.g.,

descriptions of current work, scenarios for future possibilities, mock-ups, and prototypes (Greenbaum and Kyng, 1991; Grønbæk et al., 1997).

3.2.1.3 The Object-Orientation Perspective

In the Dragon Project, object-oriented modelling using the UML was started right from the beginning of the project, and it was an intentionally integrated part of each of the prototyping cycles. For the prototype, it was the starting point for implementation, and due to implementation it was continually updated.

The model was also used for triggering business discussion. In this context, the business representatives were able to see referent system concepts instead of model system concepts; comments on, e.g., multiplicities in associations were put forward in the context and linguistic terms of the business instead of in object-oriented terminology. In the context of formal reviews, in which participants from the company had been involved in a Business Process Improvement (BPI) project, and thus had an abstract perception of the business, the model triggered business discussions regarding future practice. The model was, however, not intelligible in all settings. For example, participants who had not been part of the BPI and did not have knowledge of formal methodologies for design had a hard time grasping the whole intent of discussing such a model.

3.2.1.4 Collaboration and Coordination

The abstract form of an instance provided a common representation for a variety of views on the same phenomena by different practices. When discussing the concept of "booking", e.g., the shipping agents would typically discuss it in terms of concrete tasks and work, the ethnographer would try to abstract work situations from concrete data, the participatory designer would design solutions to user interfaces, and the object-oriented developers would think in terms of classes and objects.

Another example is "rerouting": When rerouting entered the agenda, company personnel gave a brief introduction to the problem. The ethnographer collected a series of examples of reroutings. The participatory designer started to come up with ideas for supporting rerouting based on existing knowledge and experience. After three to four successions of discussions between the ethnographer and the participatory designer, an understanding of the problem as well as a first suggestion for the design emerged. (See Figure 3.1.)

3.2.2 Uncertainty

http://www.m-w.com

"Uncertain: **1**: Indefinite, indeterminate"

The fact that requirements are uncertain in a given project is one of the main reasons for using experimental system development. When building a new system, designers and developers will typically have to learn about a completely unknown

or little known problem domain. These kinds of uncertainties are exemplified in the studied projects:

3.2.2.1 Uncertainty as Indeterminateness

One kind of uncertainty in modelling is related to missing knowledge about an area of the problem domain, i.e., uncertainty about what the solution to a specific problem is. This leads to the inability to *decide* on how parts of the problem domain should be modelled.

In the Dragon Project, e.g., the first phase of development was concerned with exploring functional requirements for the system while the second phase used this knowledge to explore the technical foundation for the system in more detail — something that would not have been possible in the first phase due to uncertainty about *what* the system should do.

In the Mjølner study, lack of knowledge about a part of the integrated development environment was handled by having the developer, who was responsible for that part, model and restructure it. Only the sum of knowledge about the development environment allowed for a collaborative redesign of it.

3.2.2.2 Uncertainty as Indefiniteness

A second kind of uncertainty in modelling is related to imprecise knowledge about an area of the problem domain, i.e., uncertainty about which problem to solve.

In the COT Project this uncertainty was reciprocal: the researchers had little knowledge about engineering of flow meters, and the developers had little knowledge about object-orientation in general and object-oriented modelling and design in particular. This meant that the researchers were uncertain about *how to model* the control system for the flow meter. On the other hand, the developers were uncertain about *how to use modelling* for building the control system. The resolution was thus to combine these two perspectives and areas of uncertainty in modelling.

In the Dragon Project, the same kind of uncertainty led to repeated involvement of customer service agents by ethnographers, participatory designers, and object-oriented developers. As discussed below, the uncertainty — or at least uncertainty about how to build the prototype — was iteratively and incrementally removed during these interactions.

3.2.3 Iteration, Increments, and Parallelism

> "Iterate: **1**: the action or a process of iterating or repeating: as **a**: a procedure in which repetition of a sequence of operations yields results successively closer to a desired result" `http://www.m-w.com`

> "Increment: 1 : the action or process of increasing especially in quantity or value"

> "Parallel: 3 a: similar, analogous, or interdependent in tendency or development"

There is a product and a process perspective to "iteration" in system development: if development is iterative, the product is reworked from a product perspective and the process used to create one rework of the product is redone for the next rework. When working "incrementally", the product is gradually extended to become more complete. In this sense, experimental system development is always iterative and most often incremental. The system of concern is iteratively developed through a series of activities and a number of increments of a prototype. Models of problem domains are part of this iterative and incremental process: as activities contribute to a more satisfactory understanding of the system to be developed and the context in which it is to be implemented, models need to be iteratively and incrementally refined in order to fit this understanding. And these processes are often carried out in parallel.

The Dragon and the COT Project provide examples of this on different scales:

3.2.3.1 Activities and Concerns in the Dragon Project: Iteration, Increments, and Parallelism in the Large

Figure 3.6 shows some of the activities of the Dragon Project schematically. Time is shown horizontally and each column on the vertical axis represents development between two reviews. Each box represents an increment and iteration on the functionality connected to a particular area of concern: The booking component, e.g., went through seven increments and iterations from paper sketch to a highly vertical prototype while being extended in functionality. The overlaps on the vertical axis show parallel work: the fourth increment and iteration on the booking functionality was, e.g., done in parallel with work on six other functional components in the same iteration. The first phase (left) was mostly incremental and the second phase was mostly iterative seen from a product perspective.

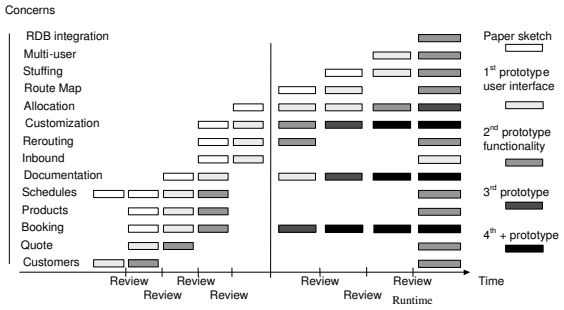


Figure 3.6: Activities in the Dragon Project

Each of the increments and iterations on a component involved input from ethnography and participatory design, and later an object-oriented implementation of a revised design was involved. Furthermore, evaluations in the form of workshops, demonstrations, or formal use evaluations were performed in connection to each review. This means that modelling, as well, had to be done iteratively, incrementally, and in parallel. As the knowledge of the part of the problem domain pertaining to each concern developed, new areas were modelled and old areas were revised. Reviews were used as synchronization points to keep development on track.

3.2.3.2 Modelling in COT: Iteration, Increments, and Parallelism in the Small

The development group in the COT Project was small and homogeneous in contrast to the development group of the Dragon Project. Furthermore, development was done in short, intensive iterations with longer breaks in-between. Nevertheless, iteration, increments, and parallelism were prevalent in this project as well.

During an iteration, the work product of each day would be a small report comprising a model in a Computer-Aided Software Engineering (CASE) tool and (digital) "whiteboard snapshots" of the results of modelling sessions. Later, this would be complemented with a running system and a listing of the code of that system. Incrementally, these products would converge to the final prototype of the system. Moreover, during the course of a day, subgroups would work on different parts of the system: one subgroup would, e.g., implement a design from a model by inputting the contents of a whiteboard into a CASE tool and subsequently write code, while another subgroup would continue modelling by working on a new part of the problem domain or discussing refinements of a previously modelled part. The work of both of these subgroups hadimplications for the complete model and had to be synchronized.

3.2.4 Heterogeneity

> "Heterogeneous: consisting of dissimilar or diverse ingredients or constituents" http://www.m-w.com

Models are represented in heterogeneous ways and sources of models are heterogeneous: models may, e.g., be shown graphically in the form of UML diagrams on whiteboards, they may be embedded in code, or they may be communicated verbally as shared understanding between developers. Models may, e.g., be created using observations, interviews, work artefacts, or previous systems as sources.

3.2.4.1 Heterogeneous Material as Input to Modelling

In the Dragon Project, a diversity of sources to modelling were available: *users* co-located with the development team were available for discussions and for giving *verbal or graphical accounts* of customer service work. *Horizontal prototypes*

designed with users showed a rich source of concepts and relationships that had to be covered by object-oriented models. Various types of *documents* also provided input: timetables, "bills of lading", and schedules, e.g., provided a basis for modelling the routing component of the system, and diagrams created during the previous BPI effort provided an initial source of concepts and scenarios to be covered by models. *Previous systems* were studied particularly in connection to current use to give input to an understanding of how work should — or should not — be supported by the future system. *Ethnographic analyses* provided critical, systematic input from actual work.

The starting point of the COT project was an earlier system that the participating engineers had built. This system, in combination with its specifications, provided a functional "checklist" for modelling. The engineers and the object-orientation experts provided different input for validation and verification of the constructed models: simplistically, the engineers, with problem domain expertise, could validate whether the models covered problem domain areas in sufficient detail and precision, and the object-orientation experts could verify the semantic correctness of the models with respect to object-oriented modelling practices in general and the UML in particular.

3.2.4.2 Heterogeneous Material as Output from Modelling

In both the COT Project and the Dragon Project, the prototypes were the single most important output from modelling. In-between this persistent executable embodiment and the informal and transient sketches drawn on whiteboards, a continuum of model representation was used.

Typically, CASE tools were used to draw with and to store models after these models had been drawn on whiteboards or paper or discussed verbally. Figure 3.2 shows an example from the Dragon Project of a UML model decorated with non-UML constructs (labels and drawings) that represent the semantic context to the model.

The model was tightly integrated with the code through the CASE tool used (Christensen and Sandvad, 1996), but simplified versions of the model were continuously created for presentations and workshops. Other views created included abstracted relationships between objects in the system in the form of architectural diagrams.

The object-oriented model and the user interface had a dual relationship, particularly in the Dragon Project: the user interface design was, as discussed above, used as input to modelling but modelling sessions also provided input to user interface design. In fact, a subsequent analysis showed a close correspondence between objects from the object-oriented solution domain model and graphical user interface widgets from the graphical user interface (Damm et al., 1997).

3.2.5 Open-endedness

> "Open-ended: not rigorously fixed: as **a**: adaptable to the developing needs of a situation" http://www.m-w.com

During modelling, situations occur in which any given (formal) modelling notation is not able to express what its users need it to express. Physical tools, such as pen and paper or whiteboards, support needed extensions well: the semantics and extents of the formal language used for modelling are in the mind of the designers. Computerized tools, however, support this poorly: an implementation of a formal notation (or formality in general) need to be coupled with the kind of openness needed for modelling.

We observed two major types of ongoing changes needed to the modelling languages used:

3.2.5.1 Situated Extensions

In the Mjølner case, several extensions to the UML were needed. The language used for the implementation of the tool is the BETA language (Madsen et al., 1993), which among other supports nested (or inner) classes and virtual classes (Madsen and Møller-Pedersen, 1989) both of which are not supported by the UML. A notation for this was developed on the fly by a designer and subsequently used for the rest of the session. In the Dragon Project, an example of a graphical extension was how colours were used to show semantic groupings of classes into, e.g., classes concerning bookings and classes concerning routing.

Common to all our observations was the coupling of different kinds of material. Freehand drawings illustrating a point graphically and informally was the most common of such coupling. However, coupling of formal languages also occurred, e.g., in coupling sketches of user interfaces or of flowcharts for an earlier system to a UML model.

3.2.5.2 Changing Semantics

Another kind of openness needed in the studied cases was an ability for the formal notation to adapt to the level of detail and viewpoint needed in a particular situation.

In the Mjølner case, an example was the need for "filters" on a diagram, e.g., in the form of shortcutting relationships between classes so that if class A was associated to class B which was associated to class C, only class A, class C, and an association between these would be shown. This kind of incompleteness was also often used to, e.g., avoid giving an element a name or to create "dangling" relationships only connected to one element in order to show that an element had a reference to another element, but that this other element was not known.

3.3 Summary

The preceeding sections have presented characteristic challenges to be dealt with in experimental modelling. Obviously, this affects the way tools and techniques for experimental system development need to be designed and constructed. In summary: To create appropriate tools and techniques for experimental modelling, we need to try to resolve the following challenges:

- *Multiple Practices.* The input from different perspectives is essential to understand and tackle the complexity of an experimental system development project.
- *Uncertainty.* Not knowing *what* to do or *how* to do it will happen at various points in time for various areas that are modelled in experimental system development.
- *Iteration, Increments, and Parallelism.* Models need to be produced and revised iteratively and incrementally as the understanding of a problem domain evolves. Multiple perspectives collaborating often implies a degree of parallelism also in modelling.
- *Heterogeneity.* Material used in modelling takes a variety of forms and in order to support, e.g., collaboration between multiple perspectives, material produced during modelling will have to take diverse forms.
- *Open-endedness.* A fixed notation or a prescribed process for modelling will often break down in experimental system development in general and in experimental modelling in particular.

The next two chapters will discuss work that addresses these challenges.

Chapter 4

Some Techniques for Experimental Modelling

Instances and prototyping as techniques for experimental modelling were discussed in chapter 3. Based on the challenges of that chapter, we now focus on two specific sets of techniques for experimental modelling:

- *Techniques for supporting experimental modelling*. Which techniques are appropriate for handling challenges such as uncertainty, iteration, and parallelism in experimental modelling? Here we focus on strategies and patterns for software architecture.

- *Techniques for representing experimental modelling*. How should models be represented while they are evolved through experimental modelling? We focus on object modelling notations, and in particular the UML, for doing this.

The material in this chapter is based in part on (Christensen et al., 1999), (Hansen and Thomsen, 1999), and (Damm et al., 2000c).

4.1 Supporting Experimental Modelling — an Architectural View

Handling the challenge of uncertainty, there are basically two approaches to designing a software architecture: try to predict or anticipate the change and design the software architecture to cope with that change, or face the change with appropriate tools and techniques to restructure the software architecture if needed. The "architectural strategy", a set of software architectures, their context, and evolution, outlined in this section, attempts to balance these approaches: architectural patterns that attempt to isolate impacts of change are presented along with a general approach to architectural restructuring and refactoring.

4.1.1 What is "Software Architecture"?

http://www.sei.cmu.edu/architecture/definitions.html

There are numerous definitions of software architectures that can be categorized into two basic kinds of software architecture definitions that guide how software architecture is handled during system development. The first focuses on structures of the software being developed and is typical of, e.g., (Bass et al., 1998; Garlan and Shaw, 1993):

> *The software architecture of a software system is the structure(s) of a software system in terms of components, their external properties, and their interrelationships*

The second focuses on software architecture as an an overarching understanding and is typical of, e.g., (Crispen and Stuckey, 1994; Hayes-Roth, 1995):

> *The software architecture of a software system is an overall understanding of the system that guides its construction and use*

This second use of the words "software architecture" is exemplified by Extreme Programming's use of "metaphor" to convey architecture (Beck, 1999, page 56) or Cockburn's conception of the software architect as "storyteller" (Cockburn, 2001, page 6).

The creation and representation of software architectures are, in both cases, a modelling process: structures and relations or metaphors are created to represent the solution of a set of high-level problems that the system needs to face.

We consider software architectures from both perspectives since each perspective has its strengths and weaknesses: the structural view is good for analysis and representation whereas the metaphorical view is good for getting an overall understanding of the system that is being developed.

4.1.1.1 Software Architecture Notation

To describe software architectures in a somewhat precise manner, we will use the notation presented in Figure 4.1.

The notation is an extension of the notation of Bass et al. (Bass et al., 1998). In general, solid lines denote processing and dashed lines denote data: a "computational component" could, in this context, be a running thread, a "computational data component" could, e.g., be an active relational database, and a "data component" could, e.g., be a file.

As noted by several authors, the UML has problems in representing software architectures (Egyed and Medvidovic, 1999; Hilliard, 1999) from different views on a sufficient level of abstraction. Thus, the present notation will be used interchangeably with UML diagrams for explaining architectures. It should be noted that the above notation could just be considered a UML "profile" (OMG, 2001b) — a lightweight extension to the UML that, in this case, also introduces new icons for model elements.

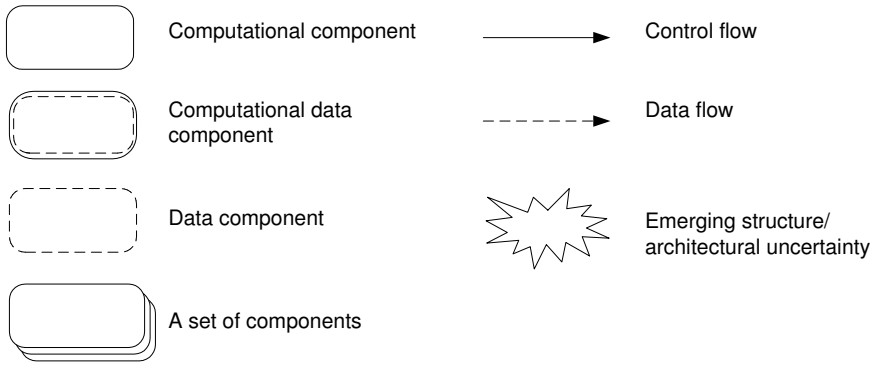


Figure 4.1: Software Architecture Notation

4.1.2 An Architectural Strategy in Experimental System Development

Next, we consider software architecture in the Dragon Project and in particular how they might generalize to experimental system development in general as an "architectural strategy".

In the Dragon Project, two senior developers outlined an initial architecture, following a short phase of a couple of weeks in which a first set of horizontal user interfaces and a problem domain model for a very limited part of the business were made. This lead to an initial architecture:

4.1.2.1 The Prototyping Phase: Initial Dragon Architecture

During the first phase of development a number of constraints on how the software architecture should be designed were identified particularly related to the challenges of iteration, increments, and parallelism. In particular the architecture should

- *provide a stable structure* in which the system could evolve horizontally in terms of functionality covered and vertically in terms of completeness of a specific functionality,
- *be flexible* to allow for experimentation within the prototyping cycles, and
- *support a proper work organization* in which developers could work in parallel on different areas of the application.

Figure 4.2 shows how we attempted to achieve this in the prototyping phase. The presented architecture diagram was not actually produced at that time, but it *is* a faithfull rendering of how the architecture was meant to be and how it was evolved during the prototyping phase.

There are three major parts of the architecture: the *User Interface* components, the *Problem Domain Objects*, and the *Business Functions*.

The star around the Business Functions, which implement complex, business-related functionality, shows that they represented an architectural uncertainty: they, and their internal structure in particular, were unknown at that point of time and flexibility in the structure of this part had to be provided for in order to prototype the business functions.

The Problem Domain Objects implement the concepts and phenomena found in the customer service problem domain in a "pure" form: they are kept free from implementation-specific details such as distribution or persistence implementations.

Each of the User Interface components corresponds to an area of the use domain such as booking, quoting, and scheduling. The User Interface Mirror part of the User Interface components acted as an interface between the Problem Domain Objects and the actual user interface controls. The details of this part of the architecture are presented in pattern format in section 4.1.3.4 including a depiction of how the architecture tried to resolve another architectural uncertainty resulting from the constant evolution of the user interface components.

4.1.2.2 The Experimental System Development Phase: New Dragon Architecture

At the beginning of the second phase, the architecture was inconvenient since it showed an unclear separation of concerns between components and inconvenient dependencies between components. Moreover, the focus and scope of the project had changed so that new requirements for the architecture included that it should

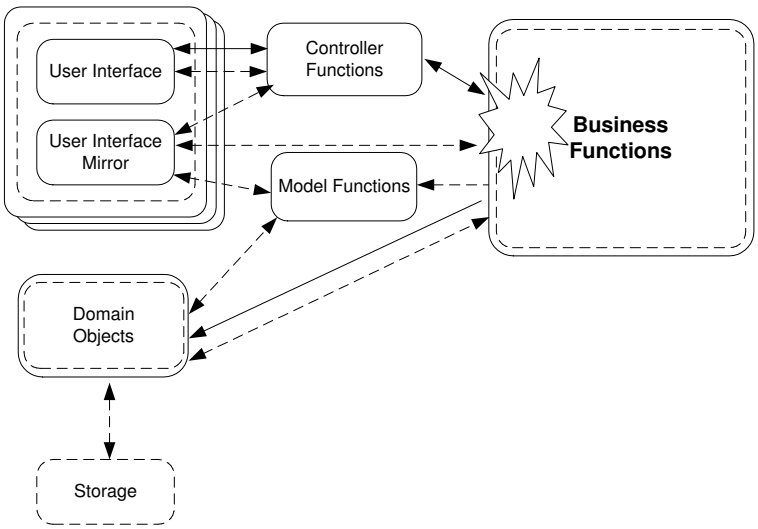


Figure 4.2: Initial Dragon Architecture

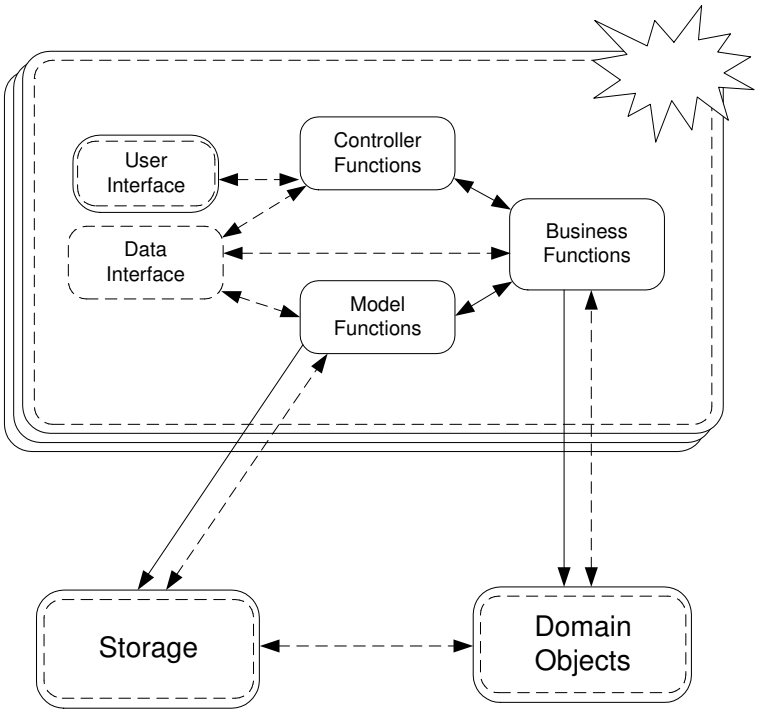


Figure 4.3: New Dragon Architecture

- still *support business functionality experiments* as new areas of the use domain were considered,
- *support technical explorations* such as the implementation of distributed multi-user functionality and integration with relational databases, and
- *support identification and creation of black-box components and their topology*.

Figure 4.3 shows the new architecture after an architectural refactoring (see Section 4.1.2.3) that took these constraints into account.

This shift of focus resulted in that the new architectural uncertainty was concerned the definition of the *Problem Domain Area* components. The previous phase had shown that dividing the architecture along the lines of the identified functionality areas would be sound. In this way, we were able to focus on the component topology in a flexible way during the experimental system development phase, i.e., the actual high-level components had been identified in the previous phase and this phase could focus on experimenting with how to actually connect components. Moreover, experiments could continue with the contents of the Problem Domain Area components without impacting the architecture.

4.1.2.3 Architectural Refactorings

The two architectures were the result of successful *architectural refactorings*. Whereas (code) refactorings (Opdyke, 1992; Fowler et al., 1999) are concerned with semanctic-preserving transformations of classes, attributes, and methods, architectural refactorings are concerned with a functionality-preserving transformation of the software architecture of a system. The goal of (code) refactoring is to create a better low-level design in terms of, e.g., greater flexibility or less duplication of code whereas the goal of architectural refactoring needs to be seen in a broader perspective: the reasons for architectural changes are both social and technical.

The success of the architectural refactorings in the Dragon Project had three prerequisites: it was

- *desirable* to do the refactorings since the current architectures were deemed impractical for the current goals,
- *practical* to do refactorings between phases, and
- *possible* to define new architectures because of increased knowledge obtained during previous phases.

The sum of the architectural refactorings and the focus on uncertainties created an architectural strategy that is applicable to projects as the Dragon Project: through a focus on providing flexibility as needed, and in particular in response to architectural uncertainties, explicit, scheduled architectural refactorings can help ensure that the architecture evolves along with the technical and social context of the system development project.

4.1.3 Addressing Architectural Uncertainty with Architectural Patterns

We define uncertainty within a software architecture as follows:

> *An* architectural uncertainty *represents either indefiniteness in the understanding of an architectural structure or indeterminateness in the representation of an architectural structure.*

In the Dragon project, the development process employed a high level of participation by users, which implied frequent changes to the user interfaces. These change requests forced us to continually evolve the part of the application that implemented the user interface. Thus, the user interface presented an architectural uncertainty concerning the components that communicated with it. This was particularly the case in the first phases of development as outlined in section 4.1.2.

(Hansen et al., 1999) presents an architectural uncertainty regarding the problem domain model instead of the user interface: while extending the Webvise hypermedia system (Grønbæk et al., 1999), the Webvise server's object-oriented

model of hypermedia was about to be replaced with a standardized version. From the Webvise clients' view, the server's model thus presented an architectural uncertainty.

As a way of resolving or addressing this uncertainty, the software architectures in the two projects were designed to anticipate the most likely changes caused by the uncertainties. We will use architectural patterns (Buschmann et al., 1996; Shaw, 1995) to present the software architectures of the projects: the "Domain Model Concealer" pattern describes the solution of the Webvise-related project and the "Application Moderator" pattern describes the solution of the Dragon Project.

The reason for using architectural patterns in our presentation is two-fold. First, it allows a short and succinct presentation with the focus on commonalities with other software architectures. Secondly, architectural patterns, like design patterns (Gamma et al., 1995), present a common solution to a common problem in a common context which matches the concept of architectural uncertainty well.

In the nature of the pattern concept, the actual presentation of the architectural patterns is taken almost verbatim from (Hansen and Thomsen, 1999).

4.1.3.1 Pattern Format

The pattern format we use to describe architectural patterns varies somewhat from the format of (Buschmann et al., 1996), (Shaw, 1995), and (Shaw, 1996). The first can be too verbose to efficiently communicate the essence of architectural patterns, and the last two can be too short to give sufficient understanding for actually applying the patterns. Our format is introduced below:

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 an effective solution to the stated problem? This section includes a deqscription of the pattern's high-level static structure in the form of a UML 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?

It 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 (Brown et al., 1998) "deductive mini-pattern" template.

To enable better dissemination of the patterns, the traditional "known uses" and "related patterns" are discussed externally to the presentation of the patterns. Also, a common, sample implementation is used:

4.1.3.2 A Sample Implementation

We will use a common and simplistic implementation in the description of our patterns. It is a common implementation since that allows for a more direct comparison between the two patterns. It is a simplistic implementation since it has to be presented in a short space and should be easy to replicate. The problem domain model, in the form of a class diagram, and the user interface of our common *Financial History* application are shown in Figure 4.4.

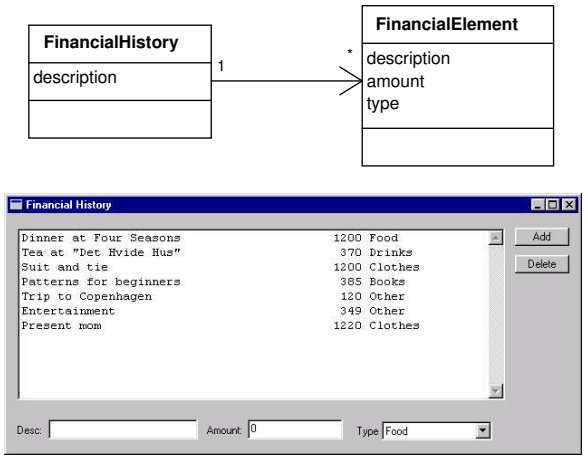


Figure 4.4: Problem Domain Model and User Interface for the Financial History Application

Given this application, a user is able to maintain a very simple financial history with financial elements describing what a given expense was for. Financial elements may be added or deleted via the user interface.

4.1.3.3 Domain Model Concealer

> *The Domain Model Concealer architectural pattern divides interactive applications into a problem domain-related part (the Problem Domain Model), an interface to this model (Domain Model Concealer), a User Interface part containing the user interface, and a User Interface Logic part that implements user interface related functionality by communicating with the Problem Domain Model through the Domain Model Concealer.*

Problem. How does one design the architecture of an interactive system so that the user interface functionality is separated from the problem domain- related functionality? How does one ensure that frequent changes to the Problem Domain Model do not require frequent changes to the whole application?

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, and changes to the problem domain model should have no or minimal impact on the rest of the application.

Solution. The Domain Model Concealer pattern divides the application into four parts: User Interface, User Interface Logic, Domain Model Concealer, and Problem Domain Model. Optionally, a Testing component can be implemented. The overall structure is shown in Figure 4.5.

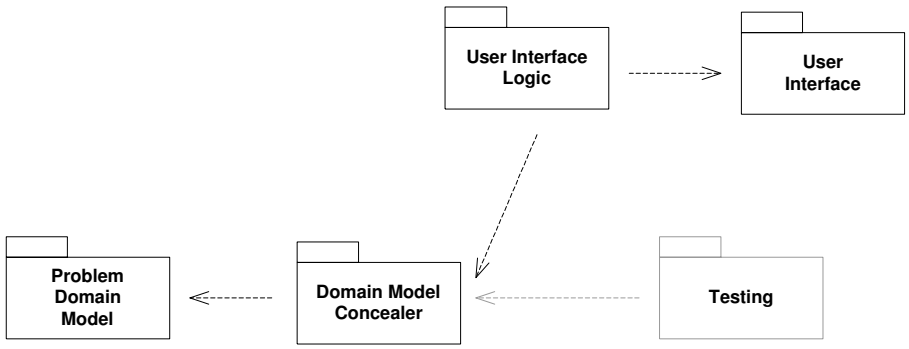


Figure 4.5: Overall Structure of the Problem Domain Concealer Pattern

The User Interface component implements the user interface and consists of declarations and instantiations of the widgets in the user-interface of the application. The Problem Domain Model contains the domain-related classes and functions, and possibly other functionality. It can be implemented independently of the User Interface component. The Domain Model Concealer contains an interface to the Problem Domain Model component. The Domain Model Concealer is thus dependent on the Problem Domain Model component but independent of the User Interface component.

The User Interface Logic implements the functionality offered by the User Interface component by collaborating with the Problem Domain Model component through the Domain Model Concealer interface. This makes the User Interface Logic component dependent on both the Domain Model Concealer and the User Interface components. Finally, the Testing component allows testing of the Problem Domain Model component via the Domain Model Concealer component.

Sample Implementation. An application of the Domain Model Concealer pattern 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 4.6.

Implement the Problem Domain Model. The Problem Domain Model is as described above.

Implement the Domain Model Concealer. The Domain Model Concealer is implemented as the FinancialConcealer class. In the Financial History application, operations to add financial elements (add), delete financial elements (delete), and iterate over financial elements (createIterator) are needed. The Concealer should hold a reference (financialHistory) to the object in the Problem Domain Model of which it is currently showing the state. Furthermore, methods to set the state of the Concealer (load) and to set the state of the Problem Domain Model through the Concealer (save) are needed.

Implement the User Interface and User Interface Logic Components. The User Interface component (InputUI) can be implemented, e.g., by using a standard user interface toolkit. The component contains the declarations and instantiations of the widgets and corresponds to what may be generated by a user interface builder. The User Interface Logic component (InputUILogic) implements the actual functionality of the User Interface by subscribing to events in the User Interface. To communicate with the Problem Domain Model it uses the associated FinancialConcealer.

Optionally implement the Testing Component. A Testing component can be created to systematically test the Problem Domain Model via the Domain Model Concealer.

Consequences. This architectural pattern has both benefits and liabilities.

Benefits:

- Separation of the parts that require domain knowledge and the parts that require user interface toolkit knowledge is supported.

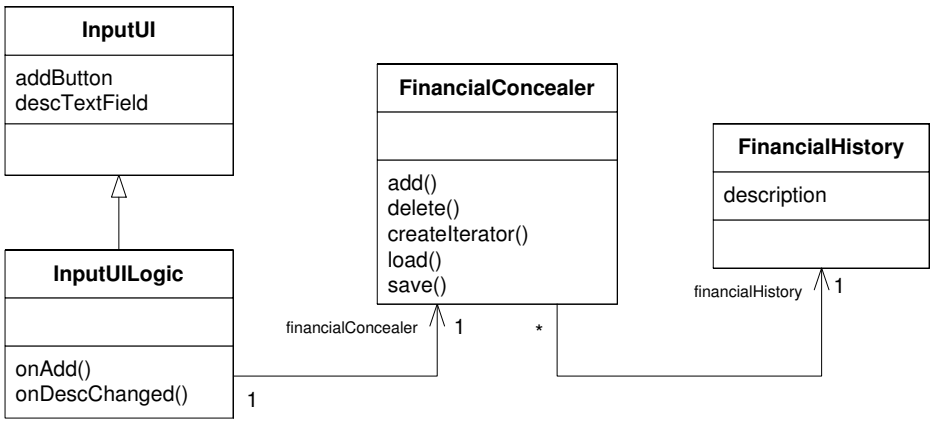


Figure 4.6: Class diagram for the Financial History Application

- Change in technology is supported, since the User Interface and User Interface Logic components can be substituted by new components when the user interface toolkit changes.

- Testing of the Problem Domain Model and the Domain Model Concealer is facilitated.

- The User Interface and User Interface Logic components are shielded from changes in the Problem Domain Model since they do not interface with it directly.

Liabilities:

- As the Problem Domain Model changes, either more work is involved in mapping between the Problem Domain Model and the Domain Model Concealer, or the Domain Model Concealer must be changed, which may require changes to the User Interface Logic.

- Multiple view consistency is not accommodated. The Observer pattern (Gamma et al., 1995) can provide this by having the Domain Model Concealer as subject and several User Interface Logic components as Observers.

- Depending on the nature of the changes, frequent changes to the User Interface also imply frequent changes to the User Interface Logic component.

4.1.3.4 Application Moderator

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 so that the user interface functionality is separated from the problem domain related functionality and so 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, and 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 4.7.

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, such as a button press, that occur in the user interface. 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 appropriate event members.

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 by 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. An application of the Application Moderator pattern 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 4.8.

Implement the Problem Domain Model. The Problem Domain Model is shown

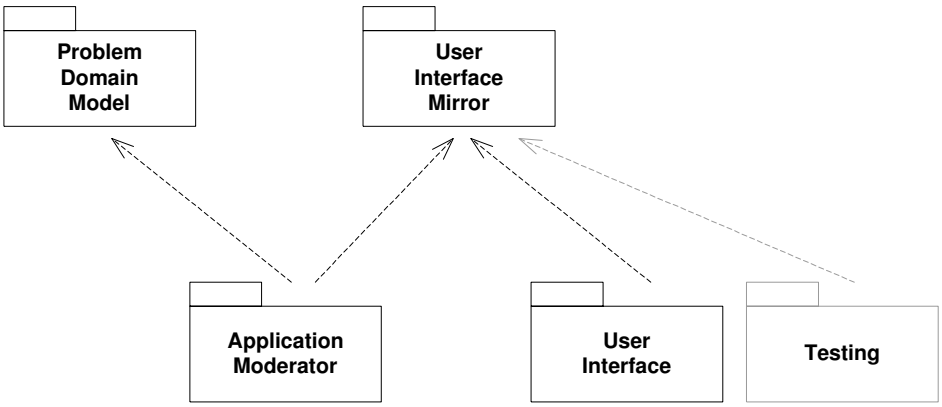


Figure 4.7: Overall Structure of the Application Moderator Pattern

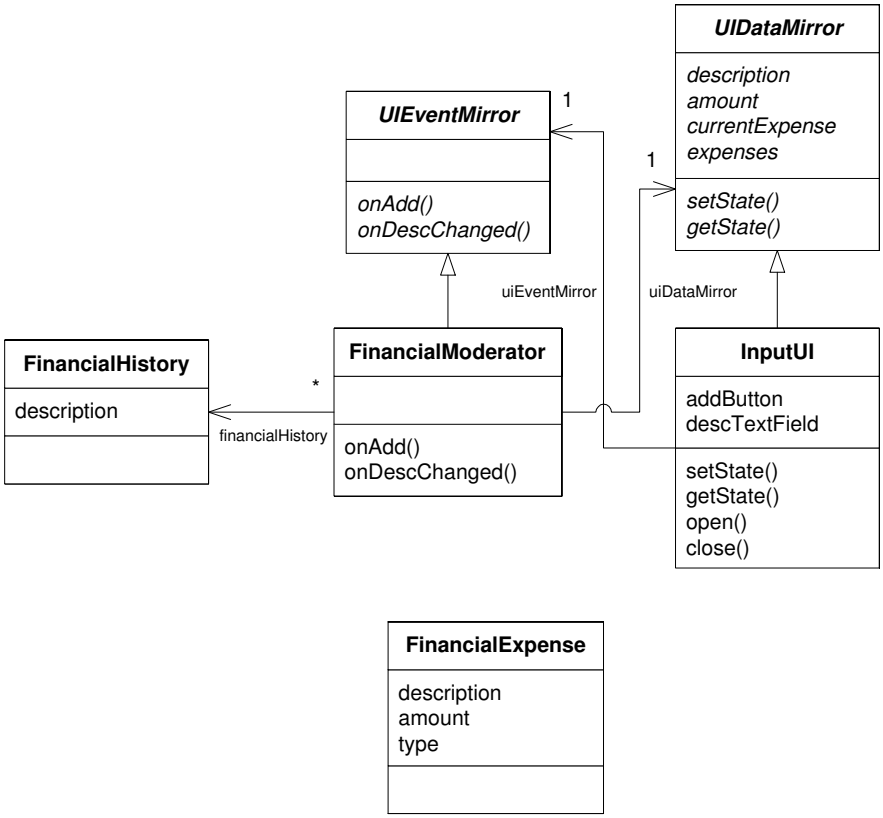


Figure 4.8: Class Diagram for the Financial History Application and FinancialExpense Data Interface

in Figure 4.4.

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.

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 4.8) 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 9). 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 UIDataMirror class. The event members of the User Interface Mirror are implemented as a number of abstract methods on the UIEvent-Mirror 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.

Refine the User Interface component. The User Interface component is refined to implement the setState and getState operations and to call the event methods in the User 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 UIEvent-Mirror which is associated through the User Interface component's uiEventMirror association.

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 by 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.

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 (i.e., in UIEventMirror) and then tests the state of the data members (i.e., 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, data conversion, and data replication.

4.1.3.5 Pattern Discussion

Known Uses. Three-tier client-server architectures (Tsichiritzis and Klug, 1978) divide interactive applications into three tiers: *Database*, *Domain*, and *Client* tier, often according to "External", "Conceptual", and "Internal" database schemas. In an interactive application with a thin client, the representation of the domain on the client in many ways resemble a Domain Model Concealer, and the Domain and Database tiers resemble the Problem Domain Model in the Problem Domain Model Concealer pattern. Fowler (Fowler, 1997) discusses an "Application Facade" architectural model in a non-pattern format. In this architecture, the Application Facade acts as a Domain Model Concealer and is used in the same way as in the Domain Model Concealer pattern. (Hansen and Thomsen, 1999) discusses another use of the pattern based on the Webvise-related project mentioned in section 4.1.3.

The Devise Open Hypermedia System (Grønbæk et al., 1994) uses the Application Moderator pattern extensively to divide user interface and the hypermedia data model in the client (runtime layer) of the system. Each dialogue in the system communicates with the non-client layers through a mirror of the hypermedia data model as in the Application Moderator pattern. In Microsoft Visual C++ , the Document/View architecture uses the Application Moderator to separate the user interface (View) and the domain model (Document). The class wizard of the development environment generates implementations of the setState and getState methods in what corresponds to the User Interface Mirror. The Dragon Project represents a third application of the pattern as discussed in section 4.1.3.

`http://msdn.microsoft.com/visualc/`

Relations to Other Patterns. A large number of architectures for interactive systems have been proposed. Examples include Seeheim (Pfaff, 1985), Model-View-Controller (MVC (Krasner and Pope, 1988)), Presentation-Abstraction-Control (PAC (Coutaz, 1987)), Arch/Slinky (Bass et al., 1992), and PAC* (Calvary et al., 1997). However, few of these have been treated in a pattern form. The most well known patterns are probably the MVC and the PAC patterns described in (Buschmann et al., 1996).

The Application Moderator pattern complements the MVC pattern. The MVC pattern is mostly concerned with obtaining multiple view consistency. Whereas it decouples the Model from the Views and Controllers via the Observer, it allows the Views and Controllers direct access to the Model. The Application Moderator pattern does not allow this.

The PAC pattern differs considerably from the other patterns. PAC has as its dividing principle a vertical split of the application so that it is divided into separate agents according to divisions found in the problem domain. Each of the agents contains their own user interface and functionality.

A main difference between the Domain Model Concealer and Application Moderator lies in what is decoupled: the Application Moderator architecture adds an interface layer to the user interface, thereby shielding the clients of the user interface from changes to it. The Domain Model Concealer architecture does almost the opposite: it adds an interface to the Problem Domain Model and in that way hides changes from its dependants.

4.2 Representing Experimental Modelling — a Notational View

This section proposes an extension to the UML based on a number of observed problems with the current specification of the UML compared to actual modelling practice as presented in Chapter 3. In order to be able to present the extension, we present the architecture of the UML in more detail than did Section 2.2.2.1:

4.2.1 Architecture of the UML

The current UML 1.4 specification (OMG, 2001b) defines a four-level layered architecture for the UML as shown in Table 4.1.

An instance of objects on level 3 (a model) was shown in Section 2.2.2.1. The language of level 1 (meta-metamodels) is the OMG Meta Object Facility (MOF; (OMG, 2001a)). Figure 4.9 shows a part of the UML metamodel from level 2 related to the definition of classes and associations.

It specifies, e.g., that an Association has two or more AssociationEnds each of which are connected to precisely one Classifier. This specifies that in a valid UML model, an Association *must* connect two or more Classifiers. The diagram part of the UML metamodel cannot fully specify all constraints between metamodel classes, so the diagrams are combined with "Object Constraint Language's" (OCL) wellformedness rules (Warmer and Kleppe, 1999). Part of the rules for an AssociationEnd are, e.g., that:

> [1] The AssociationEnds must have a unique name within the Association
>
> ```
> self.allConnections->forAll( r1, r2 |
> ```

Layer	*Description*	*Example*
1. Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	`MetaClass`, `MetaAttribute`, `MetaOperation`
2. Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	`Class`, `Attribute`, `Operation`, `Component`
3. Model	An instance of a metamodel. Defines a language to describe an information domain.	`StockShare`, `askPrice`, `sellLimitOrder`, `StockQuoteServer`
4. User objects (user data)	An instance of a model. Defines a specific information domain.	`Acme_SW_Share`, `654.56`, `sell_limit_order`, `Stock_Quote_Svr`

Table 4.1: The Four Layer Metamodelling Architecture of the UML (from (OMG, 2001b, page 2-4))

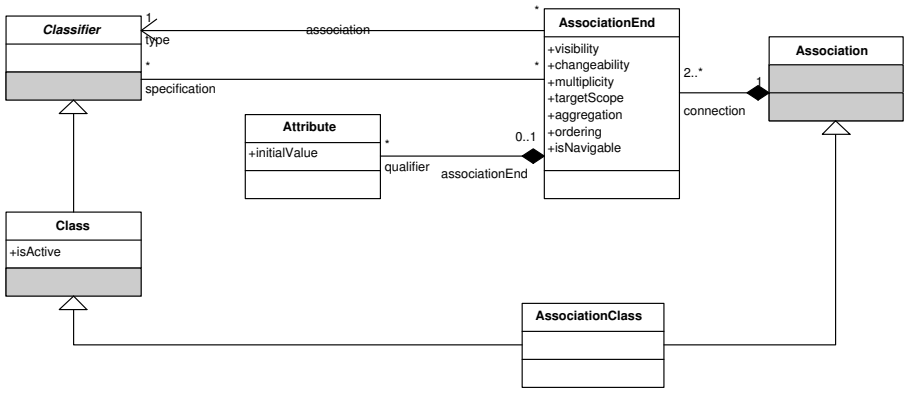


Figure 4.9: UML Metamodel Excerpt

```
        r1.name = r2.name implies r1 = r2 )
```

[2] At most one AssociationEnd may be an aggregation or composition.

```
self.allConnections->select(aggregation <#none)
     ->size <= 1
```

(OMG, 2001b, page 2-52).

Finally, a verbal "Detailed Semantics" section completes the definition of the metamodel elements:

> An association may represent an aggregation; that is, a whole/part relationship. In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts (OMG, 2001b, page 2-66).

As noted in Section 2.2.2.1, the UML specification does not formalize the notational or graphical part of a UML model, but contains a "Notation Guide" which defines the notational part verbally such as in:

> A binary association is drawn as a solid path connecting two classifier symbols (both ends may be connected to the same classifier, but the two ends are distinct) (OMG, 2001b, page 3-68).

This verbal definition is complemented by examples in the form of drawings of the defined graphical elements.

However, as part of UML 2.0, the diagram part of the UML is being formalized (OMG, 2000a) among others to enable UML-enabled tools to exchange diagram information. The rest of this section will thus assume the existence of a formalization of the diagram part of the UML in some form related to the formalization of the model part. For current lack of wording we will call this formalization the UML *meta-diagram* while we will call the model part of the metamodel the UML *meta-model*.

4.2.2 Representing Uncertainty: Introducing Levels of Restrictions

As described above, the UML specification consists of one metamodel with a connected semantics and a connected set of OCL constraints. This fixes the use of the UML on one level of "restriction": it is only possible to create UML models with

one, specific level of expressiveness in diagrams and models. Classes, e.g., *must* be drawn as shown to the right in Figure 4.10. Even though the less formal-looking representations to the left in the figure look very much like UML classes, they are not. Moreover, even though the class to the right does look like a UML class, it might violate the wellformedness rule that states that all model elements within a namespace must have a unique name in that namespace (OMG, 2001b, page 2-63).

Figure 4.10: Possible UML Class Representations ("Informal", "Semiformal", "Formal")

When working creatively in general, it is often an advantage to have less precision and more expressiveness when working with diagrams (Goel, 1995; Gross and Do, 1996; Hearst et al., 1989). Having more imprecise diagramming notations may also be a way to involve multiple perspectives in diagramming and to keep diagramming open-ended. Thus, these two, simpe examples show how it might be beneficial to work on different, more *expressive* meta-diagram and meta-model levels than the current, default level of restriction. The instances of Chapter 3 offer further empirical examples and arguments for this.

Moreover, as noted by (Bunse and Atkinson, 1999), working with more *restrictive* meta-models (and meta-diagrams) might be necessary. Bunse and Atkinson introduce the *Normal Object Form* (NOF). The NOF is a subset of the UML especially tuned towards reverse engineering and code generation. In doing this, they distinguish beween *refinement* and *transformation* of UML models. Refinement is used to gradually make a UML model more precise in order for it to be transformed to and from object-oriented language code. An example of a transformation of a normal UML model that is needed before code generation can be done in a precise way is the specification of exactly how associations should be implemented. Bunse and Atkinson support our claim that there should be multiple versions of the UML meta-model, but it is not concerned with meta-diagrams and only aims at refining UML models to conform to a more restrictive meta-model. In addition to this, a default, or lowest, level of restriction of the UML should be more expressive than the current UML specification in order for more flexible modelling to be directly supported. And even more: different parts of a UML models need to be able to be in different levels of restriction in order to allow for iterative, incremental, and parallel modelling.

Section 4.2.2.1 will give a precise meaning to this notion of restrictiveness and expressiveness and present a framework for how it may be represented as part of

the UML specification. Table 4.2 summarizes how different levels of restriction may be utilized.

diagram \ model	*expressive*	*restrictive*
expressive	freehand drawing	freehand icons for elements
restrictive	structured drawing	formal, unified presentation and representation

Table 4.2: Uses of Expressiveness and Restriction in Diagrams and Models

4.2.2.1 Incorporating Levels of Restriction into the UML

Consider the set of metamodels that can be described by the constructs used in the UML specification: given two metamodels A and B, A is more *restrictive* than B if any model conforming to A is also a model conforming to B. This restriction relation imposes a partial ordering on the set of metamodels. To incorporate restrictions into the UML, we need to:

- define how metamodels of different levels of restriction can and may be constructed and
- define a "bottom" least restrictive/most expressive meta-model and meta-diagram

The previous section argued that this "bottom" metamodel should be more expressive than the current meamodel. For actually defining levels of restriction in the UML, several options are available:

1. create an expressive meta-diagram and meta-model and leave it up to users and tool providers to restrict if needed,
2. create several levels of meta-diagrams and meta-models and allow these to be combined independently, or
3. create a base meta-diagram and meta-model and use wellformedness rules to define restrictions

The first option is impractical since the very reason for the creation of the UML was standardization of modelling languages across usage and tools. But given the current UML specification it is the only practical option in order to demonstrate the usefulness of multiple levels of restrictions. Chapter 5 will discuss a tool that incorporates levels of restrictions in this way.

In (Damm et al., 2000c), we argue for the second option in contrast to the first. However, since levels need to be combined within one model, and since the last option would provide the least incompatibilities with the current UML

specification, it is to be prefferred: the default set of of rules could correspond to the current meta-model, and even though, e.g., multiplicities would be different in the meta-model, the wellformedness would specify that a valid model on that level of restriction would look exactly like a current, valid model.

As an example of how this might work, consider Figure 4.11, which shows a more expressive version of Figure 4.9. It allows associations to be "incomplete", i.e., connected to less than two classifiers.

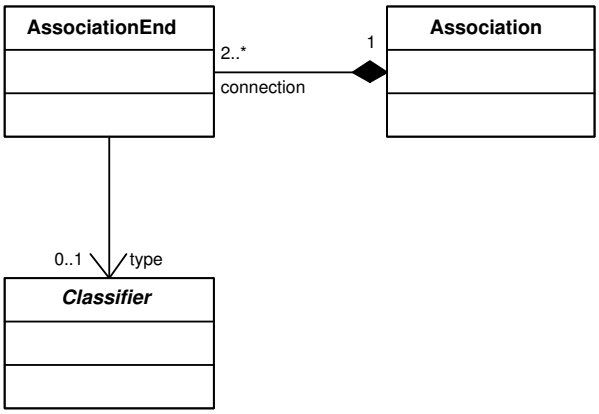


Figure 4.11: UML Metamodel Excerpt — Modified Version

We may also make the wellformedness criteria for association ends, presented above, less restrictive by omitting some of the rules.

4.3 Summary

This chapter has shown how modelling processes, types of modelling (in the form of patterns), and notations for modelling may help in addressing the challenges outlined in Chapter 3. In particular, we have tried to address the challenges of uncertainty, iterative, incremental, and parallel development.

Chapter 5

A Tool for Experimental Modelling

In this chapter, we focus on a specific test bed for tool support for experimental modelling: the *Knight* tool. In particular, we discuss how it tries to address the challenges of multiple practices, uncertainty, heterogeneity, and open-endedness in experimental modelling as formulated in Chapter 3. Since the Knight Tool *is* a test bed, all functionality that is discussed has been implemented and evaluated at some point in time, but not all functionality is necessarily preserved in the current version.

The material in this chapter is based in part on (Damm et al., 2000a; Damm et al., 2000b; Damm et al., 2000c; Damm et al., 2000d; Hansen and Ratzer, 2002).

5.1 Rationale for the Knight Tool

The initial design goal for the Knight Tool was to support co-located collaborative modelling using the UML. Based on experience from the Dragon Project, we noted that tool support for, among others, collaborative modelling activities were almost non-existent in traditional CASE tools. The empirical studies that grounded these observations were discussed in Chapter 3. In particular, the studies presented in Section 3.1.3.2 about COT and Section 3.1.4 about Mjølner, had as a major goal to uncover problems with current tool support for modelling.

A second strand of foundation work for the Knight Tool was an investigation of current technologies. We considered two main categories of tools for modelling:

- *Computer-supported diagramming tools*. General, structured drawing programs such as Microsoft Visio or Micrografx Designer support drawing and editing of UML diagrams either in the form of general box and line diagrams, by the use of stencils, or by a similar mechanism. CASE tools such as Rational Rose and Together ControlCenter support a number of software

`http://www.microsoft.com/office/visio/`
`http://www.micrografx.com/`

engineering activities such as forward and reverse engineering in addition to diagramming.

http://www.rational.com/

http://www.togethersoft.com

- *Physical diagramming tools*. Whiteboards and paper support freehand diagramming, also of the UML.

Table 5.1 summarizes some advantages and disadvantages of these two categories of tools:

	CASE Tools	Diagramming Tools	Whiteboards	Paper
Syntax and semantics checking	+	÷	÷	÷
Software engineering support	+	÷	÷	÷
Integration with computational environment	+	–	÷	÷
Process support	+	–	–	–
Structured diagramming support	+	+	÷	÷
Usability	÷	–	+	+
Learnability	÷	–	+	+
Co-located collaboration support	÷	÷	+	–
Extensibility	÷	+	+	+
Informal and incomplete diagrams	÷	+	+	+
Mobility	÷	–	÷	+

Table 5.1: Aspects of Tools for Modelling (+: yes/good, –: maybe/neutral, ÷: no/bad)

In connection to CASE tools, several researches have reported on the adoption, or rather lack of adoption, of CASE tools in development organizations (Aaen et al., 1992; Kemerer, 1992; Iivari, 1996; Lending and Chervany, 1998; Jarzabek and Huang, 1998; Scott et al., 2000; Sharma and Rai, 2000).

In their recent, national US survey, Sharma and Rai found that approximately 13% of the surveyed companies had adopted CASE tools (Sharma and Rai, 2000). Adopters used CASE tools for around 60% of the tasks that could be supported by the CASE tools. Facilities for creating diagrams and models ("representation" in the study), was, with 95% adoption, by far the task for which CASE tools had been most highly adopted. Lending and Chervany (Lending and Chervany, 1998) report the same usage characteristics of CASE tools: only a small amount of the

functionality of CASE tools was used and the most used functionality was the functionality for diagramming and modelling. The only functionality that was used at least "sometimes" was functionality for semantic analyses of diagrams. These two studies suggest that it is important to support diagramming well in current CASE tools.

Iivari (Iivari, 1996) confirms this and also concludes that a high degree of CASE tool usage increases productivity, but if developers are able to choose whether they want to use CASE tools, they prefer *not* to use CASE tools. A major reason for this was that developers perceived CASE tools as having high complexity, and Iivari thus points to that reducing the perceived complexity of CASE tools may be profitable. Jarzabek and Huang concur and conclude that current CASE tools are far too foucsed on hard aspects of software development (Jarzabek and Huang, 1998). To be more successful, "CASE tools must be more user-oriented, and support creative problem-solving aspects of software development" in addition to supporting harder, software engineering-related aspects of software development. Also, current tools are too method-oriented and should allow developers to work more freely and naturally and preferably "at the level of application end users".

In contrast to CASE tools, blackboards, whiteboards, and paper are extremely effective in creative and fluent problem-solving cross-cutting multiple perspective through their versatile, unconstrained, and ubiquitous character. The use and effectiveness of whiteboards has been studied in such diverse settings as meeting room (Moran et al., 1996), class rooms (Abowd et al., 1996), and personal offices (Mynatt, 1999). Similarly, the use of paper has been studied in such diverse settings as construction drawing (Mackay et al., 1993), flight control (Hughes et al., 1992), and video prototyping (Mackay and Pagani, 1994).

The obvious idea is then to try to combine the benefits of computerized (CASE) tools and whiteboards or paper. The user studies, technology comparisons, and this insight led to the following original design criteria for the Knight tool:

- *Provide a direct and fluid interaction.* The interaction with the tool should never be the focus when using it and the tool should have a low threshold of initial use.

- *Support cooperative work.* Software developers should be able to work together collaboratively using the tool. Domain experts should be able to use the tool in conjunction with developers.

- *Integrate formal, informal, and incomplete elements.* Beside formal UML elements, informal drawings and incomplete UML elements should be incorporated and integrated in order to support fluid modelling processes.

- *Integrate with development environment.* Additional software engineering capabilities such as forward and reverse engineering and documentation is

essential in modelling. Thus, integration with existing CASE tool functionality is needed.

- *Support large models.* Users should not be restrained by the size of models. In addition, filtering capabilities should be available.

The design criteria led to the implementation discussed in the next section.

5.2 Design and Implementation of the Knight Tool

This section presents the implementation of the Knight tool from a use and technical perspective: the interaction with the tool is presented with particular emphasis on the support for levels of restriction in the UML (see Section 4.2.2) and the technical implementation is discussed with emphasis on integration with other tools.

5.2.1 Interaction

The Knight tool implements a "whiteboard" metaphor of interaction: the workspace on which models are created is a potentially infinite, blank surface (Figure 5.1). Element creation, deletion, and updating is done primarily using gesture input:

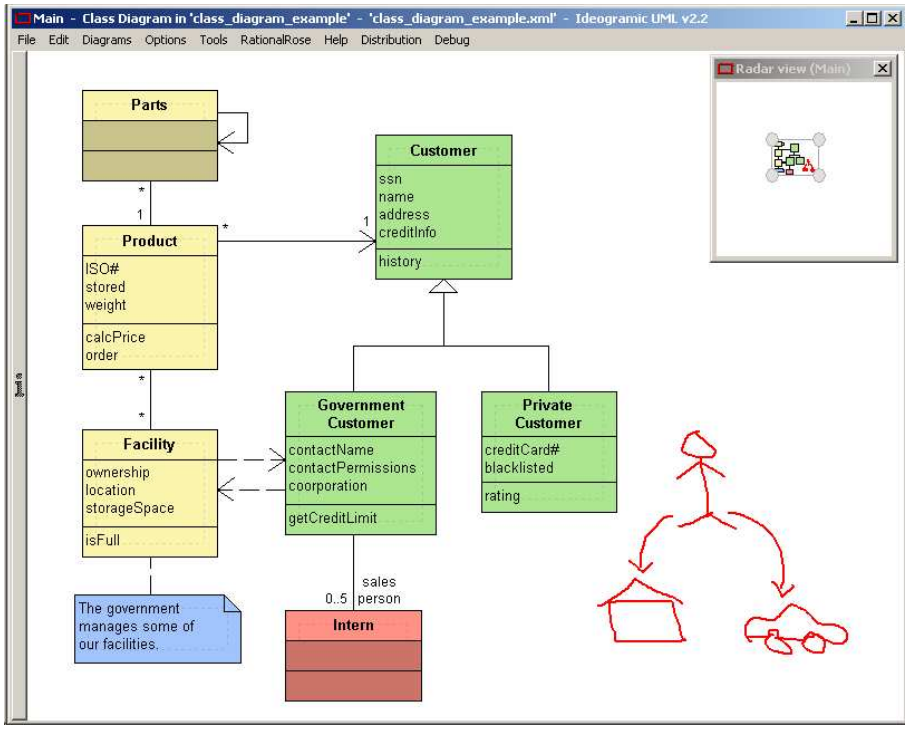


Figure 5.1: Knight User Interface

5.2.1.1 Gestural Interaction

The Knight tool has two primary modes: a freehand mode in which all input strokes are uninterpreted, and a gesture mode in which input strokes are interpreted as gestures. The freehand mode works well for all types of users and the gesture mode works well for users with some knowledge of the UML (see Section 5.3, page 89ff). In this context, gestures are marks drawn on the workspace and subsequently recognized by a computer tool (Buxton, 1986). In the Knight tool, the most frequently used gestures bear a direct resemblance to what would have been drawn on an ordinary whiteboard when drawing UML diagrams. Figure 5.2 shows examples of this: at the top it is shown what the user draws, and at the bottom it is shown how the Knight tool represents the user input after it has been recognized. By using gestural interaction, the interaction becomes fast and direct: a command and its parameter are specified by a gesture that resembles the intended result of the command. Two potential problems with gesture recognition are that only a limited

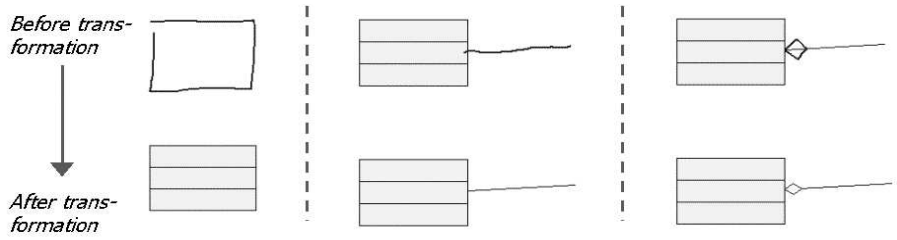


Figure 5.2: Gesture Recognition in the Knight Tool

number of different gestures can be recognized and that no feedback is given when a gesture is drawn. To reduce the number of needed gestures, we use *compound gestures* (Landay and Myers, 1995): if a diamond gesture is drawn close to an association, it changes that association to an aggregation (Figure 5.2, right). To provide incremental feedback, we use *eager recognition* (Rubine, 1991) for e.g. moving elements: as the user is drawing, the gesture is incrementally recognized, and if it can be classified as a move gesture, a move is initiated, and the rest of the user's gesture is interpreted as the trajectory of that move.

Gestures are also used for less frequently used commands in combination with context-dependent, hierarchical pie menus (Figure 5.3) in order to make pie menus work as *marking menus* (Kurtenbach, 1994): making a short stroke in the direction of a desired command on a pie menu invokes that command. For expert users, this provides a highly efficient way of invoking commands. The gesture recognizer is an implementation of Rubine's algorithm (Rubine, 1991). The algorithm uses simple pattern recognition to recognize single-stroke, two-dimensional gestures and consists of two phases:

Training. Each type, or *class*, of gesture is specified by a set of examples. For each gesture example, a *feature vector* is calculated. This feature vector

contains 13 features of the gesture including start angle, length, and total angle traversed for the gesture. Based on the feature vectors for the examples, an average feature vector is calculated for the entire gesture class.

Recognition. During recognition, the feature vector of the user's gesture is calculated and compared to the feature vectors of the gesture classes created during training. The most resembling gesture class is chosen, and a probability whether the guess was right is returned.

Since the gesture recognition is based on statistical analyses, the gesture class output from the algorithm, in response to a gesture made by the user, may be the wrong gesture class. However, since the Knight tool supports unlimited undo and redo, the commands implied by a recognized gesture are always executed since the user may easily repair a wrong guess made by the algorithm.

The use of Rubine's algorithm is complicated slightly by the fact that features of a gesture class are required to be normally distributed. This means that a single gesture class cannot cater for, e.g., left-handed and right-handed users who are typically drawing figures in different directions since this would make the feature space be multi-modally distributed. To do this, we create a basic gesture class, such as a rectangle drawn in one specific way, and rotate and mirror those to create new gesture classes. In this way, all our 74 gesture classes on class diagrams alone can be created from only 14 base gesture classes. Experience and evaluations show that this approach works well: we made different people create an example set of more than 9000 strokes, asked these people to manually classify them, and tested their classification against our recognizer. The current recognition rate is above 96%. Section 5.3 discusses results from qualitative evaluations.

5.2.1.2 Input Devices

The Knight tool is designed to work on a range of input and output devices, but since a main goal was to design a tool for collaboration, we have focussed on mak-

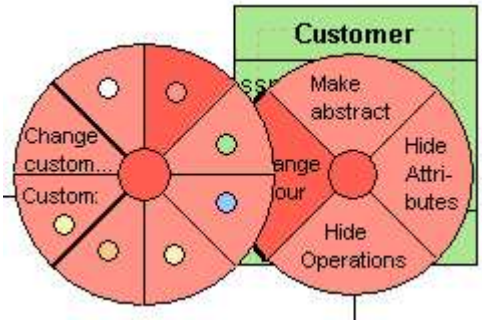


Figure 5.3: Pie Menus in the Knight Tool

ing the interaction work well on *electronic whiteboards*. Electronic whiteboards (Figure 5.4) provide a large screen on which users draw with pens much like on ordinary whiteboards. However, the pens are non-marking: a computer receives the pen strokes as input and a projector provides output. The electronic whiteboard

Figure 5.4: Electronic Whiteboards

to the left in Figure 5.4 is a "SMARTBoard" which has a touch-sensitive screen that is operated with special markers. The "Mimio" to the right can be mounted on any ordinary whiteboard and in combination with a projector turn this into an electronic whiteboard.

`http://www.smarttech.com`
`http://www.mimio.com`

Two aspects of electronic whiteboards have to be handled in order to get an effective interaction, namely that they have *limited input capabilities*, and a *large physical size*.

The main input device is a pen and, generally, the only input the pen can give is whether or not it is pressed down, and if it is pressed down, where it is. This means that, conceptually, the only input an electronic whiteboard can give is that of a single-button mouse with no mouse movement dection capabilities. In the Knight tool, this means, e.g., that pie menus must be activated by pressing the pen and holding it down for a small period of time. A related problem is that of text input as discussed in Section 5.2.1.4.

Electronic whiteboards are much larger than ordinary desktop monitors and in combination with the absolute input device this raises some issues: it is, e.g., physically impossible to stand at the right side of an electronic whiteboard and reach the left side of the electronic whiteboard with the pen. In the Knight tool this is handled by using gestural input and marking menus instead of ordinary toolbars and pull-down menus: gestures can always be drawn at the user's physical position. Replicating widgets is another example of handling the large size of the input device:

5.2.1.3 Handling a Large Workspace

The workspace of the Knight tool should be potentially infinite. This is handled by radar windows shown in Figure 5.5. The radar window can be resized to pro-

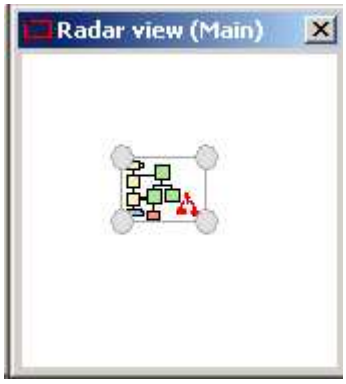


Figure 5.5: Radar Windows in the Knight Tool

vide a larger workspace, the view port of the radar window can be dragged to pan, and the handles on the viewport can be dragged in order to zoom. This means that multiple actions that are typically distributed to a number of widgets in an ordinary user interface is now collected in one widget, which is important on electronic whiteboards. Also, any number of radar windows can be opened so that it is possible, e.g., to have a radar window within reach on both sides of the electronic whiteboard.

5.2.1.4 Text Input

Apart from the ordinary keyboard that is generally connected to the computer that drives an electronic whiteboard, there is a need for text input mechanisms that can be used by those working at the electronic whiteboard. Figure 5.6 shows the text input possibilities we have experimented with in the Knight tool. In order of difficulty to in learning and effectiveness in operation, these are:

- A *virtual keyboard* (a) that is operated much like an ordinary keyboard but with "sticky keys" for, e.g., the shift key.

- *Cirrin* (b), a circular single-stroke keyboard on which the user draws a stroke through the letters of the word that the user is trying to input (Mankoff and Abowd, 1998).

- *Stylus-based gestures* (c) as on PDAs providing a predefined symbols for letters and numbers.

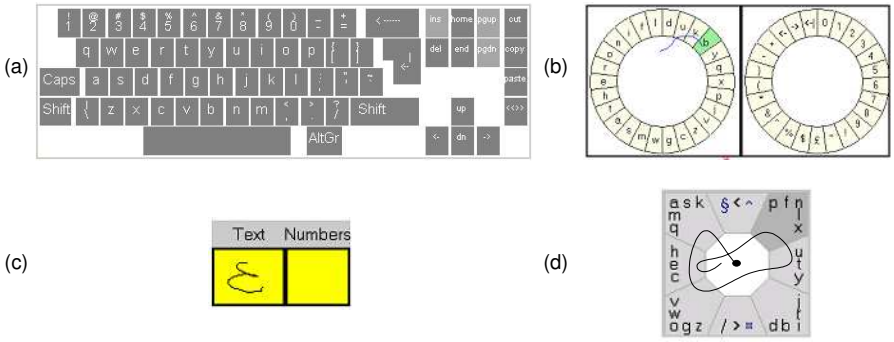


Figure 5.6: Text Input Possibilities in the Knight Tool

- *Quikwriting* (d) is another single-stroke keyboard (Perlin, 1998). The figure shows a stroke that results in the word "que": to write the "q", which is at the bottom of a field with five letters, the user makes a stroke to the field containing the "q", continues two fields down, and out in the centre. To write the "u", which is at the top of a field with three letters, the user draws a stroke to the field containing the "u", continues one field up, and out. Finally, to write the "e" which is at the middle of its field, the users draws a stroke to the field containing the "e" and continues out.

  In general, to write the middle letter in a field of letters, the user simply draws a stroke into the field and out of the same field. To draw the letter to the right of the middle of a middle, the user draws a stroke into the field, continues the stroke to the field to the right, and continues the stroke out of that field. And in a field with five letters, the user has two continue the stroke two fields from the field containing the five letters in order to write one of the letters on the outside of the field.

In the end, we settled for the virtual keyboard as text input for two main reasons: first, it is the text input method that is the easiest to learn. Secondly, creative modelling seldom involves extensive text input, and thus speed of text input may not be critical. In fact, one of our user studies showed that only approximately 25% of a meeting was spent on actually diagramming and only a minor part of that was spent on writing text (Damm et al., 2000a).

A longer-term wish is to include handwriting recognition when this technology matures.

5.2.2 Support for Levels of Restriction

We aim at supporting different levels of restriction in diagrams and models in the Knight tool in order to obtain open-endedness in diagrams and models. Since models may be at various levels of restriction, tools should support the mixing of

different levels of restrictions and transitions between these levels.

This section discusses the current support for this.

5.2.2.1 Different Levels of Restriction

The default level of restriction in the Knight tool is more expressive than the current UML specification: in our user studies (see Chapter 3) the default level of restriction in actual modelling was more expressive than that of the current metamodel. Section 5.2.2.2 discusses how the Knight tool supports the shift between levels of restriction.

For models, the increased expressiveness means, e.g., that two model elements (such as classes) can have the same name, or no name, even if they are in the same namespace, that relationships (such as associations) can be incomplete, and that model elements (such as state machines) do not need to be semantically connected as may be required in the current metamodel.

For diagrams, we found freeform drawings as an addition to UML diagram elements particularly useful. Also, we have focussed on preserving the look of diagrams: in CASE tools, elements are often required to *look* finished even though they are not (Figure 5.7) and, as discussed in Section 4.2.2, page 66, there is evidence that such formal representations hinder creativity.

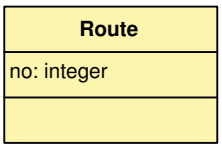


Figure 5.7: Unfinished UML Elements May Look Finished

Thus, in the Knight tool, an element can be displayed in three different ways:

- *Untransformed:* an element may look exactly like it is drawn even though it has been recognized, e.g., as a class.
- *Semi-transformed:* an element may be presented using a sketchy look.
- *Transformed:* an element may look as it is required by the UML specification.

Figure 5.8, left, shows the three different transformations in Knight on recognized classes. Figure 5.8, right, shows how an unfinished class may look indeed unfinished using semi-transformation. As detailed in Section 5.2.2.2, the different transformations can be mixed within a single model and even within a single diagram.

5.2.2.2 Shifting Between Levels of Restriction

There are three main ways in which shifting between levels of restriction can be supported by a tool:

- *Automated:* The tool may automatically, i.e., without intervention by the user, transform the model to a certain level of restriction.
- *Guided:* Upon request by the user, the tool may suggest different transformation that the user may choose among.
- *Manual:* The tool may offer funtionality for the user to change levels of restrictions without any special, automated support from the tool.

The Knight tool supports transformations of all three kinds to various degrees.

Initially, strokes are, conceptually, elements in a very expressive meta-model and -diagram. The default behaviour is to restrict those automatically to UML model elements with a standard UML presentation, making a restriction in model as well as diagram. When integrating with CASE tools (see Section 5.2.4, page 84), the needed level of restriction is the standard UML level, and thus an automatic transformation is also performed in this case. Freehand elements are, e.g., converted to comments and incomplete associations are connected to a hidden class.

Figure 5.9 shows an example of guided transformations: the user has asked Knight to do a post-processing on freehand elements and recognize these as UML elements. This may be useful, e.g., for collaborative sessions in which an ongoing recognition of elements would interfere. Another way of restricting models is to ask the Knight tool to mark elements that are illegal in a more restrictive meta-model. The tool will then highlight illegal elements, such as incomplete relationships, and leave it up to the user to decide what to do with these illegal elements.

In Figure 5.10, three examples of manual transformations are shown. In all examples, an illustrative freehand drawing, (a), representing the concept of a “route”,

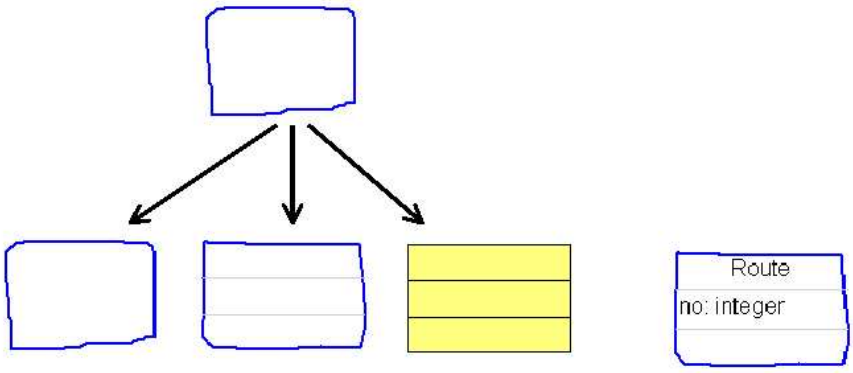


Figure 5.8: Different Class Transformation in Knight

has been drawn. In (b), the user has chosen to transform the drawing into a class since the drawing is no longer important. If the drawing is still important, the user may transform the drawing to a class and a relation to the drawing, in effect using the drawing as an icon for the class (shown in (c)). In (d), the user has chosen to use the drawing to document a more elaborate model of a route.

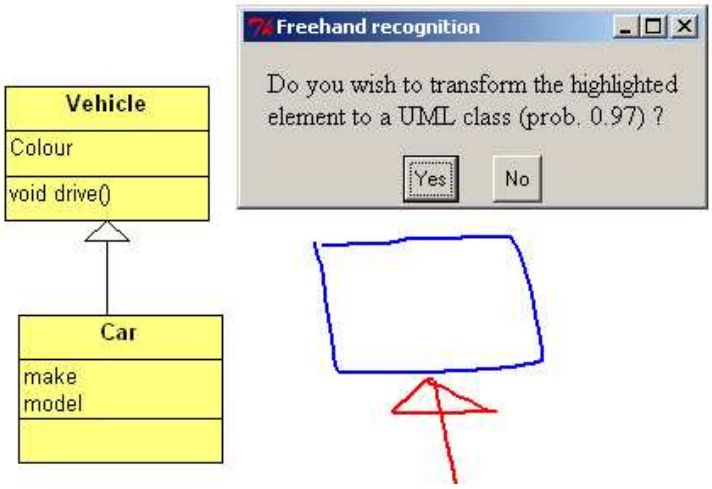


Figure 5.9: Guided Transformation of Freehand Elements in the Knight Tool

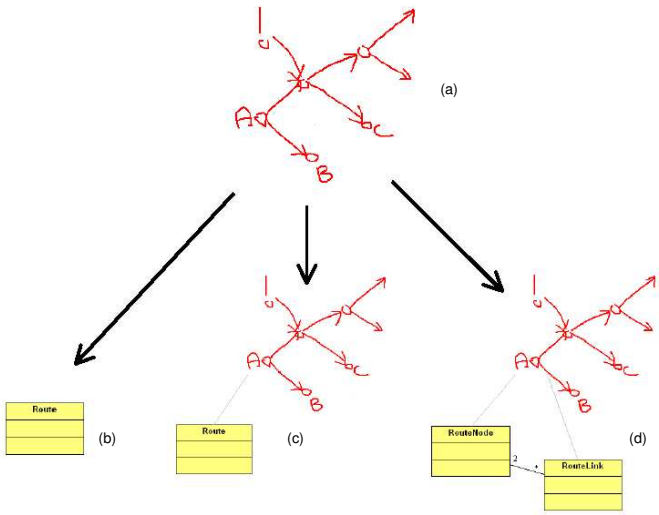


Figure 5.10: Restrictive Transformations of a Freehand Drawing

5.2.3 Software Architecture

Figure 5.11 shows a conceptual view of the software architecture of the Knight Tool. The central component follows the "Repository" architectural pattern (Shaw, 1996): it contains richly structured, centralized data with a number of computational components working on it. Concretely, it is a collection of diagrams and models: the models are instances of the UML 1.3 metamodel (OMG, 2000b) whereas the diagrams are implemented using a self-defined class structure since diagrams are not covered by the UML standard.

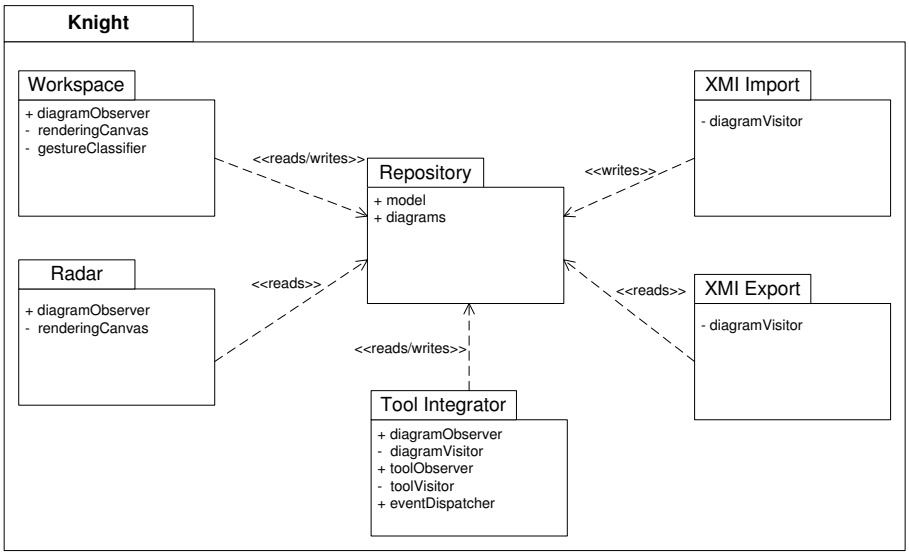


Figure 5.11: Software Architecture of the Knight Tool

Other components read from and write to the Repository. Reading is done as a result of creations, deletions, or changes to the model and is triggered by notifications. Notifications are implemented using the Observer pattern of (Gamma et al., 1995). The Radars are examples of Observers: when a diagram and model element is created, the Radars create corresponding visual representations of this model element. The Tool Integrator component is also an Observer. This component will be discussed in more detail in Section 5.2.4.

Writing is done using a combination of Command and Composite patterns (Gamma et al., 1995), providing unlimited undo/redo capabilities on any desired level of granularity.

The tool is implemented in a mix of [incr Tcl] (McLennan, 1993), an untyped, class-based, object-oriented extension of Tcl (Ousterhout, 1994), and C and C++ for performance-critical functionality such as gesture recognition.

5.2.4 Tool Integration

The primary reason for integrating the Knight tool with other tools is connected to our goal of building an environment that supports concrete activities in system development (cf. Section 2.4): one tool cannot support all system development activities equally well, and thus tool integration becomes important. A primary outcome of this is support for open-endedness in the combination of tools and a support for iteration through the collective work on material by different tools.

This section focuses on two integration efforts made in order to

- *integrate with CASE tools* in order to enhance the software engineering capabilities of the Knight tool (Section 5.2.4.1 and 5.2.4.2), and
- *integrate with the Topos tool* in order to address more of the challenges of Chapter 3 than otherwise possible (Section 5.2.4.3).

Tool integration may be viewed on an architectural level as a cooperation between high-level *components* interacting via high-level *connectors*. We can distinguish between *data* and *processing* components (cf. Section 4.1.1.1, page 50) with the processing components operating on the data components. In order to integrate, two processing components will need to work on the same logical data, which may, physically, be either shared or separate. In both cases, the processing components may either run simultaneously (synchronously) or asynchronously. Table 5.2 (inspired by Ellis et al.'s taxononmy of Computer-Supported Cooperative Work (Ellis et al., 1991)) summarizes our taxonomy of component collaboration on a common, logical core of data and gives examples: In the Knight case, the common, logical

Time \ Data	*Shared Data*	*Separate Data*
Asynchronous	File import/export	Merging software configuration management systems
Synchronous	Microsoft ActiveX with linking instead of embedding	Component technology interaction between applications

Table 5.2: Component Collaboration Taxonomy with Examples

core of data is data described by the UML metamodel. In the space described by the taxonomy, several possible integration scenarios are possible in the Knight case, e.g., by integrating

1. separate tools using a common interchange format (asynchronous, shared),
2. separate tools working on different aspects of the separate data (asynchronous, separate),
3. components collaborating through a well-defined interface to the common core of data (synchronous, shared), or

4. interaction between separate tools with separate data via component technology (synchronous, separate).

Section 5.2.4.1 considers the first scenario, Section 5.2.4.2 the fourth scenario, and Section 5.2.4.3 the third scenario. We have not investigated the second scenario.

5.2.4.1 XMI Integration with CASE Tools

We have implemented asynchronous integration with shared data in Knight through the OMG XML Metadata Interchange (XMI, (OMG, 2002)) standard that describes how to interchange UML models as XML documents. In fact, XMI is intended for describing interchange formats for all metamodels written in the OMG MOF metametamodel (see Section 4.2.1, page 64). Take the simple diagram in Figure 5.12 as an example in the UML case. Figure 5.13 shows part of the XMI that is generated by Knight to describe the "Car" model and diagram elements. Currently, Knight uses the XMI 1.0 format (OMG, 2000c).

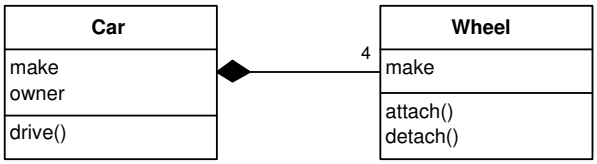


Figure 5.12: Simple Model for a Car

```
<Foundation.Core.Class xmi.id="idclass0-3C839BFF">
    <Foundation.Core.ModelElement.name>
         Car
    </Foundation.Core.ModelElement.name>
...
<Diagramming.Diagram.element>
    <Diagramming.DiagramElement
            xmi.id="idclassView0-3C839BFF">
        <Diagramming.DiagramElement.geometry>
             3420.0,3672.0,421.2,327.6,
        </Diagramming.DiagramElement.geometry>
```

Figure 5.13: Part of XMI generated from the Model in Figure 5.12

The XMI describing the model elements follow the UML metamodel faithfully as exemplified at the top of Figure 5.13. For diagram elements, the interchange is more problematic, however, since the current UML specification as discussed

above does not define the appearance of model elements. The bottom of Figure 5.13 (with linebreaks added for readability) actually shows a non-XMI extension that describes diagram information. The diagram format currently used by Knight is the same non-standard one as used by Rational Rose . Thus, the Knight tool is able to interchange model information with all CASE tools that implement XMI 1.0 for UML 1.3, but it is able to interchange models *and* diagrams only with Rational Rose.

http://www.rational.com/

This use of tool-specific extensions raises a deeper problem: how does one ensure that the changes one tool makes do not conflict with the extensions another tool has embedded in an XMI file? Since the semantics of extensions is, by definition, not covered by XMI, there is no general and simple solution to this problem. Nevertheless, it *would* be useful to have a time stamp included on XMI elements so that tools may at least discover if extensions they have written have become out-of-date.

A more practical problem is that a large number of CASE tools do support XMI interchange but that only a few of them can actually interoperate: UML and XMI are both evolving as standards (and both rely on MOF which is evolving as well) and tool producers may choose to actully support different versions of each of these standards.

5.2.4.2 Component Integration with CASE Tools

The XMI integration discussed in the previous section is impractical when different tools work on different aspects of data and frequent synchronization is needed; in this case, synchronous integration is useful. Usually, synchronous integration is component-based using technologies such as COM (Rogerson, 1997) or CORBA (Henning and Vinoski, 1999). We chose to use COM since this is the component technology used by most CASE tools.

A logical view of the architecture of Knight pertaining to tool integration is shown in Figure 5.14. The main component in this figure is the Tool Integrator component (cf. Section 5.11, page 83) that implements the integration of two or more tools, in this case the two tools “Tool A” and “Tool B”. Each tool contains separate but compatible data and is, in principle, required to have a COM interface.

The Tool Integrator enables a simple integration of tools that have compatible meta-models and a COM interface that enables the creation, manipulation, and deletion of model and diagram elements. Details can be found in (Damm et al., 2000d).

The architecture has proven effective for the integration of Knight and CASE tools: currently, we have experimented with integration with Rational Rose and Microgold WithClass , but we believe it will be adequate for the integration with other CASE tools as well.

http://www.microgold.com

5.2.4.3 Component Integration with Topos

A major challenge to experimental modelling is "heterogeneity" as presented in Section 3.2.4, page 45: sources to models are heterogeneous and products of modelling are heterogeneous. The *Topos* (formerly Manufaktur) tool allows for manipulation and maintenance of relationships among material in a 3D spatial environment (Büscher et al., 1999b; Büscher et al., 2000; Grønbæk et al., 2001). It integrates with standard applications in order to show thumbnails of documents mapped onto 3D objects that may be freely arranged in space (Figure 2.2, page 23). Using Topos, it is possible to create a richly structured, hyperlinked collection of material from a diverse set of applications. Further, one of the current focus areas for the development of Topos is its integration into an augmented reality working environment.

Integrating the Knight tool with Topos provides a useful setup for experimenting with heterogeneity. Consider the Dragon project: if such a tool had been available during the project, UML models could have been connected with, and documented by, bills of ladings, route plans, video and sound from ethnographical studies, and much more. The richly structured environment of Topos would allow for the co-location of models and material used for producing the models without impeding actual modelling work.

To enable such experimentations, we have built a prototype integration between Topos and Knight. A Topos workspace featuring integrated UML diagrams is shown in Figure 5.15. Including a Knight file in Topos results in the displaying of all diagrams in the file as different objects in Topos. Each object is updated as the corresponding diagram is updated through Knight, and double-clicking such an object will result in Knight displaying the corresponding diagram.

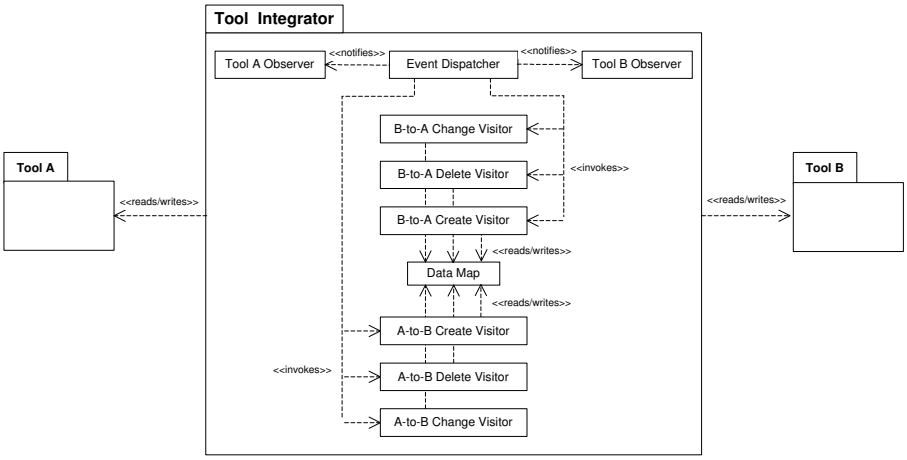


Figure 5.14: Knight Architecture for Component Integration

Technically, we created two new component interfaces to achieve the integration:

- *IToposClient*. Knight implements this interface. When the user includes a Knight XMI file in Topos, Topos calls this interface to obtain handles to and images of the objects in the file. The ToposClient is responsible for deciding what is objects, for giving handles to objects, for creating images for objects, and for re-opening objects.

- *IToposApplication*. Topos is the only instance of the IToposApplication interface. ToposClients use this interface to make callbacks when objects are changed. The ToposApplication is responsible for displaying objects and maintaining the handles to these.

Given a file, the only requirement from Topos is ordinarily that the application that handles this file is an ActiveX container. This enables Topos to get a persistent name for the file and to get an image of the document in the file. Our integration lies between the ordinary integration and an application-specific integration (which has been done with Webvise (Grønbæk et al., 1999)) in terms of coupling: the client application is required to contain an object that implements the IToposClient interface but is free to decide what the granularity and semantics of its objects are.

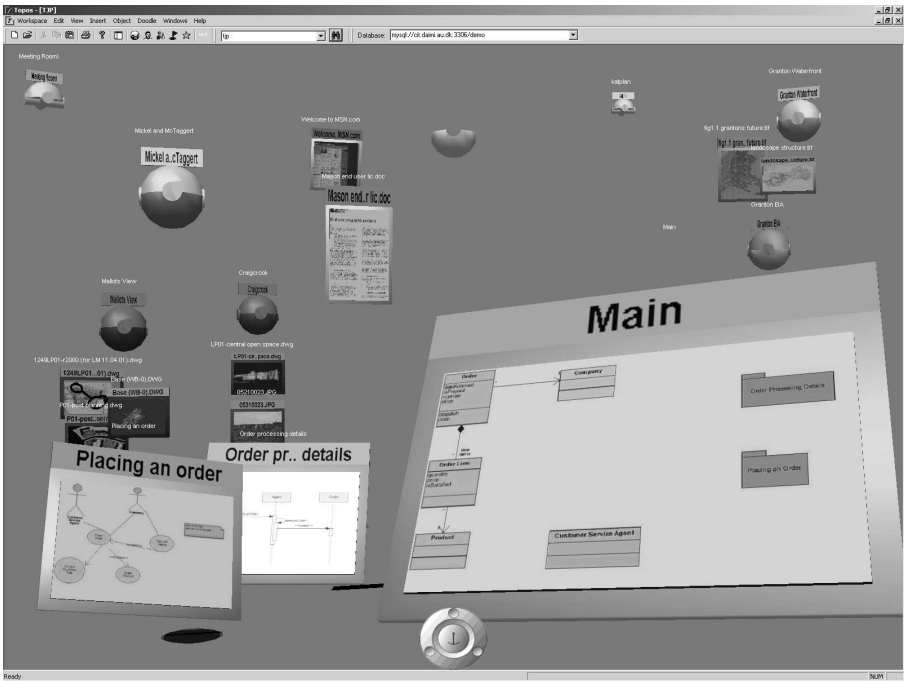


Figure 5.15: Example of Topos and Knight Integration

5.3 Evaluations of the Knight Tool

The Knight tool has been commercialized as "Ideogramic UML" by Ideogramic ApS. The commercialization and subsequent sale and use of Knight provides an anecdotal evaluation of its usefulness and appropriateness. In order to make grounded assesments, we have formally evaluated two important aspects of Knight: its use in ordinary software development and its use in computer science education. The first evaluation was done prior to the commercialization, and the second evaluation was done subsequent to the commercialization.

http://www.ideogramic.com

This section briefly reviews these evaluations. Some of the redesign suggestions from the evaluations will be discussed in Section 6.4, page 97.

5.3.1 Software Development

The first thorough evaluations of the Knight tool were done through two design sessions that involved a mixture of expert and novice modellers. The evaluations were done to validate the implementation of the original design criteria of Section 5.1, page 71.

5.3.1.1 Experimental Background

The subjects of the first design session were participants in the CPN2000 project (Beaudouin-Lafon et al., 2000; Beaudouin-Lafon and Lassen, 2000). The CPN2000 project experimented with novel interaction techniques for a Petri Net design tool. As part of the project, three designers had constructed different object-oriented designs for event handling, and certain interface elements had been constructed. We observed the designers use the Knight tool to integrate these designs. One of the designers was knowledgeable of UML and traditional CASE tools, the other two designers had little knowledge of the UML.

http://www.daimi.au.dk/CPnets/CPN2000

The subjects of the second evaluation session were participants in the DESARTE project (Büscher et al., 1999b; Büscher et al., 1999a). The DESARTE project was concerned with constructing electronic support environments for (landscape) architects. For one of the tools in this environment, Manufaktur, a conceptual model had been developed. We observed a meeting in which two participants used the Knight tool to restructure this model. Both designers had good knowledge of object-oriented modelling.

5.3.1.2 Evaluation Execution

In both cases, the designers received an initial, brief introduction to the interaction style of the Knight tool by a facilitator. During the evaluation, the facilitator provided tool-specific help if the designers asked for help. Furthermore, we videotaped the sessions and took notes.

5.3.1.3 Results

Both evaluations were encouraging: each session lasted about an hour and a half, and after the brief introduction, the designers were able to focus on modelling instead of on the tool or on the evaluation. The only exceptions were situations in which the tool actually crashed due to its prototype nature.

Figure 5.16 shows the model produced by the participants of the first session. In particular, we observed the following in relation to four of our five initial design

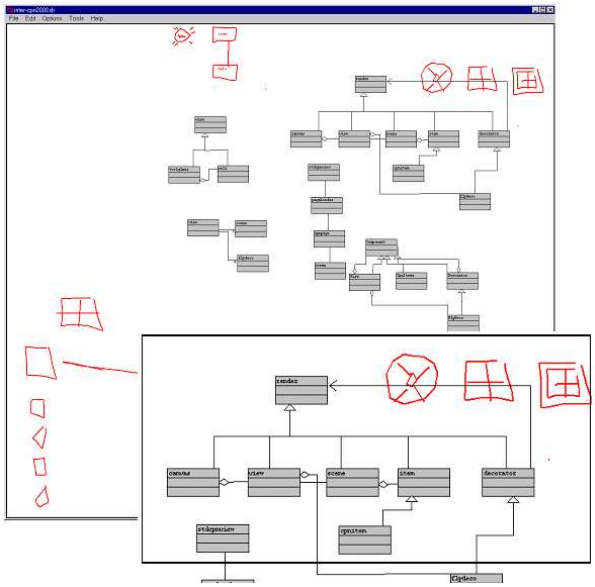


Figure 5.16: Model Produced in Evaluation Session

criteria (the last, integration with the development environment, was not considered):

Provision for a direct and fluid interaction. The tool showed to have a low threshold of initial use: after the short introduction, all participants were able to use the core functionality of the system. Some participants had problems drawing gestures due to lack of training or individual differences in how people draw (this has since been catered for). The marking menus were particularly hard to use.

Support for cooperative work. The electronic whiteboard worked well for collaboration support. The only major problem was that only one person could draw at a time. Nevertheless, both groups quickly developed the necessary coordination mechanisms for not interferring with each others work.

Integration of formal, informal, and incomplete elements. Freehand drawings were widely used and appreciated. However, the low resolution of the input on

the electronic whiteboard meant that participants had trouble particularly in writing freehand text. Incomplete UML elements were considered useful but not widely used.

Support for large models. As the models grew in size, the radar windows were used for zooming and panning efficiently. That the radar windows were the only way to navigate around the workspace was, however, problematic: to move a class from one corner of the diagram to another, one participant had to move, pan, and then move again.

This evaluation is discussed in more detail in (Damm et al., 2000a).

5.3.2 Computer Science Education

From the perspective of education, the UML has major problems:

- *Extent and Complexity:* With nine diagram types, more than 100 metamodel elements, and even more relationships between these, the UML is unwieldy. Moreover, a large number of these are necessary to learn and there is no concept of level of use in the UML. Thus, the UML is primarily suited for software development experts and problematic to use in teaching and learning object-oriented modelling (Astrachan, 2001; Raner, 2001).
- *Usability:* As discussed in this chapter and Chapter 3, the UML has no provision for many commonly occuring modelling practices: the UML provides one, formal and fixed, view of what a model is regardless of the modelling situation.

We wanted to evaluate to what degree the Knight tool could help with these problems in a teaching and learning situation and in particular when used in a peer group.

This evaluation is discussed in more detail in (Hansen and Ratzer, 2002).

5.3.2.1 Experimental Background

We studied participants in a second-year computer science course on object-oriented programming at the University of Aarhus doing a project assignment. Part of this project assignment was to create a problem domain model for a system for archiving, browsing, and replaying music files. We studied four project groups doing this.

The partipants had all attended a one-hour lecture on UML in which the Knight tool had been presented through a video. The lecture covered the basics of the UML with a particular focus on its relation to object-oriented programming languages.

Table 5.3 summarizes the experience the participants had with techniques relevant to the evaluation.

	Programming	*Object-Orientation*	*UML*	*Gestures*
Never	0%	0%	58%	42%
A few times	25%	17%	33%	42%
Some	33%	58%	8%	N/A%
Frequent	42%	25%	0%	17%

Table 5.3: Background Information on Frequency of Use of Techniques for Test Subjects

5.3.2.2 Evaluation Execution

Following a summary of the lecture on UML given to the course, the students started with a brief trial round in which they tried to become familiar with the uncommon interaction techniques used in the Knight tool. Subsequently, the groups used the tool for creating a domain model without us providing specific tasks to solve. In this way, with the exception of the Knight tool, we tried to make the evaluation situation resemble how the groups would typically model (Figure 5.17. Two

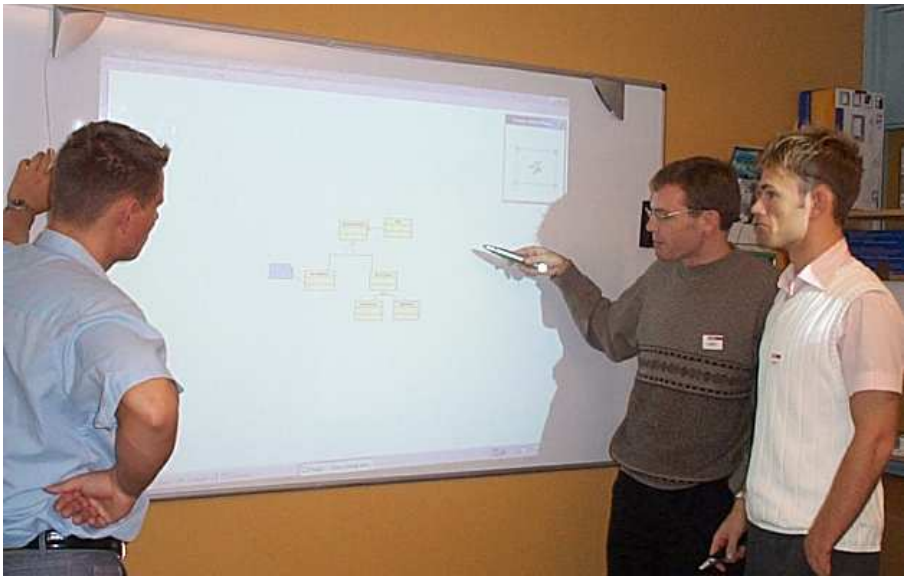


Figure 5.17: Student Use of Knight During Evaluations

experimenters were present during the study: one responsible for the study execution and observation and one responsible for providing help to the participants. We focuses on breakdowns and focus shifts (Bødker, 1996) in the use of the tool as well as in the use of the UML for modelling.

After the study, the subjects were interviewed.

5.3.2.3 Results

Generally, the Knight tools addressed some of the problems that the UML has in teaching and learning situations and was effective as a modelling tool even for novice UML users. In particular, we had the following observations:

Extent and Complexity. Using a gesture-based tool helps limit the amount of functionality presented to novice users: an instructor can choose to introduce only the gestures that are relevant to the students' current level and ignore more advanced UML elements. The directness of the gestures in the Knight tool is an important aspect of supporting learnability, particularly for novice users. The subsequent interviews supported this: all groups stated that the tool was "easy to use" and "easy to get started on".

Usability. In all studies, the freehand mode of the tool was used extensively for, e.g., user interface sketching, runtime object layout, and small code excerpts. All groups found the freehand mode very important in UML modelling. The expressive UML modelling also enabled users to start modelling with only a basic understanding of the UML.

Figure 5.18 shows an example of a model created by one of the groups in the study. It was created in less than one and a half hour by users with no prior experience in UML modelling and only a very basic introduction to the Knight tool.

5.4 Summary

This chapter has presented the Knight tool for gesture-based, collaborative object-oriented modelling and discussed how it addresses the challenges of multiple practices, heterogeneity, and open-endedness to experimental modelling

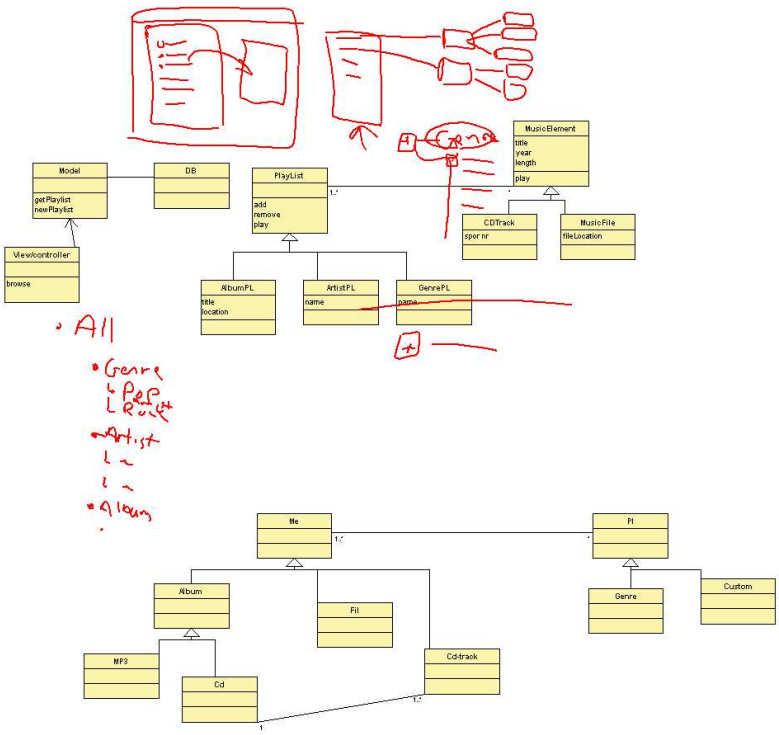


Figure 5.18: Model Created by Students during an Evaluation

Chapter 6

Conclusions and Future Work

This thesis has discussed modelling in the context of experimental system development. As evident, the research presented has had a large empirical and experimental component itself: we have modelled, observed modelling, used and created techniques for experimental modelling, and built and evaluated tools for experimental modelling.

We believe that as experimental techniques become more prominent in mainstream system development, tools and techniques for modelling will need to adapt as outlined in this thesis. In the rest of this chapter, we summarize our contributions towards this goal and point to areas in which the work presented in this thesis can be extended.

6.1 Grounding of Experimental Modelling

We have investigated object-oriented modelling in experimental system development through a number of project involvements and observations. Based on these investigations, we have extracted a number of challenges in the form of work instances. Particularly, we have identified the following, essential challenges to modelling in experimental system development. Experimental modelling is performed

- *by multiple practices,*
- *in an uncertain environment,*
- *iteratively, incrementally, and in parallel,*
- *with heterogeneous input and output, and*
- *in an open-ended way.*

This has implications for tool and techniques for experimental modelling, among others: Since experimental modelling needs to be performed by multiple practices, it cannot be supported by one, fixed, formal modelling notation or technique. Since

requirements are uncertain in experimental system development, models cannot be regarded as prescriptive specifications for systems. Since experimental modelling is iterative, incremental, and done in parallel, models need to be modular, extendible, and malleable. Since material for experimental modelling and output needs to be heterogeneous, tools and techniques should support the inclusion, and independence, of a mulititude of material. And since experimental modelling is open-ended, all tools and techniques should be changeable, to some degrees even within single modelling activities.

6.2 Techniques for Experimental Modelling

This thesis presented two sets of techniques in experimental modelling for

- *creating appropriate models and*
- *representing models appropriately.*

As a way of creating models that can be used as a basis for experimentation during system development, we outlined an architectural strategy with a focus on *architectural uncertainty*. Moreover, we pointed to how architectural patterns could be used to handle this uncertainty. For model representation, we proposed extensions to the existing UML modelling language based on the idea of *levels of restrictions* in conceptual and visual elements of object-oriented experimental models. By varying levels of restrictions, the same underlying modelling language can accommodate different users and uses as appropriate.

6.3 Tools for Experimental Modelling

We have discussed what tools for experimental modelling *should* support and presented how a tool for experimental modelling *could* be designed. This tool, *Knight*, supports collaborative, gesture-based modelling particularly tuned to electronic whiteboards. In this way, Knight tries to preserve, support, and enhance current, effective experimental modelling activities, such as brainstorming on whiteboards or collaborative discussions of models, as they evolve.

We have presented the design and implementation of Knight and in particular pointed to how Knight can be extended by other tools via a simple and effective integration architecture.

Finally, we have validated the effectiveness of Knight as a tool for software development and computer science education through a series of formal evaluations and a subsequent commercialization.

6.4 Future Work

The results presented in this thesis points to a large number of possible extensions that would be both useful and necessary. For each of the areas in which we have examined experimental modelling, there is future work to be done: with respect to grounding, further investigations, also of other types of modelling activities, are necessary. With respect to techniques, there are myriads of techniques in object-orientation and, e.g., participatory design that could be examined and contrasted. And in the area of software architecture, something like a catalogue of architectural restructurings and refactorings, a few of which were presented in Chapter 4, would be very useful in experimental modelling. With respect to tools, it would be possible to take the challenges more directly as input to a tool design and end up in quite different places than we did with the Knight tool.

The rest of this section extracts and discusses some main pointers to such future work.

6.4.1 Support for Diverse Experimental Modelling Processes

We have investigated experimental modelling in object-oriented system development. Two types of extensions of this work would be interesting: studies of other types of experimental modelling and more thorough studies of object-oriented experimental modelling.

Experimental modelling, i.e., the creation of representations of a future system through experimental techniques, is not limited to object-oriented modelling: requirements engineering, user interface design, and behavioural modelling, e.g., through Petri Nets are examples of effective modelling with experimental elements. Part of these explorations have already begun (Hansen, 2001). Moreover, the characteristics of, and challenges to, experimental modelling are applicable to collaborative "modelling" activities not connected to system development. In fact work on collaborative modelling in other settings (general brainstorming and process innovation) using the same techniques as for Knight has already been completed commercially by Ideogramic. A systematic investigation of and infrastructure for supporting such processes would be useful.

As extensions to our studies of experimental object-oriented modelling, it would be particularly interesting to follow modelling in an experimental system development project over time, i.e., conduct longitudinal studies of experimental object-oriented modelling. Knight will, given its overall stability, provide an excellent vehicle for longitudinal use and study of such use.

6.4.2 Pervasive Modelling

"Pervasive computing" considers the scaling of computational powers to support activities and processes distributed in space and time and tries to provide answers to questions raised by a spatially and temporally distributed computing infrastructure.

Such an infrastructure would be extremely useful for experimental modelling tools as well. We are currently working at distributing Knight in space but supporting temporally pervasive modelling, in particular based on results from the studies described in Section 6.4.1, would also be interesting in order to effectively create a true "pervasive modelling" technology. If true pervasiveness has to be achieved, most of the challenges of Chapter 3 and reconciling support for collaborative modelling by multiple competencies distributed in time or space or both will be particularly challenging.

In this context, it would also be interesting to extend the integration with Topos: model elements from Knight could be "first class" elements in Topos and the integration could be furthered to even include virtual as well as physical material.

Technically, new software architectures for modelling tools will have to be created to accommodate pervasive modelling. New advances in object-oriented publish/subscribe technologies (Eugster et al., 2001) and peer-to-peer computing will need to be considered. A peer-to-peer-distributed model representation technology with model elements shared by peers seems particularly attractive.

http://openp2p.com/

6.4.3 Pervasive User Interface Technology

To some extent, the development of the Knight tool has been a study in pervasive interaction techniques: part of the vision of Knight is for it to be available *when* and *where* needed. Currently, this boils down to supporting an interaction style based on gestures that works on input devices ranging from electronic whiteboard over desktop computers to portable Tablet PCs . We have also studied these kinds of interaction techniques in our work on the Kimura augment office environment (MacIntyre et al., 2001) that tries to support background awareness through a seamless integration of a focal desktop computer and peripheral interactive displays.

http://www.microsoft.com/windowsxp/tabletpc/

Equally interesting would be to support these transitions between interaction devices through various interactive visualizations of the common, conceptual core of the activities that a tool supports. In the Knight case, the interaction on an electronic whiteboard would resemble the current interaction, the interaction on an ordinary desktop PC would perhaps resemble a traditional direct manipulation interface, and the interaction on a handheld computer would be attuned to its limited physical size. Some technological groundwork for this has actually already been done in the context of the Knigt tool in (Tyrsted and Wibling, 2000) with respect to migration of applications. This would, obviously, depend on the pervasive infrastructure of Section 6.4.2 and a fully pervasive computational infrastructure would, ultimately, allow this to continue into the realm of augmented reality.

Bibliography

Aaen, I., Siltanen, A., Sørensen, C., and Tahvanainen, V.-P. (1992). A tale of two countries: CASE experiences and expectations. In Kendall, K., Lyytinen, K., and DeGross, J., editors, *The Impact of Computer Supported Technologies on Information Systems Development*, IFIP Transactions A (Computer Science and Technology), pages 61–93. Kluwer Academic Publishers.

Abowd, G., Atkeson, C., Feinstein, A., Hmelo, C., Kooper, R., Long, S., Sawhney, N., and Tani, M. (1996). Teaching and learning as multimedia: The classroom 2000 project. In *Proceedings of ACM Multimedia'96*, pages 187–198. ACM Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language*. Oxford University Press.

Andersen, C., Hansen, K., Sandvad, E., Thomsen, M., and Tyrsted, M. (2000). Tool support for iterative system development activities: Issues and experiences. In *Proceedings of NWPER'2000*, pages 1–21.

Astrachan, O. (2001). OO overkill: When simple is better than not. In *Proceedings of the ACM SIGCSE Symposium 2001*, pages 302–306.

Bally, L., Brittan, J., and Wagner, K. (1977). A prototype approach to information system design and development. *Information & Management*, 1(1):21–26.

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley.

Bass, L., Faneuf, R., Little, R., Mayer, M., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S., and Szczur, M. (1992). A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1):32–37.

Bødker, S. (1996). Applying activity theory to video analysis: How to make sense of video data in HCI. In Nardi, B., editor, *Context and Consciousness*, pages 147–174. MIT Press.

Beaudouin-Lafon, M. and Lassen, H. (2000). The architecture and implementation of CPN2000, a post-WIMP graphical application. In *Proceedings of UIST*

2000, ACM Symposium on User Interface Software Technology, pages 181–190.

Beaudouin-Lafon, M., Mackay, W., Andersen, P., Janecek, P., Jensen, M., Lassen, H., Lund, K., Mortensen, K., Munck, S., Ratzer, A., Ravn, K., Christensen, S., and Jensen, K. (2000). CPN/Tools: A post-WIMP interface for editing and simulating Coloured Petri Nets. In *Proceedings of Petri Nets 2000*, pages 19–28.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Blomberg, J., Suchman, L., and Trigg, R. (1994). Reflections on a work-oriented design project. In *Proceedings of PDC'94*, pages 99–110.

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72.

Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings.

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

Büscher, M., Christensen, M., Grønbæk, K., Krogh, P., Mogensen, P., Shapiro, D., and Ørbæk, P. (2000). Collaborative augmented reality environments: Integrating VR, working materials, and distributed work spaces. In *Proceedings CVE'2000*, pages 47–56.

Büscher, M., Kompast, M., Lainer, R., and Wagner, I. (1999a). The architect's Wunderkammer: Aesthetic pleasure & engagement in electronic spaces. *Digital Creativity*, 10(1):1–17.

Büscher, M., Mogensen, P., Shapiro, D., and Wagner, I. (1999b). The Manufaktur. Supporting work practice in (landscape) architecture. In *Proceedings of ECSCW'99*, pages 21–40.

Budde, R., Kautz, K., Kuhlenkamp, K., and Züllighoven, H. (1992). *Prototyping – An Approach to Evolutionary System Development*. Springer Verlag.

Bunse, C. and Atkinson, C. (1999). The normal object form: Bridging the gap from models to code. In *Proceedings of UML'99*, pages 691–705.

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

Buxton, W. (1986). There's more to interaction than meets the eye: Some issues in manual input. In Norman, D. and Draper, S., editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 319–337. Lawrence Erlbaum Associates.

Calvary, G., Coutaz, J., and Nigay, L. (1997). From single user architectural design to PAC*: a generic software architecture model for CSCW. In *Proceedings of CHI 1997, ACM Conference on Human Factors in Computing Systems*, pages 242–249.

Christensen, M., Crabtree, A., Damm, C., Hansen, K., Madsen, O., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., and Thomsen, M. (1998a). The M.A.D. experience: Multiperspective Application Development in evolutionary prototyping. In *Proceedings of the ECOOP'1998*, pages 13–40.

Christensen, M., Damm, C., Hansen, K., Sandvad, E., and Thomsen, M. (1998b). Architectures of prototypes and architectural prototyping. In *Proceedings of NWPER'98*, pages 247–267.

Christensen, M., Damm, C., Hansen, K., Sandvad, E., and Thomsen, M. (1999). Creation and evolution of software architecture in practice. In *Proceedings of TOOLS Pacific'99*, pages 2–15.

Christensen, M. and Sandvad, E. (1996). Integrated tool support for design and implementation. In *Proceedings of NWPER'96*, pages 168–184.

Coad, P. and Yordon, E. (1990). *Object-Oriented Analysis*. Prentice Hall/Yordon Press.

Coad, P. and Yordon, E. (1991). *Object-Oriented Design*. Prentice Hall/Yordon Press.

Cockburn, A. (1997). Structuring use cases with goals. *Journal of Object-Oriented Programming*. September/October.

Cockburn, A. (2001). *Agile Software Development: Software Through People*. Addison-Wesley.

Coutaz, J. (1987). PAC, an object-oriented model for dialog design. In *Proceedings of IFIP INTERACT'87*, pages 431–436.

Crabtree, A., Twidale, M., O'Brien, J., and Nichols, D. (1997). Talking in the library: Implications for the design of digital libraries. In *Proceedings of DL'97, ACM Conference on Digital Libraries*, pages 221–228.

Crispen, R. and Stuckey, L. (1994). Structural model: Architecture for software designers. In *Proceedings of TRI-Ada'94*, pages 272–281.

Damm, C., Hansen, K., and Thomsen, M. (1997). Issues from the GCSS prototyping project - experiences and thoughts on practice. Technical report, Computer Science Department, University of Aarhus.

Damm, C., Hansen, K., and Thomsen, M. (2000a). Tool support for object-oriented cooperative design: Gesture-based modeling on an electronic whiteboard. In *Proceedings of CHI 2000, ACM Conference on Human Factors in Computing Systems*, pages 518–525.

Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000b). Creative object-oriented modelling: Support for creativity, flexibility, and collaboration in CASE tools. In *Proceedings of ECOOP'2000*, pages 27–43.

Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000c). Supporting several levels of restriction in the UML. In *Proceedings of UML'2000*, pages 396–409.

Damm, C., Hansen, K., Thomsen, M., and Tyrsted, M. (2000d). Tool integration: Experiences and issues in using XMI and component technology. In *Proceedings of TOOLS Europe'2000*, pages 94–107.

Egyed, A. and Medvidovic, N. (1999). Extending architectural representation in the UML with view representation. In *Proceedings of UML'99*, pages 2–16.

Ellis, C., Gibbs, S., and Rein, G. (1991). Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58.

Eugster, P., Guerraoui, R., and Damm, C. (2001). On objects and events. In *Proceedings of ACM OOPSLA'2001*, pages 254–269.

Floyd, C., Reisin, F.-M., and Schmidt, G. (1989). STEPS to software development with users. In *Proceedings of ESEC'89*, pages 48–64.

Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Fowler, M., Beck, K., Brant, J., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*. August.

Fowler, M. and Scott, K. (2000). *UML Distilled*. Addison-Wesley, 2nd edition.

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

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In Ambriola, V. and Tortora, G., editors, *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company.

Goel, V. (1995). *Sketches of Thought*. MIT Press.

Greenbaum, J. and Kyng, M., editors (1991). *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates.

Grønbæk, K., Gundersen, K., Mogensen, P., and Ørbæk, P. (2001). Interactive room support for complex and distributed design projects. In *Proceedings of the Interact'01*, pages 407–414.

Grønbæk, K., Hem, A., Madsen, O., and Sloth, L. (1994). Designing Dexter-based and cooperative hypermedia systems. *Communications of the ACM*, 37(2):25–38.

Grønbæk, K., Kyng, M., and Mogensen, P. (1997). Toward a cooperative experimental system development approach. In Kyng, M. and Mathiassen, L., editors, *Computers and Design in Context*, pages 201–235. MIT Press.

Grønbæk, K., Sloth, L., and Ørbæk, P. (1999). Webvise: Browser and proxy support for open hypermedia structuring mechanisms on the WWW. *Computer Networks*, 31:1331–1345.

Gross, M. and Do, E. (1996). Ambiguous intentions: a paper-like interface for creative design. In *Proceedings of UIST'96, ACM Symposium on User Interface Software Technology*, pages 183–192.

Grudin, J. (1989). Why groupware applications fail: Problems in design and evaluation. *Office: Technology and People*, 4(3):245–264.

Grudin, J. (1991). Interactive systems: Bridging the gaps between developers and users. *IEEE Computer*, April:59–69.

Hansen, K. (1998). Exploiting architecture in experimental system development. In *ECOOP'98 Workshop Reader*, pages 110–114.

Hansen, K. (2001). Towards a Coloured Petri Net profile for the Unified Modeling Language – Issues, Definition, and Implementation. Technical Report Internal COT/2-52, Centre for Object Technology.

Hansen, K. and Ratzer, A. (2002). Tool support for collaborative teaching and learning of object-oriented modelling. To appear in Proceedings of ACM ITiCSE'2002.

Hansen, K., Yndigegn, C., and Grønbæk, K. (1999). Dynamic use of digital library material - supporting users with typed links in open hypermedia. In *Proceedings of the European Conference on Digital Libraries (ECDL'99)*, pages 254–273.

Hansen, K. M. and Thomsen, M. (1999). The "Domain Model Concealer" and "Application Moderator" patterns: Addressing architectural uncertainty in interactive systems. In *Proceedings of TOOLS Asia'99*, pages 177–190.

Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems. *Artificial Intelligence*, 72(12):329–365.

Hearst, M., Gross, M., Landay, J., and Stahovich, T. (1989). Sketching intelligent systems. *IEEE Intelligent Systems*, 13(3):10–19.

Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Addison-Wesley.

Highsmith, J. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House.

Hilliard, R. (1999). Using the UML for architectural description. In *Proceedings of UML'99*, pages 32–48.

Hughes, J., King, V., Rodden, T., and Andersen, H. (1994). Moving out of the control room: Ethnography in system design. In *Proceedings of CSCW'94, ACM Conference on Computer-Supported Cooperative Work*, pages 429–439.

Hughes, J., Randall, D., and Shapiro, D. (1992). Faltering from ethnography to design. In *Proceedings of CSCW'92, ACM Conference on Computer-Supported Cooperative Work*, pages 115–122.

Iivari, J. (1996). Why are CASE tools not used? *Communications of the ACM*, 39(10):94–103.

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

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press.

Jarzabek, S. and Huang, R. (1998). The case for user-centered CASE tools. *Communications of the ACM*, 41(8):93–99.

Kemerer, C. (1992). How the learning curve affects CASE tool adoption. *IEEE Software*, 9(3):23–28.

Kensing, F. and Madsen, K. (1991). Toward a cooperative experimental system development approach. In (Greenbaum and Kyng, 1991), pages 155–168.

Knudsen, J., Löfgren, M., Madsen, O., and Magnusson, B., editors (1994). *Object-Oriented Environments: The Mjølner Approach*. Prentice Hall.

Kobryn, C. (1999). UML 2001: A standardization odyssey. *Communications of the ACM*, 42(10):29–37.

Krasner, G. and Pope, S. (1988). A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49.

Kristensen, B. and Østerbye, K. (1994). Conceptual modeling and programming languages. *ACM SIGPLAN Notices*, 29(9):81–90.

Kruchten, P. (1999). *The Rational Unified Process, An Introduction*. Addison-Wesley, 2nd edition.

Kurtenbach, G. (1994). *The Design and Evaluation of Marking Menus*. PhD thesis, University of Toronto, Canada.

Landay, J. and Myers, B. (1995). Interactive sketching for the early stages of user interface design. In *Proceedings of CHI 1995, ACM Conference on Human Factors in Computing Systems*, pages 691–705.

Lending, D. and Chervany, N. (1998). The use of CASE tools. In *Proceedings of the 1998 ACM SIGCPR Conference*, pages 49–58.

MacIntyre, B., Mynatt, E., Voida, S., Hansen, K., Tullio, J., and Corso, G. (2001). Support for multitasking and background awareness using interactive peripheral displays. In *Proceedings of UIST 2001, ACM Symposium on User Interface Software Technology*, pages 41–50.

Mackay, W. and Pagani, D. (1994). Video Mosaic: Laying out time in a physical space. In *Proceedings of ACM Multimedia'94*, pages 165–172.

Mackay, W., Velay, G., Carter, K., Ma, C., and Pagani, D. (1993). Augmenting reality: Adding computational dimensions to paper. *Communications of the ACM*, 36(7):96–97.

Madsen, O. (1995). Open issues in object-oriented programming – a Scandinavian perspective. *Software – Practices and Experience*, 25(S4):3–43.

Madsen, O. and Møller-Pedersen, B. (1989). Virtual classes: a powerful mechanism in object-oriented programming. *ACM SIGPLAN Notices*, 24(10):397–406.

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

Mankoff, J. and Abowd, G. (1998). Cirrin: A word-level unistroke keyboard for pen input. In *Proceedings of UIST 1998, ACM Symposium on User Interface Software Technology*, pages 213–214.

McLennan, M. (1993). [incr tcl]: Object-oriented programming. In *Proceedings of the Tcl/Tk Workshop*, pages 31–38.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, 2nd edition.

Mogensen, P. (1994). *Challenging Practice: An Approach to Cooperative Analysis*. PhD thesis, University of Aarhus, Aarhus, Denmark.

Moran, T., Chiu, P., Harrison, S., Kurtenbach, G., Minneman, S., and van Melle, W. (1996). Evolutionary engagement in an ongoing collaborative work process: A case study. In *Proceedings of CSCW'96, ACM Conference on Computer-Supported Cooperative Work*, pages 150–159. ACM Press.

Mynatt, E. (1999). The writing on the wall. In *Proceedings of INTERACT'99*, pages 196–204.

OMG (2000a). Unified Modeling Language 2.0 request for proposals. Technical Report ad/00-09-05, Object Management Group.

OMG (2000b). Unified Modeling Language specification 1.3. Technical Report formal/00-03-01, Object Management Group.

OMG (2000c). XML Metadata Interchange 1.0. Technical Report formal/2000-06-01, Object Management Group.

OMG (2001a). Meta Object Facility 1.3.1. Technical Report formal/01-11-02, Object Management Group.

OMG (2001b). Unified Modeling Language specification 1.4. Technical Report formal/01-09-67, Object Management Group.

OMG (2002). XML Metadat Interchange 1.0. Technical Report formal/2002-01-01, Object Management Group.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Ousterhout, J. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.

Perlin, K. (1998). Quikwriting: Continuous stylus-based text entry. In *Proceedings of UIST 1998, ACM Symposium on User Interface Software Technology*, pages 215–216.

Pfaff, G. (1985). *User Interface Systems*. Eurographics Seminars. Springer-Verlag.

Raner, M. (2001). Teaching object-orientation with the Object Visualization and Annotation Language (OVAL). In *Proceedings of the ACM ITiCSE'2001*, pages 45–48.

Rogerson, D. (1997). *Inside COM. Microsoft's Component Object Model*. Microsoft Press.

Rouncefield, M., Hughes, J., Rodden, T., and Viller, S. (1994). Working with "constant interruption": CSCW and the small office. In *Proceedings of CSCW'94, ACM Conference on Computer-Supported Cooperative Work*, pages 275–286.

Royse, W. (1970). Managing the development of large software systems: Concepts and techniques. In *Proceedings of IEEE WESCON*, pages 328–338.

Rubine, D. (1991). Specifying gestures by example. *Computer Graphics*, 25(3):329–337.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loresen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

Scott, L., Horvath, L., and Day, D. (2000). Characterizing CASE constraints. *Communications of the ACM*, 44(11es):232–238.

Sharma, S. and Rai, A. (2000). CASE deployment in IS organizations. *Communications of the ACM*, 43(1):80–88.

Shaw, M. (1995). Patterns for software architectures. In Coplien, J. and Schmidt, D., editors, *Pattern Languages of Program Design*, pages 453–467. Addison-Wesley.

Shaw, M. (1996). Some patterns for software architectures. In Vlissides, J., Coplien, J., and Kerth, N., editors, *Pattern Languages of Program Design 2*, pages 255–271. Addison-Wesley.

Shlaer, S. and Mellor, S. (1989). *Object-Oriented System Analysis: Modeling the World in Data*. Yordon Press.

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method : The Method in Practice*. Addison-Wesley.

Tsichiritzis, D. and Klug, A. (1978). The Ansi/X3/SPARC DBMS framework: Report of the study group on database management systems. *Information Systems*, 3(3):173–191.

Tyrsted, M. and Wibling, G. (2000). Application migration in a pervasive computing environment. Master's thesis, University of Aarhus, Denmark.

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

Part II

Papers

In Proceedings of ECOOP'98

The M.A.D. Experience:

Multiperspective Application Development in evolutionary prototyping

Michael Christensen, Andy Crabtree, Christian Heide Damm, Klaus Marius Hansen, Ole Lehrmann Madsen, Pernille Marqvardsen, Preben Mogensen, Elmer Sandvad, Lennert Sloth, Michael Thomsen

Department of Computer Science, University of Aarhus, Building 540,
Ny Munkegade, DK-8000 Aarhus C, Denmark.

{toby, andyc, damm, marius, olm, pernille, preben, ess, les, miksen}@daimi.aau.dk

Abstract. This paper describes experience obtained through a joint project between a university research group and a shipping company in developing a prototype for a global customer service system. The research group had no previous knowledge of the complex business of shipping, but succeeded in developing a prototype that more than fulfilled the expectations of the shipping company. A major reason for the success of the project is due to an experimental and multiperspective approach to designing for practice. Some of the lessons to be learned for object-orientation are (1) analysis is more than finding nouns and verbs, (2) design is more than filling in details in the object-oriented analysis model, and (3) implementation is more than translating design models into code. Implications for system development in general and object-orientation in particular consist in the preliminary respecification of the classical working order: analysis – design – implementation.

Keywords: Large-scale system development, multiperspective application development, cooperative design, ethnography, object-orientation, rapid prototyping, evolutionary prototyping, OO-tools, experience report.

1 Introduction

In late January 1997 a globally distributed shipping company contacted CIT, proposing a joint project for the purpose of developing a prototype of a Global Customer Service System (GCSS). The initial concept of GCSS emerged from within the company as a means of supporting the global implementation of business process improvements (BPI). A month later a team was assembled that had never worked

together before and had no knowledge of shipping whatsoever. A project description was made and ten weeks and 5 major iterations later, the prototype was appraised and approved by the company's highest executive body. Work is still on-going. This experience report tries to describe central success criteria and lessons learnt to date.

Partners in the project (known as the Dragon Project) were the company (which cannot be named for commercial reasons) and the DEVISE research group.

The project was funded by The Danish National Centre for IT-research (CIT, `http://www.cit.dk`). The company is a large world-wide provider of containerised transport solutions, employing some two and a half thousand people in over 250 offices in some 70 countries on six continents. Having a superior customer service in an increasingly competitive market as its key value, the company has and is investing substantially in IT support for proposed business solutions.

The DEVISE group was originally founded with the research goal of increasing productivity and quality in the development of large, complex systems, and now incorporates a large number of diverse competencies. Participants in the Dragon Project include, from the university: one project coordinator, one participatory designer, three full-time and three half-time object-oriented developers, one ethnographer and for a part of the project, one usability expert. Participants from the business include senior management, business representatives from the major continental regions, administrative staff, customer service personnel and members from a private consultancy company.

Although the essence of shipping in our context might be described as the actual transport of cargo in containers, our concern is with the provision and delivery of customer services, which the transport of containers relies upon. Work here includes the handling of interactions with customers in formulating prices for transport (quoting), in booking containers, in arranging inland haulage, in documenting ownership of cargo, in notifying the consignee of cargo's arrival, and so on.

From this all too brief description, the job of design might sound like a relatively straightforward task but for many reasons the challenge that GCSS has to meet is anything but simple. Currently, no global system or formal practice exists, yet GCSS should support coordination between globally distributed sites as well as support a customer service that, while streamlined (i.e. operating with less resources), is both effective and efficient and at the same time respects local needs in the various regions. Thus, the main design issue in many respects was to develop a prototype supporting these needs. Important issues in addressing these needs consisted in:

- Developing an approach for obtaining detailed knowledge of shipping in a short period of time; this knowledge had to be developed during the project due to the very short project period.
- Similarly, the initial requirements and formal specifications for the prototype from the business were vague and had to be developed during the project.
- The architecture of the prototype / system had to be flexible and allow for local customisations.

In order to satisfy these issues we thought it necessary to form a development group consisting of people with diverse but nevertheless complementary qualifications, who would be able to utilise their own specialities and at the same time be able to work together in an extremely focused group. The focus being to develop a prototype to support customer service. The roles of the different team members may, somewhat simplistically, be described as follows:

- The ethnographer: focus on current practice within customer service and related areas.
- The participatory designer: focus on the design of future practice and technological support with users.
- The OO developers: develop the object model, and implementation.

In what follows we describe the contributions to the project from the perspectives of ethnography, cooperative design and object-orientation. Having explicated what we take to be the approach's main strength, namely its experimental and multiperspective character in designing for practice, the paper concludes by stating lessons that we take to be of value with particular regard to object-orientation. In so much as object-orientation played a major role as a technology for developing a model of shipping and for implementing the prototype, then a number of lessons for object-orientation were learned. Perhaps the most important ones are:

1. *Analysis is more than finding nouns and verbs.* We will argue that it is often necessary and useful to base analysis on much more powerful means than currently advocated, such as ethnographic analyses of the social organisation of work.
2. *Design is more than filling in details in the OO analysis model*. Here we will argue that often techniques from participatory or cooperative design are necessary and useful in formulating concrete design-solutions and relating them to practice.
3. *Implementation is more than translating design models into code*. Our experience shows that it is important to start implementation early in the design. The initial design model will always be changed during implementation and the feedback from initial implementation should appear early in the project so as to facilitate further development.

The above lessons are of course not relevant for all kinds of projects but for a project as the one described in this paper we think they are relevant. Perhaps most important are the implications our work has for structured techniques of object-oriented analysis, design and implementation. Specifically, towards the inadequacies of formal development methods in accomplishing rapid prototyping.

2 M.A.D – system development using a common frame of reference

The Dragon development team came from backgrounds ranging from ethnography to cooperative design to object-oriented development. These competencies should not be treated in strict separation but rather, in the ways in which they interact and complement one another and in this way achieve a synergetic effect.

A guiding principle in this multiperspective approach was (indeed is) an orientation to a common frame of reference: everyday business practice in customer service. In this paper, we use an example – a real world 'instance' of working practice, namely "rerouting" - that was employed by all the perspectives at work to show how design was practically achieved. In so much as the instance was oriented to by each perspective, we use it to illuminate the *character* of each perspective. Furthermore, in so much as the instance was employed by each perspective, then we use it to illuminate the *ways* in which the perspectives *interrelated* in practice and, reciprocally, while preserving individual achievements, to display the *organisation of work* from which GCSS emerged as a design product.

The instance of rerouting was only one of many used during development. Furthermore, the competencies or perspectives did not treat the instances sequentially but rather, treated them in parallel or at different times thereby mutually informing one another and the prototype as work progressed.

In order to display the organisation of multiperspective application development in rapid, evolutionary prototyping, we now turn to a description of how the three main perspectives, that is the ethnographic perspective, the cooperative design perspective and the object-oriented perspective, approached development and coordinated their activities into a coherent 'process' from which a concrete product (more than) satisfying the requirements of a large geographically distributed organisation emerged. After this account we will outline some of the major lessons learnt.

3 The ethnographic perspective

Whether operating within the context of rapid prototyping or not, the ethnographic perspective places an emphasis on securing a real world reference to work and organisation, informing the design of cooperative systems (Heath et al., 92; COMIC 2.2, 93; Blomberg et al., 94). Working on the presupposition that systems design is work design (in contrast to model building for example[1]), it might otherwise be said that ethnography seeks to base design on an 'understanding of praxis' in which the system is to be *put to work*. The intention is to predicate design on enacted practice thus supporting the development of systems that resonate or 'fit' with the activities in which they are to be embedded, thereby circumventing a major cause of systems

[1] This is not to abnegate modelling but rather, to remind designers of the relevance of modelling – it is not an end in itself but a means to an end, namely the (re)organisation of human activities.

failure (Grudin, 89; Shmidt et al., 93; Hughes et al., 94). The notion of understanding praxis is 'bracketed' in that it is selective – ethnography orients to praxis in a certain or particular way.

From ethnography's point of view, work is seen to be socially organised with organisation itself emerging as a local production out of the *routine accomplishment and coordination of tasks* experienced by practitioners (and generally understood) as the working division of labour (Anderson et al., 89). Ethnography's task is to make the working division of labour visible and available to designers in the concrete details of its accomplishment – i.e. in terms of work's real time organisation in contrast to idealised form (Rouncefield et al., 94). This is achieved through direct observation and naturalistic description of working practice portrayed from the point of view of parties to the work (Crabtree et al., 97). The notion of 'social organisation' refers to the conventional ways in which activities are accomplished and coordinated by *any and every* competent member engaged in the work. The notion of 'convention' refers, conjointly, to members' reoccurring actions and interactions, the artefacts used and the practical reasoning employed in doing the work. Work's social organisation is discovered through 'mapping the grammar' of the domain by empirical instance[2].

Simply and briefly put, the notion of 'mapping grammar' refers to *ordinary language* which is seen to embody 'language-games' (Wittgenstein, 68). Language-games are distinct practices - computer science, sociology, customer service in shipping for example are *unique* ways of talking and thus of *acting* in the world – each of which consists in a unique family of concepts ('quoting', 'export handling', 'documentation', 'rerouting' in customer service for example). In so much as language-game concepts *are* activities and in so much as they are *intersubjectively* employed ~ enacted by a unique *collectivity* of people (customer service staff for example), then to describe the actual *performative* details of their use is to describe the *social* organisation of the named activity. The activity of description is called 'mapping', the concepts mapped the 'grammar' of the language-game, the description itself the 'instance'. Empirical examples or 'real world' instances of language-game concepts-in-use not only preserve the social organisation of work in describing that organisation *in its own terms* but in so doing *make visible* what the work is 'really all about'[3] (Hughes et al., 92).

3.1 The Bremerhaven Instance (1)

Mapping grammar, a practical example: Customer service work in the container shipping business consists in the activities of 'quoting', 'export handling', 'alloca-

---

[2] The notion of 'mapping grammar' is explicated in *Talking Work: language-games, organisations and computer supported cooperative work* (Crabtree, forthcoming).

[3] It should be said that in so far as ethnography does preserve the real world character of work then it does so through an attention to concepts-in-use, to 'rerouting' as a contingent but nevertheless routine human achievement in, of and as work in contrast to 'rerouting' as an abstract information process (Garfinkel et al., 70; Hughes et.al, 96). This is not to rule abstraction out of play but to ground abstraction in formal properties of enacted (in contrast to idealised) praxis

tion', 'documentation' and 'inbound handling'. These concepts or categories delineate the working division of labour. One important feature of allocation work is to 'reroute' freight in response to contingencies. Rerouting occasions collaboration between equipment management, export and import handling, and documentation[4] and is itself occasioned for various reasons: bad weather, customer requirements, running off schedule etc. Rerouting consists in 'reallocating' containers to a substitute vessel or vessels - there may be several 'legs' and different vessels on any particular container's journey. The substitute vessel may be located in a different port to either the load or leg port. More: the destination port of rerouted cargo from a particular vessel may well be different. Thus, the activity of rerouting is all about arranging appropriate transport for cargo going to multiple destinations from some contingent point either to the destinations direct or, failing that, to a point from which cargo can be delivered to its respective destination ports. Real world instances of rerouting's accomplishment – actual *responses* to a vessel being delayed – revealed that route alterations occasioned not only changing the first leg of a journey but also the first half of the second leg for instance, or, indeed several legs on a journey. Furthermore, these changes were discovered to be subject to criteria of rerouting, specifically of time and cost: independent local carrier was specified wherever possible if time allowed, rail failing local carrier, truck failing rail; local carrier is more cost effective than rail, rail more cost effective than truck although time pressures may necessitate everything being moved by truck. It also transpired that rerouted cargo must be grouped: in some locations refrigerated containers, used for perishable products in particular, cannot be moved to transhipment points by rail for example (due to concerns of supervision – the temperature of refrigerated containers must be monitored at regular intervals), freight for this or that destination must be identified and the criteria applied. Furthermore, as a result of 'hard-wiring' working processes into the existing system, rerouting had to be accomplished individually – groups of freight could not be selected and assigned to alternate vessels except in the simplest of cases: 'roll over', i.e. temporal rescheduling to the next available vessel.

3.2 Analysing the instance

Mapping the socially organised details of activities accomplishment by empirical instance not only allows us to understand praxis in the context of the working division of labour[5] but in so doing, to identify *practical problems of work* and their *situated methods of solution*. Taken together, the practical problems of work and members' intersubjective or shared methods of solving those problems, and of solving them routinely, day-in-day-out in the face of any and all contingencies, show us just 'what

---

[4] Space prevents a detailed description of the work involved here but in practice (i.e. design) these details were central to the formulation of design-solutions.

[5] Which enables us to get 'hands on' the actual ways in which activities of work are *coordinated* and thus identify what is essential to successful work-oriented design – if activities cannot be coordinated, work simply cannot be achieved and we can add another systems failure to the list.

the work is really all about' and make *concrete possibilities* for support through design *visible and available to* design. Thus, the 'instance' circumscribes a **problem-space** for design. More: in illuminating the socially organised ways in which staff routinely go about solving practical problems of work, the instance circumscribes a **solution-space** for design.

The outcome of analysing the rerouting instance for example, a collaborative task performed by members from each of the perspectives at work, was the development of a flexible 'tree structure' supporting local decision-making and the coordination of a complex task by actively displaying all destinations for cargo on a delayed vessel and through the application of which one can specify any number of leg changes and different modes of transport for any group of selected containers, updating all members of the specified group in one go.

3.3 The limits of ethnography

Clearly there is a great deal more to observing praxis than meets the unaccustomed eye. In securing a real world reference preserving the context of work the ethnographic perspective provides concrete topics and resources for design. In so much as system design is work design, ethnography thus supports the development of systems that resonate or 'fit' with the activities of work in which they are embedded thereby supporting the local production of organisation in innovative social and technical ways (Kensing et al., 97). Virtues aside, ethnography is not without its limitations: it is one thing to identify a **problem-solution space**, another to identify design-solutions – a very practical problem resolved in practice through prototyping.

4 The cooperative design perspective

Cooperative design (also known as participatory design), as developed in Scandinavia over the last decades, stresses the importance of creative involvement of potential users (end-users, managers, affected parties, etc.) in design processes (Bjerknes, Ehn, & Kyng, 87; Greenbaum & Kyng, 91). It does so both from a "moral" perspective – users are the ones who have to live with the consequences of design – and from a practical perspective – they are competent practitioners understanding the practical problems of work, a capacity which enables them to assess and / or come up with alternatives to design. From this perspective, design is seen to be a cooperative activity involving not only end users but also other groups with very different but indispensable competencies. A primary means to the end of designing successful products is, accordingly, to bring these competencies together. This is often achieved in workshop like settings through the appliance of the different perspectives and competencies on a common and concrete issue, e.g. descriptions of current work, scenarios for future possibilities, mock-ups, and prototypes. The underlying assumptions and claims as well as a number of tools and techniques for the practical employment of cooperative design may be found in (Greenbaum & Kyng, 91;

Grønbæk, 91; Grønbæk, Kyng, & Mogensen, 93; Grønbæk, Kyng, & Mogensen, In Press; Mogensen, 94).

Closely related to the cooperative design activities, a number of usability studies were conducted. In this context 'usability' is attempting to transcend a history of work "with one user in a lab thinking aloud" and move evaluation into the workplace itself. In this respect, it is a deliberate attempt to move away from a conception of users as human factors to a conception of users as human actors, and from the notion of finding 'problems' to a notion of the user as an active participant in design (Bannon, 91; Bødker, 91; Wiklund, 94). Despite struggling with an in-built cognitive bias (Grudin, 89; Bannon et al., 93; Twidale et al., 94) particular workplace studies supplemented with more traditional one-to-one lab type evaluations focusing on human computer interaction (HCI), provided detailed information relating to specific usage of the prototype and thus supplemented cooperative design activities.

Cooperative design has, by and large and up until now, been carried out in situations where the potential users constituted groups of manageable proportions. One of the major challenges for cooperative design in the Dragon Project is to resolve problems of scale: the organisation is distributed across more than 250 offices in 70 countries around the world and users may be found on many different levels in the organisation. How does one work with such dispersed groups? How does one find representative users? How does one deal with all the different (and frequently conflicting) perspectives and interests among not only different levels in the managerial hierarchy, but also culturally and regionally dispersed groups?

The strategy employed so far can be characterised as a mix of two approaches. On the one hand we have worked in various specific areas, regarding geographical locations, specific work domains, and corresponding functionality (e.g. rerouting, initially with the Aarhus site as primary point of reference). Embedding design in actual practice has helped to ensure in-depth knowledge regarding the issues being designed for at the given time. On the other hand, and at the same time, we have tried to maintain and elaborate "the big picture", both regarding the functionality GCSS eventually will provide as well as the regional aspects in order to prepare for, or at least not counteract, later developments. Both strategies have been approached in parallel and, naturally, both approaches have influenced one another.

To date, most of the concrete design activities involving developers and users have been carried out in four ways:

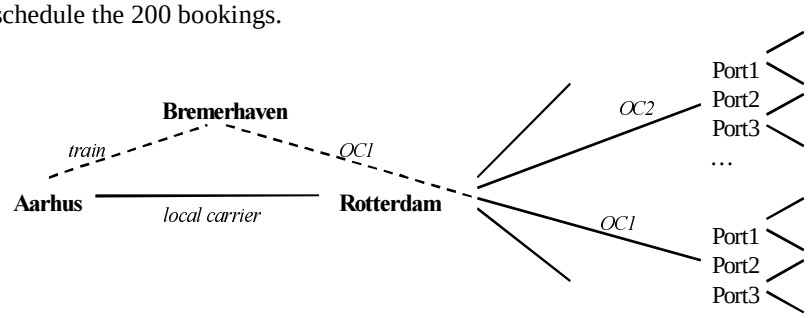
- Presentations of the prototype with subsequent comment / discussion sessions (all in all 100+ users from over 20 countries).
- Workshops (1-3 days) elaborating the details of current problems, current stage of the prototype, and various alternatives (all in all 25 users from around 10 different countries).
- Continuous workshops analysing and designing aspects in various versions of the prototype (6 users from 4 countries).
- A series of usability studies (8) with the business representatives attached to the project and with customer service staff in the local office in Aarhus (one to two users at a time).

An example of how the first two types of interaction have worked is a recent visit to Singapore: Four developers and the two business representatives working with us in Aarhus arrived in Singapore on Thursday. Friday morning we presented the prototype and its intended use to some 20 people from various positions in the Singapore office for around 3 hours. In the afternoon we all joined various people doing their usual work in the office. Saturday morning, we had a workshop with four people centred on export handling. Saturday afternoon, we discussed lessons learned so far and decided on changes to be implemented immediately, issues for redesign when we came home, issues that were out of scope, and features that would be 'nice to have'. Monday, we split up. The ethnographer focused on issues where we needed more information (allocation and pricing), observing work-in-progress and interviewing people in the office. The cooperative designer and three users discussed and elaborated details regarding booking in a 'hands on' session with the prototype. The two OO developers started to implement changes prompted by the presentation and observations of work which were agreed upon on Saturday. Tuesday morning, we presented the changes in the prototype for around 10 people and went into detail regarding the next issue on the agenda, allocation and documentation (including pricing). Tuesday afternoon, we went to Malaysia. Wednesday morning, we presented the prototype in the Malaysian office …

4.1 The Bremerhaven Instance (2)

An example of the continuous work 'back home' between developers and users is the Bremerhaven example. When rerouting came on to the agenda, company personnel gave us a brief introduction to the problem. The ethnographer went to the office in Aarhus and collected a series of examples or instances of actual reroutings dealt with at the office. The cooperative designer started to come up with ideas for supporting rerouting based on existing knowledge and experience. The next morning was spent in a continuous 'ping-pong' between various suggestions for supporting rerouting and the instances of actual reroutings coming out of the ethnography. After three to four iterations, an understanding of the problem as well as a first suggestion for the design emerged. Both were presented to and discussed with business representatives in the afternoon. Shown in Figure 1 is an account of the emergent understanding of the problem as well as the first design-solution (in real time, the example and the design was constantly produced and reproduced on paper sketches and white-boards).

You have 200 bookings out of Aarhus, going to Rotterdam via a local carrier. 100 of these bookings are transhipped on the intercontinental, *OC1*, headed for the Americas; another 100 are transhipped on to another intercontinental, *OC2*, headed for Asia.
Problem: the local carrier does not call Aarhus this week, we have to reroute or reschedule the 200 bookings.



Solution: *OC1* calls Bremerhaven the day before it calls Rotterdam, so we can send all the containers to Bremerhaven by train, get them on the *OC1* a day before planned - everything is "back to normal". *OC2* does not serve Bremerhaven, we have to reschedule both for another local carrier and *OC2*.

Figure 1. The Bremerhaven instance

In considering how this is achieved in practice, it became apparent that affected cargoes' *destination* were different, spreading out globally like the branches on a tree. The suggested design was to represent all affected bookings in a tree structure with alternating ports and carriers. When the user, for example, clicks on *OC1* (see the above sketch) in the tree structure, it means that the user is now operating on all bookings out of Aarhus, transhipped in Rotterdam, and going out on *OC1*. Having selected the intercontinental carrier *OC1*, all further transhipment ports and final destinations may be seen. From here it is now possible to do rerouting via Bremerhaven for any particular group of bookings

Naturally, the above instance does not capture all possible problems in rerouting, neither does the design represent the final solution. The point is that the above understanding presented, in a very compact and understandable form, a good starting point and served as a powerful tool in further development where both the design solutions and the 'Bremerhaven instance' were refined and elaborated. As a paradigm case in point, the Bremerhaven instance was:

- Used and reproduced within the development group as common point of reference for design. The example states problems and solutions from current practice in respect to a specific design problem (designing for the rerouting and rescheduling of multiple bookings). Whenever we

encountered problems in the implementation, the instance worked as a common resource: whatever the specific design ideas and problems were, the quality criteria was always whether we could support the instance, not, for example, whether we fulfilled some predefined requirements.

- Used and reproduced in a large number of presentations and workshops between people from the development team and users from various locations and levels. It is an integral part of the prototype: on the one hand it explains a problem the prototype is trying to resolve, on the other hand, discussing ways of solving the problem triggers new understanding of the problem and possible methods of solution. Roughly a week after the first formulation of the instance, it was confronted with staff from a large 'import' port. As was pointed out, the design works very well for rerouting in out-bound, it does not work when you are sitting in in-bound, because here the focus is on what is coming towards you. The instance was expanded with that example, and in the design we catered for the option of having either the receipt or the delivery port as the 'root' in the tree structure.
- Used and reproduced in the usability studies where, embodied in scenarios, it provided the starting point and context for assessing the prototype. In testing the prototype, the instance as well as the design was further elaborated. Up until testing for example, we provided for either an 'outbound' or an 'inbound' view of the tree-structure. What came out of the usability studies in this respect was the idea that we actually needed both at the same time, facilitating an overview of both what was 'coming in' and what was 'going out'.

5 The object-oriented perspective

The OO perspective applied in this project is based on the Scandinavian tradition (Madsen et al., 93) where the focus is on modelling as opposed to technical concepts like encapsulation and inheritance. The first OO language, Simula was originally designed as a means for writing simulation programs and when writing simulation programs, it is useful to have a language with good modelling capabilities. The modelling approach has been very explicit in the design of BETA (which was the language used in the implementation of the prototype). One of the main advantages of object-orientation is that it provides an integrating perspective on analysis, design and implementation, but in order to fully realise this potential, there should be a proper balance and integration between modelling and implementation capabilities of the languages (Madsen, 96).

The application being developed (in this case the prototype) is considered as a physical model of a perspective on the problem domain. Selected phenomena and concepts in the application domain are synthesised into the model and represented as classes, objects, properties of objects and relations between objects. This model is a

very explicit part of the application and it can be regarded as a view on the application expressed in problem domain-specific terms. The latest version of the model is always the one in the application.

Below we discuss the concept of modelling, how the model was produced, the evolution of the model, the architecture in which the model is embodied, and the crucial role of tools in responding to amendment and change.

5.1 The concept of modelling

Working within our OO-perspective, system development is based on an understanding of the concepts used within the settings that the system will eventually support. The setting we refer to as the *referent system* and the process of translating referent system specific concepts to concepts within the computer system we refer to as *modelling*. The result of the modelling process we refer to as the *model system* or just the model (Madsen et al., 93, pp.289, Knudsen et al., 94, pp.52). Using the model within a referent system context we denote *interpretation.* This can, for example, be discussion of business concepts with business users while referring to the model system. Below we discuss how modelling was actually achieved.

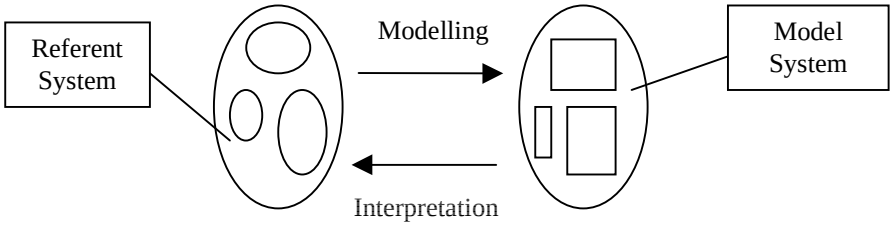


Figure 2. Customer service practice (referent system) in relation to the GCSS prototype (model system)

5.2 The production of the model

The activity of modelling was started right from the beginning of the project. Modelling has been an intentionally integrated part of each of the rapid prototyping cycles, in a sense on the same terms as the user interface and the architecture. The length of these cycles - the actual time spent on each – precludes a sharp distinction between the activities of analysis, design and implementation[6]. Working in an interdisciplinary context, our experiences extend existing notions of analysis, design and implementation. As such, a number of various resources and artefacts were used in producing the model. These included sessions with business representatives,

[6] A distinction which, as (Madsen et al., 93, pp 316) remind us, is highly questionable: 'In practice it is very difficult to do analysis without doing some design, and similarly doing design without doing some implementation.' Our experiences concur.

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*. The Yourdon diagrams, which came about as an output from the BPI analysis, exemplify the way in which the above resources and artefacts informed the development of the model, being used, much as use case diagrams may be used, as *'sensitising'* devices providing for the initial identification of objects. It should be stressed that the Yourdons were in no way used as requirements specifications (as such diagrams do not display actual practice in sufficient detail), rather it was a matter of using whatever resources were available to get an understanding of the problem domain.

Modelling proceeded in iterative phases: focussing on a particular area of business in collaboration with the business representatives attached to the project, the participatory designer and ethnographer; identifying important phenomena and concepts; analysing structure, and synthesising into a new iteration of the model. In the following example we exemplify and explain how modelling was achieved.

Modelling, a practical example - identifying concepts in the quote process*:* By a combination of review constraints, formal business process descriptions stemming from the BPI and the need to start the modelling process with an accessible instance of business, the quote process was treated first. A particular discussion about the quote process with a business representative allowed us to identify important concepts and phenomena embodied *within* the quote process, such as 'corridor', 'routing', 'container', 'rate' and 'customer'. This session, as other sessions including other participants as mentioned above, was recorded in the form of a *blackboard snapshot* showing the concepts, some properties of the concepts, examples of values of properties along with textual and pictorial annotations clarifying or explaining referent system specific concepts. Figure 3 displays the features of the snapshot from the discussion with the business representative regarding the quote process.

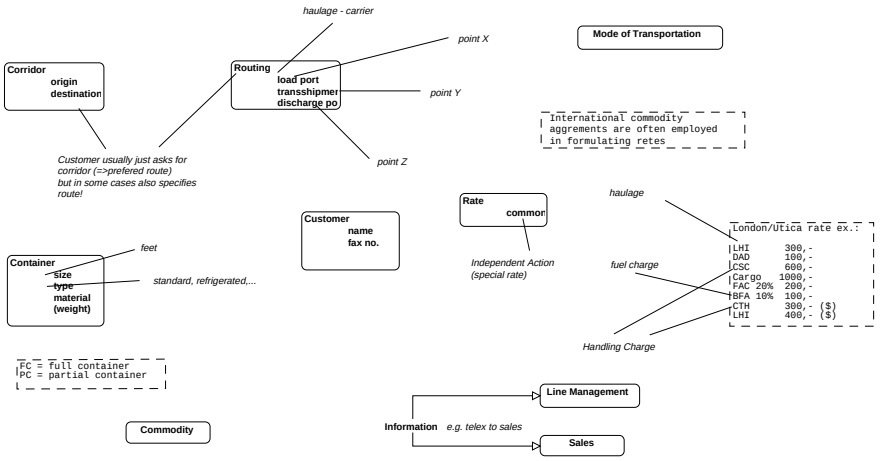


Figure 3. A blackboard snapshot

Blackboard snapshots produced through preliminary analysis of descriptions of working practice and the artefacts used were of considerable value and practical utility in creating the model. They provided rich detail and a firm basis upon which we were able to build a first model of the quote process. Furthermore, they worked as an important means of communication between the respective competencies at work.

Having created an initial model (see Figure 4), the model was then *interpreted* in the context of the referent system. The reason for this was to discuss the model with relevant members of the development team (including business representatives) in order to assess the state of the model and gain additional information as to the substance as well as the structure of the quote process. Other parts of the referent system were then analysed in the same way and synthesised with the parts of the model already elaborated[7].

Furthermore, the model was used for triggering *business* discussion. In this context the business representatives were able to see *referent system concepts instead of model system concepts;* comments on, for example, multiplicity's in associations, were put forward in the context and linguistic terms of business instead of in object-oriented terminology. In the context of formal reviews, in which participants from the

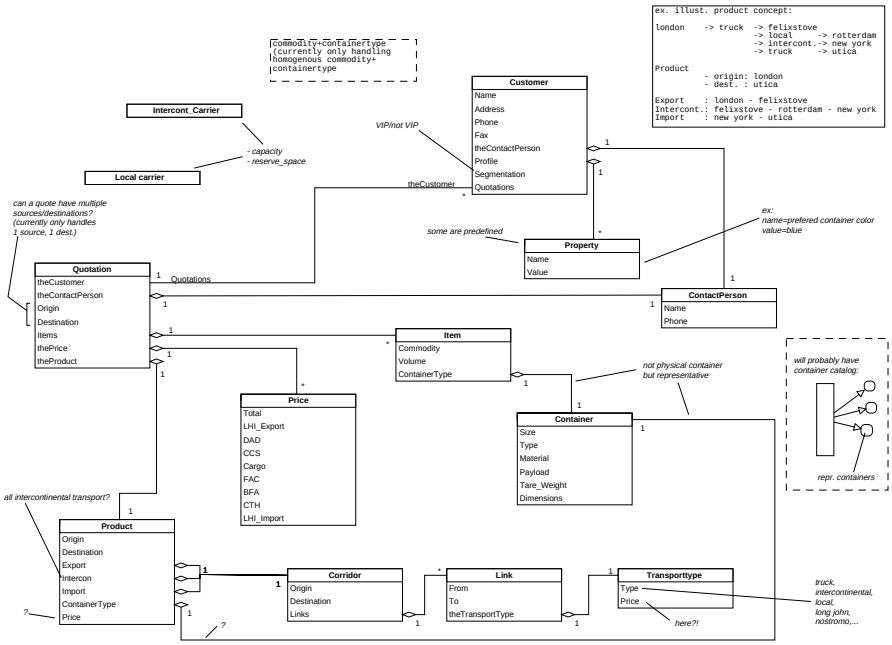


Figure 4. Initial design

[7] The success of 'snapshots' as a modelling technique has resulted in CIT investing in an electronic blackboard that we intend to use in future initial analysis / design sessions. This is just a first step in developing some supportive tools to support initial analysis / design.

business (including the consultants) had been involved in the BPI project and thus had an abstract perception of the business, the model triggered business discussions regarding *future practice*. It should be said that the model was not intelligible in all settings. In other contexts, participants who had not been part the BPI and who did not therefore have some knowledge of formal methodologies of design, had a hard time grasping the whole intent of discussing such a model.

As can be seen in Figure 4, the model exhibits technical simplicity: almost no implementation-specific attributes or classes have been added and only limited subset of UML's class diagram notation (Rational, 98), has been used namely inheritance, association and aggregation. In keeping the model "clean" it is kept interpretable and by using a formal notation the model is also executable in the sense that it is a living thing inside the actual prototype.

5.3 The evolution of the model

The model was by no means finished before coding started. It evolved along two dimensions: horizontally and vertically. Horizontally the model was gradually extended to cope with new areas in the referent system in the manner described above, with the first iteration of the prototype focusing on quoting, then booking and so on. In proceeding in this manner, both the model and the prototype were mutually elaborated. This was also the case vertically. The first iterations of a class or collection of related classes were focused on data and the relationships between the classes. In the following iterations the focus moved to high level functionality (the business functions), which introduced methods to the classes. Development of new business functions occasioned many amendments to the relevant parts of the model. Amendments primarily consisted in the addition of attributes and methods, modification of existing attributes and adjustments in class relationships.

Amendment to the model tended to be at a rather detailed level and our experience shows both that the model does not have to be complete either horizontally or vertically before coding can start, and that one does not need to spend too much time on the details in the design model before coding as they will be changed during implementation. The problem of when to start coding is not a question of validating the model but rather, in our experience, one of putting up and satisfying practical development requirements such as meeting deadlines for formal reviews. It should also be said that a major development requirement from our point of view is to have a running prototype even of minimal design to confront practice with and, in reaction, thereby elaborate modelling issues. Given this, our experience also shows that a model cannot be effectively created in isolation from implementation. As the model evolves together with the prototype in an evolutionary process, it becomes more and more stable with respect to the parts of the referent system so far covered. The Bremerhaven instance illustrates this.

5.4 The Bremerhaven Instance (3)

The problem of rerouting as elaborated through the perspectives and interaction of ethnography and participatory design and as visualised in the notion of a tree structure, emerged as a design issue relatively late in the process. Rerouting was on the agenda for the last review of the first phase of the project. At that time the basic functionality of the prototype had been developed to the point where the model supported a diverse set of business concepts including the concept of booking and related concepts such as products, schedules, allocation etc. The problem of rerouting is all about changing a bulk of bookings, a high-level operation that involves manipulation of multiple sets of bookings and related concepts. In so much as we already had the basic building blocks available, the development of the rerouting functionality was developed with only minor changes to the model. The basic structure did not need to be reconstructed. Radical reconstruction of the model was unnecessary because it was constructed on a natural understanding of the business concepts at work. Having said that, in dealing with the issue of rerouting, a new concept was introduced. The notion of rerouting being akin to a tree with an elaborate network of branches was not discovered among existing business concepts, but emerged out of a joint analysis of the work from the ethnographic and participatory design perspectives. Our approach to modelling allows new concepts to be introduced easily thus allowing the model to develop in an evolutionary manner.

The Bremerhaven instance is also a good example of how instances were used as a communication media between in this case the cooperative designer and the OO developers. Discussions between the cooperative designer and the cooperative developers regarding rerouting were centered around the Bremerhaven instance. The rerouting functionality is rather complex and it is difficult to get it right the first time. When problems arose during implementation the instance was used as a guideance for what at least should work in order to illustrate the basic idea. I.e. the Bremerhaven instance was used as a mediator in the design / implementation cycles as well as a minimal test case.

5.5 Prototype architecture

Working on the assumption that vertical, evolutionary prototyping needs to be performed within some well-defined architecture of the prototype, architecture was designed before actual programming on the prototype began, i.e. within two to three weeks. In this architecture the object model played a central role as being the *common frame of concepts* structuring the code in a referent system specific way. As we mentioned earlier, dealing with the concept of rerouting did not occasion any major reconstruction of the model, however, it did impact upon the architecture. Implementing rerouting in the prototype was problematic in that we did not know if the functionality required was already there, ready to reuse, or if abstractions of existing functionality were required. Through analysing the Bremerhaven instance we anticipated that major parts of the existing routing functionality could be reused. As it turned out however, the existing architecture did not provide the proper abstractions.

Constraints of time meant that the first design / implementation employed ‘copy-paste reuse' of existing routing functionality. This was not construed as a problem in itself because, working in an evolutionary manner, we had the opportunity to design a new architecture in the next iteration cycle. As such, after the first major iteration of the prototype, we designed and implemented a component architecture (see Figure 5). This enabled us to implement rerouting using the cleaner abstractions afforded by the new architecture. These still, of course, need further iteration and as work continues more and more aspects of the Bremerhaven and other instances of work are brought into discussion. Change is now less problematic however, as the current architecture allows us to deal with the complex issues emerging from prototyping sessions: during the trip to Asia, for example, the power of component architecture was experienced as it was actually possible to add major components to the system “on the fly”. The component architecture is the first step in the direction of an architecture with COM objects, (Microsoft, 95).

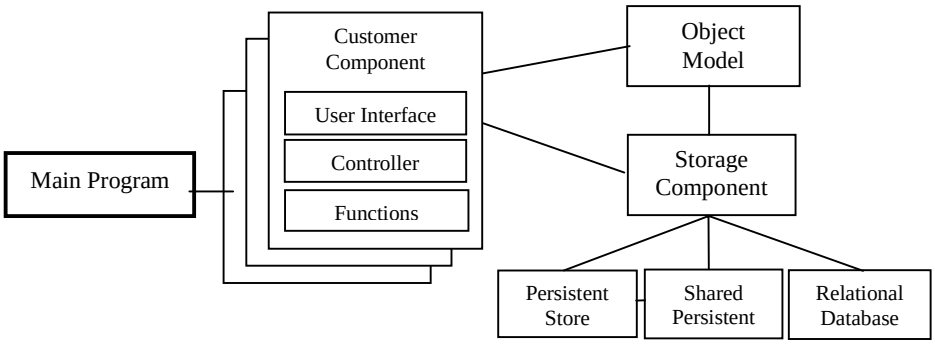


Figure 5. Current architecture of the prototype

5.6 The user interface of the prototype

In the user interface of the prototype we are combining an object-oriented user interface (in so much as part of the model is visualised in the user interface) with an activity-oriented user interface. The screen dump in Figure 6 shows this: business processes are displayed in a sequence of *process buttons* using a simple colour scheme for showing work's state of completion but not enforcing the accomplishment of work in a sequential manner, whereas the *tabbed controls* visualise the actual objects - customer, quote, transport - that are being worked on. Most upper tabs correspond to important domains in the model, whereas the sub-tabs visualise aggregated or referenced concepts.

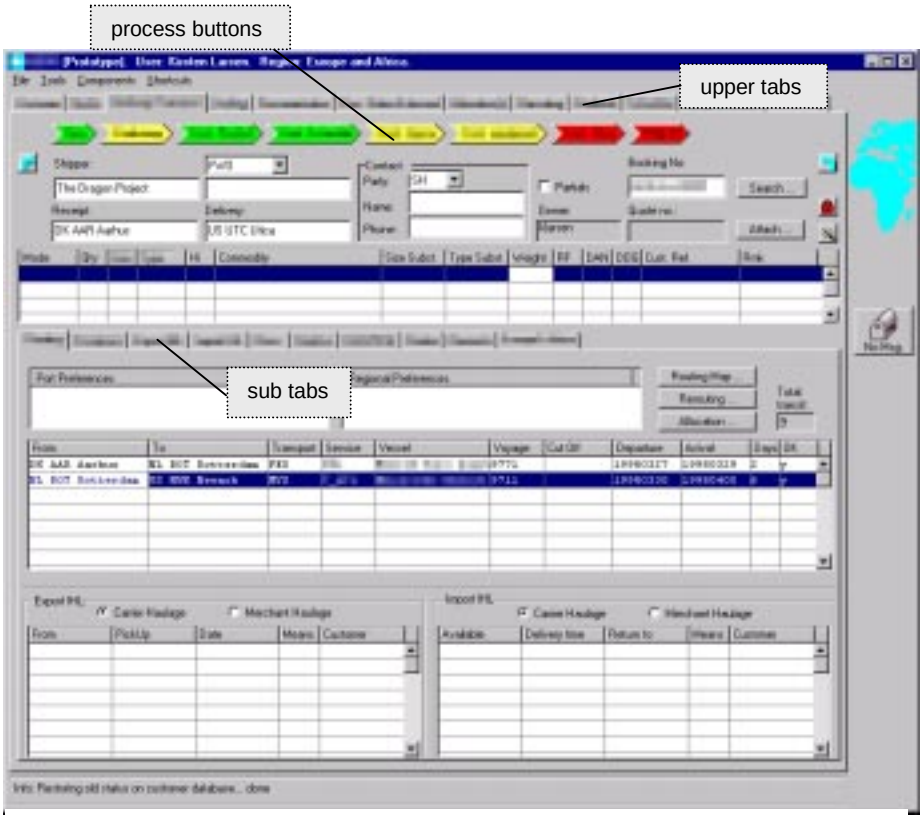


Figure 6. An overview of the main principles in the user interface of the prototype. (Image blurred for confidentiality)

6 Usage of tools

Within the context of any perspective the use of and support by tools is important. Working with evolutionary prototyping of architecture, functionality, user interface and model, tools supporting many iterations over a short period of time are a necessity. In the Dragon Project the main software engineering tools used are a CASE tool, a GUI builder, a code editor, a persistent store and a concurrent engineering tool.

The first four tools are part of the Mjølner System (Knudsen et al. 94, Ch. 2;`http://www.mjolner.com`) and the fifth tool used was concurrent versioning system (CVS, 98). The programming language used is BETA (Madsen et al., 93).

6.1 CASE tool

The CASE tool was used to draw the model using UML notation and to automatically generate the corresponding BETA code. In the beginning the CASE tool was also used to create the unstructured blackboard snapshots that later were used as input for the construction of the model. As diagrams of the model were created or modified, code was generated incrementally. Although code was thus available at any time, the early versions of the model were only used in the form of UML diagrams, as the emphasis was on communication and discussion.

The model can be changed via the diagrams or via the textual code. As emphasis moved to code and redesign cycles, model changes were mostly made in the code editor (see below) – only structural changes requiring overview of the model were made in the CASE tool. When the main emphasis is on coding, it is often more convenient to do the changes in the textual representation, especially when the changes are at a detailed level. Using the reengineering capability, the UML diagram was recreated from time to time and posters of it were made for discussion as mentioned earlier The reverse engineering capability is not based on having any additional information besides the BETA code, e.g. comments or other kinds of annotations, which means that the code editor – or in fact *any* editor – could be used to manipulate the model in its code representation.

6.2 GUI builder

The GUI builder is used to create the user interface in a direct manipulation graphical editor. Like the model the user interface can be changed via the graphical representation or via the textual code. The user interface was created from the start with the graphical editor, but was initially used for discussions only, although code is generated automatically. Changes regarding the physical appearance of the user interface, i.e. what type of UI controls used, and their concrete layout, were typically made in the graphical editor and changes regarding the basic functionality of the user interface and the interface to the model and functionality layer were typically made in the code editor. Due to reverse engineering and incrementality it is easy to alternate between using the two tools.

The user interface was often used as means for organising and/or coordinating activities between the cooperative designer and the OO developers. The initial user interface design was, by and large, made by the cooperative designer and it was then further elaborated by the OO developers in collaboration with the cooperative designer. Again due to reverse engineering it was possible for the cooperative designer to make changes to the user interface throughout the process, even very late in a prototype cycle.

The GUI builder generates code for a platform independent framework. This framework, which also had great importance to the development of the prototype, had to be extended during development in order to support platform specific UI controls. The new UI controls were manually inserted in the code, but they could coexist with

the automatically generated UI controls without affecting the reverse engineering process.

One problem or annoyance was that a large amount of time was spent on coding a strict programming interface to the user interface to achieve independence between the user interface, the model and functionality layer. The separation, however, showed to provide advantages in stability of interface and those advantages are considered to outweigh the difficulties. Furthermore, preliminary investigations show that much of this work can actually be automated (Damm et al., 97).

6.3 Code Editor

All of the tools mentioned above integrate with a structure editor. The editor knows the grammar for the BETA language allowing it to "catch" syntax errors and provide facilities for syntactic and semantic browsing of code.

Furthermore, the editor offers an abstract representation of the code, where any part of the code can be hidden. As became evident when a new developer was introduced to the system (and its architecture) relatively late in the process, this presentation of code and the semantic browsing facilities helped in gaining a quick understanding of the architecture of the relatively large prototype. Also, the abstract presentation of code and the editing operations at that level were helpful in restructuring even large pieces of code (Knudsen et al., 94, chapter 23).

6.4 Persistence

Another major player on the stage of tools was the persistent store that supports orthogonal and transparent persistence. In this way any object can be stored and given a *persistent root* all reachable objects will be made persistent transparently. This meant that only a little time had to be used on persistence issues in the first versions of the prototype. Persistent stores generated from data from existing databases simulated interfaces to legacy systems, and data non-existing in legacy systems was added. A point has been made of identifying those chunks that are within the object model to prepare for current work on interfacing to actual legacy systems. In the current version of the prototype the focus has turned to multi-user functionality in a client / server architecture and a storage component has been developed. The storage component (see Figure 4) constitutes a transparent interface to three different storage media: the initial single-user persistent store, a shared persistent store (Brandt, 93,94; Wiederhold, 97), and a relational database (for a selected part of the data).

6.5 Concurrent engineering tool

Although preliminary implementation was worked out on separate pieces of code the need for use of the versioning system CVS very quickly emerged. As implementation progressed it became evident that its presence was absolutely crucial: It facilitated

seven developers concurrently working on over 300 files containing over 100,000 lines of code, merging code with only minor problems. CVS was also used to merge changes made in regional prototype sessions with changes made at home at the University.

7 Managing development

Keeping the project "on track" and within the agreed time-frame was a major problem to be tackled. Project management and control was achieved in a number of ways. Development activities were managed and coordinated with BPI objectives through frequent business reviews ranging from the informal to the company's highest executive body. Up until now five different types of review, which vary both with respect to participants and with respect to purpose, have been held. A review of some kind has, on average, been held every two weeks during the five months of effective prototype development. In the second part of the project a major review of the prototype has been held approximately every second month. All reviews have actively involved relevant members of the development team. The reviews can be categorised as follows:

- Formal reviews with the project manager from the business, technical advisors from a consultancy and business representatives.
- Informal reviews with the same group.
- Reviews with the company's Regional Programme Coordinators (RPCs).
- Review with the company's executive body.
- Review with members of the company's Business Reference Group (BRG).

The purpose of the formal reviews – which were held more frequently in the beginning of the project than later on in development – was to assess whether or not the prototype was on the right track both in terms of scope and competency[8]. Informal reviews occurred in between the formal ones. They were informal in the sense that no single version of the prototype was built, frozen and presented for the occasion, rather it was a matter of the company's project manager and consultants visiting the development site and discussing work-in-progress[9]. Reviews with regional program-

[8] It might be noted that the "problem of culture" – academics working in a commercial context – was considered to be an issue of "high risk" by the company and its consultants; a problem resolved through frequent reviews.

[9] It should be said that for the first phase of development, the development team was located in company offices at a customer service site. While not actually located with customer service staff, access to practice was greatly facilitated. Furthermore, the transmission of knowledge was greatly facilitated through locating all the developers / perspectives in one room. This co-location supported internal awareness and coordination of development activities. It is an

me coordinators and the ethnographer, who did ‘quick and dirty’ studies of work (Hughes et al., 94) in Hong Kong and the US, were held to ensure that prototype kept a "global perspective" and, in their reaction to the current prototype, to get further input for development.

There has only been one review involving the company’s executive body, the highest authority within the company regarding business development decisions. Following the executive review of the first major iteration of the prototype (May ’97), the company decided to go ahead with developing a production version of GCSS. This decision was, amongst others, made on the basis of formal presentation of the prototype. The Business Reference Group, assembled in response to development phase two, represents the regions and is responsible for accepting features of the prototype that are to go into the production version and organising further, more “specialised” input from regional staff. By “specialised” staff is meant representatives from specific areas of the business. Although changed from time to time, one or two representatives from business have been located with the development team more or less constantly throughout design. They have been used intensively as resources on the problem domain and have also acted as coordinators between system development and BPI.

8 Concluding remarks

Returning to the question posed in the introduction, how come that the project was, and still is, successful? Based on the state of the work – still in progress – we will restrain ourselves from too firm conclusions. However, we still believe we might extract some lessons from the project so far. In the following, and we are very well aware that we are talking on the basis of one albeit comprehensive example, we will try to summarise some of the key findings in explicating some of the underlying means by which the prototype, the process, and the business results were achieved. Tentatively, and again rather simplistically, we can state that the main result seems to be the strength of an approach we might call an experimental and multiperspective approach to designing for practice.

8.1 Practice

Probably the most fundamental principle within our approach is the focus on the work practice the application has to support. One could say that practice is the subject of the analysis, a springboard for design, and the goal of the implementation. The business objective of the project as such is to design a product that fits the practice in which it is to be embedded and used, in this case customer *service*. Thus, the key feature of our approach is the focus on practice which is seen and treated as *the* fundamental resource in that it provides the possibility for grounding design,

---

organisation of work we have maintained in phase two although we are now located at the university.

identifying solutions to substantive problems, as well as providing triggers for new ideas as to how work (e.g. the *delivery* of customer service) might be achieved in the future through technological intervention.

8.2 Experimentation

In so much as design is an intervening activity very much concerned with future practice, one of the key characteristics of the whole development process is the quite comprehensive use of experimentation – i.e. the *performance* of analysis, design and implementation *in active collaboration with users.* Various methodological advise found in textbooks suggest that analysis, design and implementation should be carried out in sequence, although with iterations. The approach conducted within the Dragon Project can almost be said to go to the other extreme – there is no 'sequence' of work in the conventional sense of the word[10]. Schematically, Figure 7 depicts the approach employed and characterises experimentation.

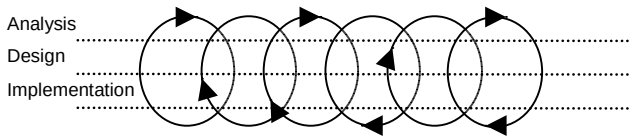


Figure 7.

In the analysis, we made extensive use of artefacts created within design and implementation. Similarly, the design activities were heavily dependent on both an in depth understanding of current practice (analysis) and a firm notion of what could actually be achieved within given constraints (implementation). Needless to say, implementation was dependent on analysis and design – notions all of which in this context are continuous, mutually elaborating and on-going: just as one starts analysing, designing and implementing in conjunction, then so one proceeds until a concrete product emerges

This 'radical parallelism' and experimentation in all phases is important and, in placing users at the centre of design activities, leads to the implementation of a prototype suitable for continuation into the product system. Just as we designed using specific experiences still keeping "the big picture" in focus, we also, and simultaneously, tried to cater for a rapid prototyping process as we developed with the

[10] Of course, in so much as this approach is iterative and in real time demands restructuring of the product, implementation of architectures supporting integration with databases, existing systems and so on, then there is a 'sequence' at work but here we are talking about the development of a technological infrastructure in contrast to the development of a functioning system that the infrastructure serves. It is in the sense of the *process of developing of functionality* (i.e. what activities the system should support) that the notion of sequence becomes redundant as analysis, design and implementation are continuous, mutually elaborative and on-going in contrast to discrete step-by-step tasks.

product system in mind – i.e. we took and take an evolutionary approach to prototyping. Central to this process were the model and the architecture, the development of which naturally requires a collection of suitable tools and techniques.

8.3 Multiperspective

In many respects the defining feature of the process is its interdisciplinary or multiperspective character. As outlined in the previous sections we have made extensive use of three very different perspectives. 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.

Naturally, these perspectives do not have fixed boundaries; for all practical purposes neither seeks to exclude the other. We all need to understand the practice in question, we all need to share the design visions, as we all need to know what is realistic in terms of implementation. On the other hand, for practical purposes, we cannot all comprehend practice to the same extent. Thus, we made extensive use of overlapping competencies and foci as well as overlapping activities and responsibilities.

8.4 Implications for object-orientation

Besides these rather general lessons, we address more specific object-oriented issues. Below is a summary of some of the notions and principles applied in the current project ordered according to where they might best inform object-oriented analysis, design and implementation.

OO analysis is more than finding nouns and verb.

- Analysis is in significant part directed towards understanding current practice.
- Ethnography is a powerful approach to understanding the social organisation of activities which *is* current practice.
- Developers need concrete experiences from within practice, complementary and in addition to bringing users into the development process.
- Prototypes, mock-ups and scenarios, complement ethnographic techniques in functioning as triggers for discussions on current practice with users.

OO design is more than filling in details in the OO analysis model.

- Design is seen as an on-going process of formulating "best matches" between current work and future possibilities.
- Cooperative design bridges between current and future practice by active user involvement in a creative process of experimentation.
- Concrete representations of design visions (prototypes, mock-ups, and scenarios) provide the possibility (1) for simulating future work through hands-on-experience and (2) for (thereby) formulating concrete design-solutions.
- The central 'challenge' in design is not so much to find representative users, rather it is to find users that can challenge representations.

And to complete the list:
OO implementation is more than translating design models into code.

- Implementation is also seen as the process of realising emergent, in contrast to predefined, design visions
- Without understanding design visions and their concrete relationship to practice, it is virtually impossible to implement or find alternatives to formal specifications.
- Implementation is in significant part constructing primary *means* for analysis and design in accomplishing evolutionary prototyping.
- The construction of robust yet readily adaptable models and architectures are crucial in implementation and depend on flexible tool support.

Naturally, the above principles do not apply to all problems in all situations. System development is, afterall, a heterogeneous enterprise not only in terms of staff but also in terms of problem domains. Up until now each of the individual perspectives outlined here have been successfully applied to a wide variety of application developments in a multiplicity of situations. Although respective

disciplinary achievements suggested the strong possibility of developing a highly effective and unified approach to system development, in so much as this is the first major attempt at combining them in large-scale development, then success in a multiplicity of settings cannot be claimed for their particular association. Thus, it is difficult to assess scope, applicability, cost, etc.

What we can say, based on the experiences so far, is that the approach has been successfully applied in a situation characterised by the following features: complex human work practices, high uncertainty regarding the specifics of the potential application, large and geographically distributed organisation. In respecifying the classical working order of design from a sequential process of analysis – design – implementation to an on-going, mutually elaborative process dependent on active user involvement (experimentation), the multiperspective approach outlined here is potentially strong, both in terms of projected cost benefit and in actual terms of practical efficacy from a client's point of view, in supporting the integration of emerging information technologies into the world of work and organisation. To reiterate: we outline here an organised approach to, *not a formal method of*, work-oriented design. Finally, it might be said that in so much as we have explicated the acronym M.A.D. in what we hope is some reasonable detail, then that acronym not only captures the essence of a unique approach but also the frenetic character of rapid prototyping at work.

Acknowledgement: This work was made possible by the Danish National Centre for IT-Research (CIT, `http://www.cit.dk`), research grant COT 74.4. We would also like to express our sincere thanks to all the people within the company who made this project possible.

9 References

(Anderson et al., 89). Anderson, R.J., Hughes, J.A., Sharrock, W.W. (1989) *Working for Profit: The Social Organisation of Calculation in an Entrepreneurial Firm*, Aldershot: Avebury.

(Bannon, 91) Bannon, L.J. (1991) *From Human Factors to Human Actors: the role of psychology and human-computer interaction studies in system design*, Design at Work: Cooperative Design of Computer Systems (eds. Greenbaum, J. & Kyng, M.), pp. 25 – 44, New Jersey: Lawrence Erlbaum Associates.

(Bannon et al., 93) Bannon, L.J. & Hughes, J.A. (1993) *The Context of CSCW*, Developing CSCW Systems: Design Concepts, Report of COST 14, 'CoTech' Working Group 4 (1991-92), pp 9 – 36.

(Bjerknes et al., 96). Bjerknes, G., Ehn, P., & Kyng, M. (1987) *Computers and Democracy: A Scandinavian Challenge*, Aldershot: Avebury.

(Blomberg et al., 94). Blomberg, J., Suchman, L., Trigg, R. (1994) *Reflections on a Work-Oriented Design Project*, Proceedings of PDC '94,pp 99 – 109, Chapel Hill, North Carolina: ACM Press.

(Brandt 93). Brandt, S., Madsen, O.L. (1994) *Object-oriented Distributed Programming in BETA*, in Lecture Notes in Computer Science, LNCS 791, Springer-Verlag 1994.

(Brandt 94). Brandt, S. (1994) *Implementing Shared and Persistent Objects in BETA*, Progress Report, Tecnical Report, Department of Computer Science, Aarhus University.

(Bødker, 91) Bødker, S. (1991) *Through the Interface: a Human Activity Approach to User Interface Design*, Hillsdale, New Jersey: Lawrence Erlbaum Associates.

(COMIC 2.2, 93) COMIC Deliverable 2.2, Esprit Basic Research Project 6225 (1993) *Field Studies and CSCW*, (eds.) Lancaster University and Manchester University.

(Crabtree et al., 97). Crabtree, A., Twidale, M., O'Brien, J., Nichols, D.M. (1997) *Talking in the Library: Implications for the Design of Digital Libraries*, Proceedings of ACM Digital Libraries '97, Phildelphia: ACM Press

(CVS, 98). Gnu, *Concurrent Version System* (1998) ftp://archive.eu.net/gnu/.

(Damm et al., 97) Damm, C.H., Hansen, K.M., Thomsen, M. (1997) *Issues from the GCSS Prototyping Project – Experiences and Thoughts on Practice*, Department of Computer Science, Aarhus University.

(Garfinkel et al., 70) Garfinkel, H. & Sacks, H. (1970) *On Formal Structures of Practical Actions*, Theoretical Sociology: Perspectives and Developments (eds. Mckinney, J.C. & Tiryakian, E.A.), pp 337 – 366, New York: Appleton-Century-Crofts, 1970.

(Greenbaum et al., 91). Greenbaum, J., & Kyng, M. (1991) *Design at Work: Cooperative Design of Computer Systems*, Hillsdale New Jersey: Lawrence Erlbaum Associates.

(Grønbæk et al., 93). Grønbæk, K., Kyng, M., & Mogensen, P. *CSCW Challenges: Cooperative Design in Engineering Projects*, Communications of the ACM 36 (6), pp 67 - 77.

(Grønbæk et al.,). Grønbæk, K., Kyng, M., & Mogensen, P. *Toward a Cooperative Experimental System Development Approach*, In M. Kyng & L. Mathiassen (Eds.), (In Press).

(Grønbæk, 91). Grønbæk, K. (1991) *Prototyping and Active User Involvement in System Development: Towards a Cooperative Prototyping Approach*. Ph.D. Thesis, Computer Science Dept., University of Aarhus.

(Grudin, 89) Grudin, J. (1989) *Why Groupware Applications Fail: Problems in Design and Evaluation*, Office: Technology and People, vol. 4 (3), pp 245 – 264.

(Heath et al., 92). Heath, C. & Luff, P. (1992) *Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Line Control Rooms*, JCSCW '92, vol. 1, the Netherlands: Kluwer Academic Publishers.

(Hughes et al., 92). Hughes, J., Randall, D., Shapiro, D. (1992) *Faltering from Ethnography to Design*, Proceedings of CSCW '92, pp 115 – 122, Toronto: ACM Press.

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

(Hughes et al., 96). Hughes, J., Kristoffersen, S., O'Brien, J., Rouncefield, M. (1996) *When Mavis met IRIS: Ending the love affair with Organisational Memory*, Proceedings of IRIS 19 'The Future', Report 8.

(Jackson, 83). Jackson, M. *System Development*, Englewood Cliffs NJ: Prentice-Hall, 1983.

(Jacobsen, 92). Jacobsen, I. (1992) *Object-Oriented Software Engineering. A Use Case Driven Approach*, Addison Wesley.

(Kensing et al., 97). Kensing, F & Simonsen, J. (1997) *Using Ethnography in Contextual Design*, Communications of the ACM, 40 (7), pp 82 - 88.

(Knudsen et al., 94). Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (1994) *Object-Oriented Environments. The Mjølner Approach*, Prentice Hall.

(Knudsen et al., 96). Knudsen, J.L., Madsen, O.L. (1996) *Using Object-Orientation as a Common Basis for System Development Education*, ECOOP '96 Teachers Symposium.

(Madsen et al., 93). Madsen, O.L., Møller-Pedersen, B., Nygaard, K. (1993) *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley.

(Madsen, 96). O.L Madsen: Open Issues in Object-Oriented Programming (1996) – a Scandinavian perspective, Software Practice and Experience.

(Microsoft, 95). *The Component Object Model Specification*, Microsoft Corporation, 1995.

(Mogensen, 94). Mogensen, P. (1994) *Challenging Practice: an Approach to Cooperative Analysis*, Ph.D thesis, Computer Science Department, University of Aarhus, Daimi PB-465.

(Rational, 98). Rational Software Cooperation (1998) *UML Notation Guide Version 1.1*, http://www.rational.com/uml/html/notation/

(Rouncefield et al., 94). Rouncefield, M, Hughes, J.A., Rodden, T, Viller, S. (1994) *Working with "Constant Interruption": CSCW and the Small Office*, Proceedings of CSCW '94, Chapel Hill: ACM Press.

(Schmidt et al., 93) Schmidt, K. & Carstensen, P. (1993) *Bridging the Gap: Requirements Analysis for System Design*, Working Paper, COMIC-RISØ, Esprit Basic Research Project 6225, (eds.) Lancaster
University and Manchester University.

(Twidale et al., 94) Twidale, M., Randall, D., Bentley, R. (1994) *Situated Evaluation for Cooperative Systems*, Proceedings of CSCW '94, pp 441 – 452, Chapel Hill, North Carolina: ACM Press.

(Weiderhold, 97) Weiderhold, J.T. (1997) *A Multi-User Persistence Framework: Building Customised Database Solutions uing the BETA Persistent Store*, MA. Thesis, Department of Computer Science, Aarhus University.

(Wiklund, 94). Wiklund, M. (1994) *Usability in Practice*, AP Professional.

(Wittgenstein, 68) Wittgenstein, L. *Philosophical Investigations*, Oxford: Basil Blackwell, 1968.

In Proceedings of TOOLS Asia'1999

The 'Domain Model Concealer' and 'Application Moderator' Patterns: Addressing Architectural Uncertainty in Interactive Systems

Klaus Marius Hansen and Michael Thomsen
Department of Computer Science, University of Aarhus,
Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: {marius, miksen}@daimi.au.dk

Abstract

Designing architectures for interactive systems is difficult: Many system failures are due to the inability to handle social and technical changes that occur during development. We present two architectural patterns for interactive systems. We applied these to a short-term and a long-term development project. In the WebviseLT Project, the conceptual model was modified to handle various extensions and to meet an emerging standard. In the Dragon Project, the user interface evolved significantly over a period of almost two years. The application of these two architectural patterns demonstrates how focusing on change, or more specifically on architectural uncertainty, can be crucial to the success of a project.

1. Introduction

Brooks' [7] vivid description of the great beasts trying to escape the tar pits evokes the problems faced by development teams entangled in failure. Because design of software architecture is the highest level of design in software development, architectural uncertainty causes significant risks. One way of addressing failure is through 'risk management' as in Boehm's *Spiral Software Development Model* [5], an iterative and evolutionary software development model. Through 'risk management' one should carefully consider what the uncertainties are, in which way these could constitute risks and what should be done to handle them. To avoid failure, special attention should be paid to architectural uncertainties.

For the purpose of our discussion of software architecture, we will use the definition of Bass et al. [2]:

> *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 order to overcome the limitations of box-and-line diagrams encountered in some software architecture discussions, we will use a variation of the notation introduced in Bass et al. [2]. Solid lines denote control and processing, whereas dashed lines denote data (Figure 1). We introduce the starred symbol to denote an architectural uncertainty. Furthermore, a shadow on a component indicates one or more components of that kind.

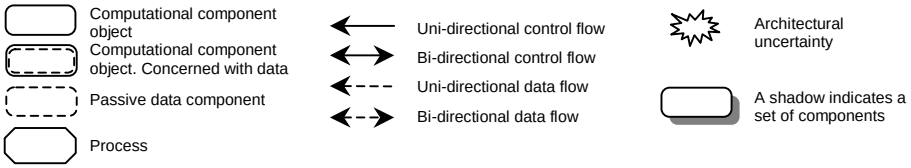


Figure 1. Software architecture notation.

1.1. Architectural uncertainty

We define uncertainty within a software architecture as follows:

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

Based on our experiences from two development projects, we discuss two important examples of architectural uncertainty: In the WebviseLT case, the Webvise hypermedia system was to be extended with typed links. This potentially required a change of the Webvise server's object-oriented model of hypermedia, while this model was about to be standardised. From the Webvise clients' view, the server's model thus presented an architectural uncertainty. In the Dragon case, the development process employed a high level of participation by users, which implied frequent changes to the user interface. These change requests forced us to continually evolve the part of the application that implemented the user interface. Thus, the user interface presented an architectural uncertainty with respect to the components that communicated with it.

The structure of the paper is as follows: In section 2, we present and discuss the two case studies of object-oriented system development; each addressing a different architectural uncertainty. Section 3 presents two architectural patterns, and section 4 discusses how these patterns were applied in each case to address the architectural uncertainty. Section 5 concludes.

2. Case studies

Each of the following case studies involves an architectural uncertainty that posed problems in the software development process.

2.1. WebviseLT: Extending the World Wide Web with typed links

The *WebviseLT Project* is a short-term research project that has currently run half a year. It extends the Webvise hypermedia system [23] that provides open hypermedia facilities for third party applications. In open hypermedia systems, structures such as links and anchors are external to the documents they link. This means that users may link from and anchor in documents they do not own. Unlike most other open hypermedia systems, Webvise also integrates with the World Wide Web. The World Wide Web is a single homogenous space of information, and its sheer size magnifies the classical problem of "being lost in Hyperspace" [14].

Link types, or more generally types for objects in hypertexts, provides the user with a manageable conceptual model in a hypertext [29]. Adding link types, also to the World Wide Web, is thus useful:

- for adding (user-defined) semantics to a hypertext,
- as a means of increasing support for user orientation by making the semantics of structures explicit, and
- as focal points for computer based analysis and synthesis of hypertexts.

Based on these considerations, part of the WebviseLT Project aims at extending the Webvise system with facilities for creation, manipulation, and use of typed links in media supported by Webvise, including World Wide Web documents [24].

Architectural uncertainty in the WebviseLT Project: Figure 2 shows the Webvise Architecture. The main parts of the Webvise system are the Webvise Client processes, the Webvise Server process, and the external applications. The server, written in BETA [25], implements an object-oriented model of a hypertext consisting of objects such as links and anchors. The client provides a generic interface for general hypermedia functionality such as link following and browsing. Also, the client implements communication with external applications such as Microsoft Word or Microsoft Internet Explorer, effectively enhancing these with hypermedia functionality.

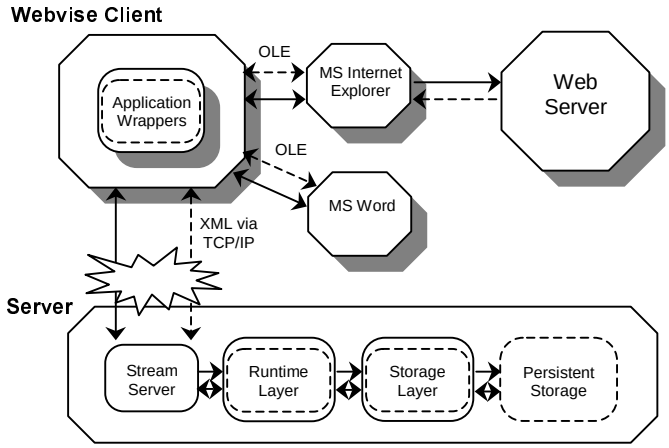


Figure 2. Webvise architecture

To support typed links, it would seem ideal to extend the storage layer of Webvise by creating a specialisation of the link class for each type. This is, however, not a very good idea since the future architecture of the server is uncertain: The server component will be replaced with the Construct system [35] in order to comply with a recent standard of the Open Hypermedia Systems Working Group (OHSWG, `http://www.ohswg.org/`). This architectural uncertainty represents a major obstacle to the introduction of typed links, since the server determines the conceptual view of a hypertext that the clients need to present to the users. Thus, it becomes important to introduce typed links in a way such that the developed functionality can coexist with both the original Webvise server and the new Construct server.

2.2. Dragon: Prototyping a global customer service system

The *Dragon Project* was initiated in 1997 as a joint project between a large, globally distributed container shipping company and University of Aarhus, Denmark. Over a period of more than 18 months, the goal of the project was to create a series of prototypes of a world-wide customer service system. The system should thus support handling of interaction with customers, e.g. in formulating prices for transport of containers (quoting), in booking containers, in arranging inland transportation of containers, in documenting ownership of cargo, and in notifying the consignee of cargo's arrival. The development process was iterative, exploratory, and multi-perspective [12].

Figure 3 shows how each concern of the project was treated iteratively. Booking, for example, was the subject of more than six reviews, with iterations in more than six sub-phases of development. Moreover, in-between each review, we explored several ways of treating the concerns. The development project involved several perspectives: object-orientation, participatory design [21], and ethnography [4].

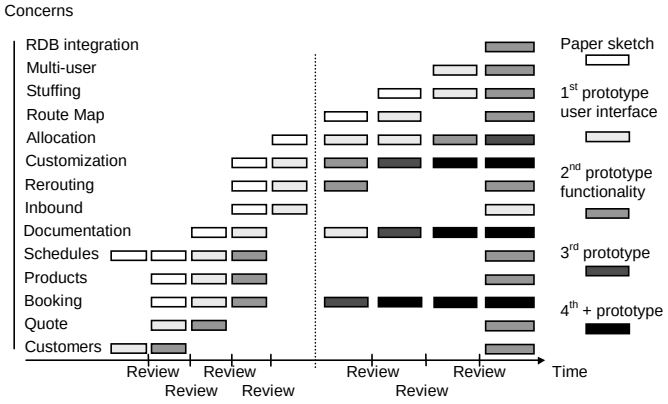


Figure 3. Evolution of concerns in the Dragon Project

The ethnographer worked on understanding and describing current practice, the participatory designer designed technological support for future practice with users, and the OO developers were responsible for the problem domain model, software architecture, and implementation [13].

Architectural uncertainty in the Dragon Project: Figure 4 shows a conceptual view of the initial architecture of the Dragon prototype. It consisted of four major components: Problem Domain Objects related to the shipping problem domain, Business Functions implementing functionality cross-cutting the Problem Domain Object structure, User Interface, and Persistent Storage.

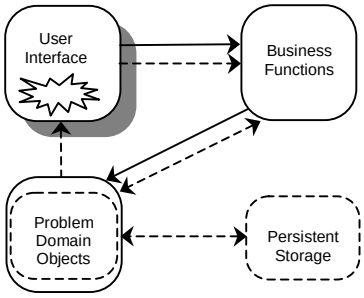


Figure 4. Conceptual view of the Dragon prototype architecture

The iterative and exploratory character of the development process meant that the user interface, problem domain model, and business functions all evolved throughout the project. For each area of the business that was covered, the problem domain model evolved horizontally to cover most phenomena of interest within a short period of time. In this way, the different areas of the problem domain model quickly became structurally stable; later changes tended to be much more vertical in the sense of elaborating on existing classes and relationships rather than introducing new ones. The user interface, however, continued to evolve more radically, as the participatory designer worked with users on new areas of the business. In this way, the instability of the user interface was an architectural uncertainty: The relative difference in stability was problematic, and the development of business functions, problem domain model, and user interface had to take this difference into account. Most important, the architecture had to be designed so that the evolution of the user interface would not overly affect the rest of the system.

3. Architectural patterns for interactive systems

Each of the case studies highlights an architectural uncertainty. In each situation, the uncertainty and potential risks were, in part, handled by designing a software architecture that anticipated the possible change implied by the uncertainty. In this section we describe these architectures. To give a more detailed understanding of the architectures, and to allow the architectures to be used in other contexts, we present them as *architectural patterns*. Architectural patterns, like design patterns [20], 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. [9], and Shaw [32].

3.1. 'Domain Model Concealer' and 'Application Moderator'

This section presents the 'Domain Model Concealer' and the 'Application Moderator' patterns. Both pattern descriptions assume that an object-oriented approach to software development is used. In object-oriented development, the system must not only be technically sound, but it must also maintain a real-world reference, i.e. it must contain a proper model of what is referenced [27], [25], [34]. This model is often referred to as the *problem domain model*. Since we are dealing with interactive systems, another important part of the application is the user interface. These two parts of the application should be separated into two different parts or components, making each easier to understand and maintain.

In our pattern descriptions below, when discussing sample implementation, we will refer to the same *Financial History* application. This facilitates comparison of the two patterns. The, somewhat artificial, application enables tracking of financial expenses. The simple problem domain model is shown in Figure 5 and the user interface is shown in Figure 6.

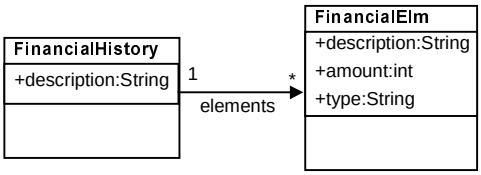


Figure 5. Problem domain model for the Financial History application

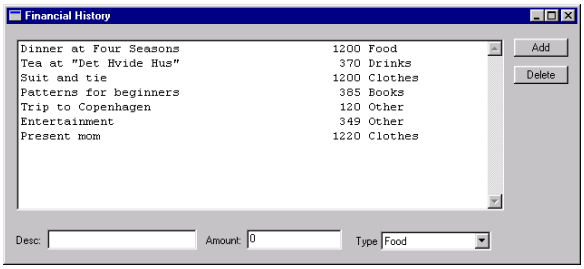


Figure 6. User interface of the Financial History application

3.2. Pattern format

The pattern format used to describe the architectural patterns in this article varies somewhat from the format of Buschmann et al. [9] and Shaw [32]. 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 7 presents the pattern format, which may be viewed as a condensed version of the format in Gamma et al. [20] or as a slight variation on Brown et al.'s [8] '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 (UML [31]) 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 7. Architectural pattern format

3.3. Domain Model Concealer

The Domain Model Concealer architectural pattern divides interactive applications into a problem domain related part (the Problem Domain Model), an interface to this model (Domain Model Concealer), a User Interface part containing the user interface, and a User Interface Logic part that implements user interface related functionality by communicating with the Problem Domain Model through the Domain Model Concealer.

Problem: How does one design the architecture of an interactive system so that the user interface functionality is separated from the problem-domain related functionality? How does

one ensure that frequent changes to the Problem Domain Model do not require frequent changes to the whole application?

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 problem domain model should have no or minimal impact on the rest of the application.

Solution: The Domain Model Concealer pattern divides the application into four parts: User Interface, User Interface Logic, Domain Model Concealer, and Problem Domain Model. Optionally, a Testing component can be implemented. The overall structure is shown in Figure 8.

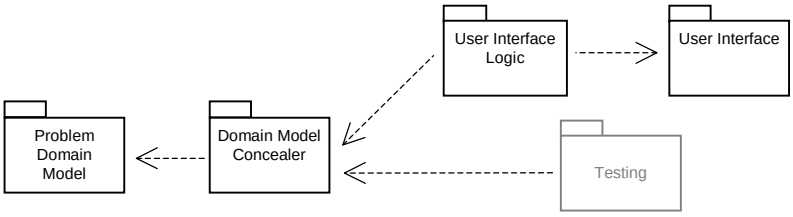


Figure 8. Overall structure of the Domain Model Concealer pattern

The User Interface component implements the user interface and consists of declarations and instantiations of the widgets in the user-interface of the application. The Problem Domain Model contains the domain-related classes and functions, and possibly other functionality. It can be implemented independently of the User Interface component. The Domain Model Concealer contains an interface to the Problem Domain Model component. The Domain Model Concealer is thus dependent on the Problem Domain Model component but independent of the User Interface component.

The User Interface Logic implements the functionality offered by the User Interface component by collaborating with the Problem Domain Model component through the Domain Model Concealer interface. This makes the User Interface Logic component dependent on both the Domain Model Concealer and the User Interface components. Finally, the Testing component allows testing of the Problem Domain Model component via the Domain Model Concealer component.

Sample implementation: An application of the Domain Model Concealer pattern 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 9.

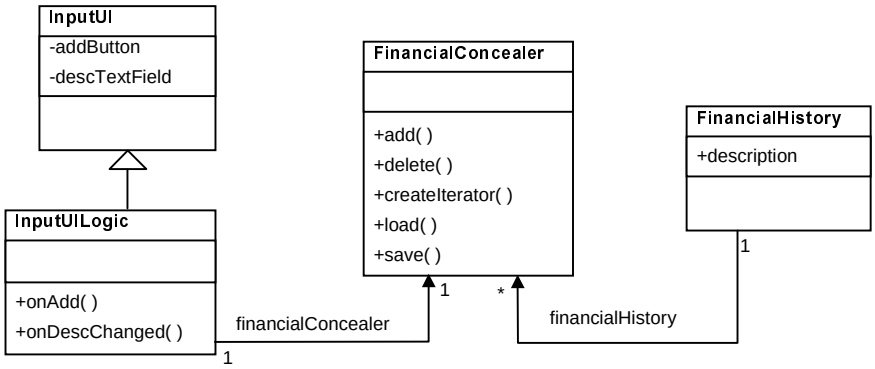


Figure 9. Class diagram for the Financial History application

1. Implement the Problem Domain Model

The Problem Domain Model is as described above.

2. Implement the Domain Model Concealer

The Domain Model Concealer is implemented as the `FinancialConcealer` class. In the Financial History application, operations to add financial elements (`add`), delete financial elements (`delete`), and iterate over financial elements (`createIterator`) are needed. The Concealer should hold a reference (`financialHistory`) to the object in the Problem Domain Model of which it is currently showing the state. Furthermore, methods to set the state of the Concealer (`load`) and to set the state of the Problem Domain Model through the Concealer (`save`) are needed.

3. Implement the User Interface and User Interface Logic Components

The User Interface component (`InputUI`) can be implemented using a user interface toolkit in the normal way. It contains the declarations and instantiations of the widgets and corresponds to what may be generated by a user interface builder. The User Interface Logic component (`InputUILogic`) implements the actual functionality of the User Interface by subscribing to events in the User Interface. To communicate with the Problem Domain Model it uses the associated `FinancialConcealer`.

4. Optionally implement the Testing Component

A Testing component can be created to systematically test the Problem Domain Model via the Domain Model Concealer.

Consequences: This architectural pattern has both benefits and liabilities.

Benefits:

- Separation of the parts that require domain knowledge and the parts that require user interface toolkit knowledge is supported.
- Change in technology is supported, since the User Interface and User Interface Logic components can be substituted by new components when the user interface toolkit changes.
- Testing of the Problem Domain Model and the Domain Model Concealer is facilitated.
- The User Interface and User Interface Logic components are shielded from changes in the Problem Domain Model, since they do not interface with it directly.

Liabilities:

- As the Problem Domain Model changes, either more work is involved in mapping between the Problem Domain Model and the Domain Model Concealer, or the Domain Model Concealer must be changed, which may require changes to the User Interface Logic.
- Multiple view consistency is not accommodated. The Observer pattern [20] can provide this by having the Domain Model Concealer as subject and several User Interface Logic components as Observers.
- Frequent changes to the User Interface also imply frequent changes to the User Interface Logic component.

3.4. Application Moderator

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 10.

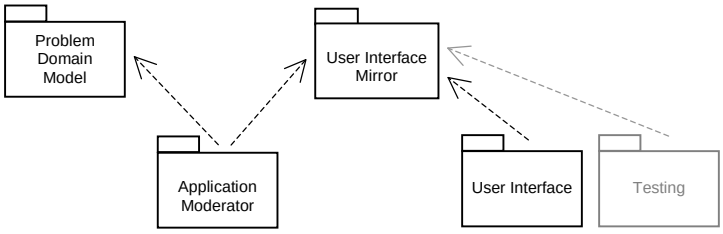


Figure 10. Overall structure of the Application Moderator pattern

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: An application of the Application Moderator pattern 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 11 and Figure 12.

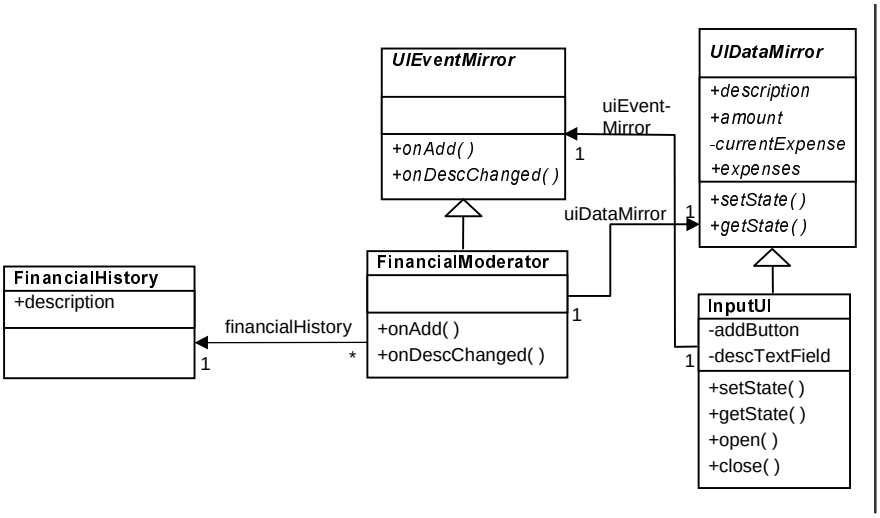


Figure 11. Class diagram for the Financial History application

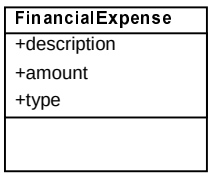


Figure 12. `FinancialExpence` data interface

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 11) 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 12). 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 `UIDataMirror` 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 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.

3.5. Pattern discussion

Known uses: Three-tier client-server architectures [33] divide (interactive) applications into three tiers: Database, Domain, and Client tier, according to External, Conceptual, and Internal schemas. In an interactive application with a thin client the representation of the domain on the client in many ways resemble a Domain Model Concealer, and the Domain and Database tiers resemble the Problem Domain Model in the Problem Domain Model Concealer Pattern. Fowler [19] discusses an 'Application Facade' architectural model in a non-pattern format. In this architecture, the Application Facade acts as a Domain Model Concealer and is used in the same way as in the Domain Model Concealer pattern. In section 4.1, we discuss another use of the Domain Model Concealer pattern.

The Devise Open Hypermedia System [22] uses the Application Moderator pattern extensively to divide user interface and the hypermedia data model in the client (runtime layer) of the system. Each dialogue in the system communicates with the non-client layers through a mirror of the hypermedia data model as in the Application Moderator pattern. In Microsoft Visual C++ [28], the Document/View architecture uses the Application Moderator to separate the user interface (View) and the domain model (Document). The class wizard in the environment generates implementations of the `setState` and `getState` methods of what corresponds to the User Interface Mirror. In section 4.2, we show another use of the Application Moderator pattern.

Relation to other architectural patterns for interactive systems: A large number of architectures have been proposed for interactive systems. Examples include Seeheim [30], Model-View-Controller (MVC [26]), Presentation-Abstraction-Control (PAC [15]), Arch/Slinky [3] and PAC* [11]. However, few of these have been treated in a pattern form. The most well known patterns are probably the MVC and the PAC patterns described by Buschmann et al [9].

Both the Application Moderator and Domain Model Concealer patterns complement the Model-View-Controller (MVC) pattern. The MVC pattern is mostly concerned with obtaining multiple view consistency. Whereas it decouples the Model from the Views and Controllers via the Observer, it allows the Views and Controllers direct access to the Model. Neither the Domain Model Concealer nor the Application Moderator allows this. In the Domain Model Concealer pattern, the Concealer shields the Problem Domain Model. In the Application Moderator pattern, only the Moderator, and not the User Interface Mirror, has access to the Problem Domain Model.

The Presentation-Abstraction-Control (PAC) pattern is different from the other three patterns. PAC has as its dividing principle a vertical split of the application such that it is divided into separate agents according to divisions found in the problem domain. Each of the agents contains their own user interface and functionality.

A main difference between the Domain Model Concealer and Application Moderator lies in what is decoupled. The Application Moderator architecture adds an interface layer to the user interface, thereby shielding the clients of the user interface from changes in it. The Domain Model Concealer architecture does almost the opposite: It adds an interface to the Problem Domain Model and in that way hides changes from its dependants.

4. Managing architectural uncertainty

This section discusses the architectural uncertainties introduced in section 2 and explains how these were handled in the two concrete cases with the help of the patterns presented in the previous section.

4.1. Deferring conceptual choice in WebviseLT: Applying the Domain Model Concealer pattern

A common problem encountered in system development is changes in technology imposed by external driving forces. In the WebviseLT case, this was experienced: The emerging OHSWG standard meant that the server component of the Webvise system was to be replaced. Since the server implements an object-oriented model of hypermedia, the introduction of typed links became problematic. Two choices were possible:

- Wait for the standard, then design the extensions, and subsequently implement the design, or
- proceed, temporarily ignoring the potential changes caused by the standard.

Both choices are problematic. Waiting would cause the project to fall behind schedule and ignoring the upcoming changes would lead to a potential re-implementation. We followed a third path. We decided to develop new functionality while the hypermedia model was evolving, and applied the Domain Model Concealer architectural pattern. Figure 13 shows the introduction of a Link Type Concealer as part of the Webvise architecture. The only way components in the Webvise client application can use link types is through the Link Type Concealer. This concealer thus provides both link type browsers and editors with a stable view

of link types. Also, external applications, such as Microsoft Internet Explorer, are enhanced with link type functionality via the Application Wrappers' use of the Link Type Concealer.

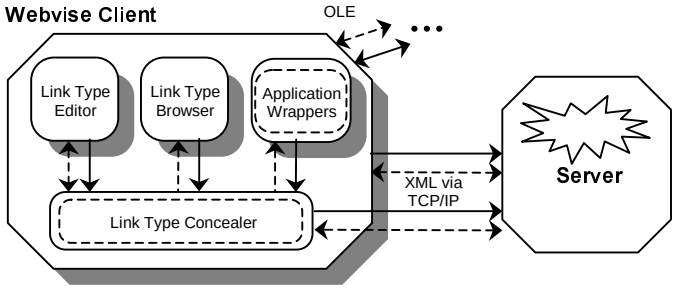


Figure 13. Introducing link types in Webvise

Currently, the link type hierarchy itself and link types are encoded in general attribute/value pairs stored in the hypertext objects. This has two consequences: First, the standardised model also specifies general attribute/value pairs on hypermedia objects. This means that after replacing the server, the only parts of the link type that has to be changed are the actual requests to the server. Second, this scheme should allow for a graceful introduction of first class types that will eventually extend the storage layer of the new standardised server.

4.2. Parallelism in the Dragon Project: Applying the Application Moderator pattern

User involvement in system development generally takes one of two forms. Either

- user involvement is primarily concentrated on requirements gathering at the beginning of a project, and acceptance testing at the end of a project, or
- user involvement and user testing is on-going and involves iterative development of the system.

Both approaches are problematic: User testing only in the end of a development project often causes missed deadlines due to users' rejection of the developed system. On-going user involvement, on the other hand, often causes missed deadlines due to evolving user requirements.

In the Dragon Project, user involvement was on-going, with change caused by evolving or emerging requirements that came from three major sources:

- Usability studies,
- participatory design workshops with users, and
- presentations to and discussions with managers and end users.

The frequent changes to the user interface throughout system development constituted a major architectural uncertainty. Figure 14 shows how the Application Moderator pattern addresses this by isolating user interface changes. The 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.

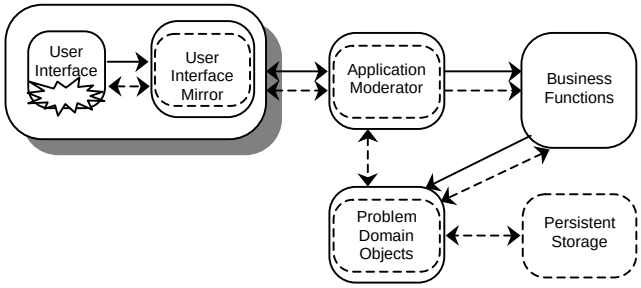


Figure 14. Isolating user interface changes in Dragon

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. The result was that the series of prototypes all were developed on schedule.

5. Conclusion

Software systems must evolve and change because of changing user requirements and external factors, such as emerging standards. This often leads to architectural uncertainties, which must be identified and addressed. This paper reports our experience from two system development projects that addressed architectural uncertainty, by constructing architectures anticipating change.

The two presented architectural patterns were used to isolate and thus handle the architectural uncertainties and both helped in the two projects being finished successfully and on time. Generally, the two patterns are concerned with the coupling of user interfaces and problem domain models of object-oriented applications. While no pattern can remedy the impact of every change, they can provide some independence: The ‘Domain Model Concealer’ pattern can help in shielding from changes to the problem domain model, and the ‘Application Moderator’ can decrease the implications of changes to the user interface. Thus, the architectures documented in the patterns can increase the flexibility of the application and help in handling certain kinds of architectural uncertainty.

6. Acknowledgements

Part of this work has been carried out in Center for Object Technology (`http://www.cit.dk/COT`) which is a project that has been partially funded by the Danish National Centre for IT Research (`http://www.cit.dk`) and the Danish Ministry of Industry. Thanks to our colleagues that participated in the two projects. Thanks also to Henrik B. Christensen, Aino Cornils, Ole L. Madsen, Henrik Røn, and Lennert Sloth for valuable comments that improved this paper. Special thanks to Wendy E. Mackay for improving our English.

7. References

[1] Andrews, K., Kappe, F., Maurer, H. (1995). Serving Information to the Web with Hyper-G. In Computer Networks and ISDN Systems, 27(6).

[2] Bass, L., Clements, P., Kazman, R. (1998). *Software Architecture in Practice*. Addison Wesley Longman.

[3] Bass, L., Faneuf, R., Little, R., Mayer, N., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S., Szczur, M.R. (1992). A Metamodel for the Runtime Architecture of an Interactive System. In *SIGCHI Bulletin, 24*(1).

[4] Blomberg, J., Suchman, L., Trigg, R. (1994). Reflections on a Work-Oriented Design Project. In *Proceedings of Third Biennial Conference on Participatory Design (PDC '94)*. Chapel Hill, North Carolina, USA.

[5] Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer, 21*(5).

[6] Booch, G., Jacobson, I., Rumbaugh, J. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

[7] Brooks, F. P. (1975). The Tar Pit. In *The Mythical Man-Month*. Addison-Wesley.

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

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

[10] Bush, V. (1945) As We May Think. *Atlantic Monthly*, July 1945.

[11] Calvary, G., Coutaz, J., Nigay, L. (1997). From Single User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW. In Proceedings of Conference on Human Factors in Computing Systems (CHI'97). Atlanta, Georgia, USA.

[12] Christensen, M., Crabtree, A., Damm, C.H., Hansen, Klaus. M.H., Madsen, O.L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M. (1998a). The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98)*. Brussels, Belgium.

[13] Christensen, M., Damm, C.H., Hansen, K.M., Sandvad, E., Thomsen, M. (1998b). Architectures of Prototypes and Architectural Prototyping. In *Proceedings of the Eight Nordic Workshop on Programming Environment Research (NWPER'98)*. Bergen, Norway.

[14] Conklin, J. (1987) Hypertext: An Introduction and Survey. In *IEEE Computer 20*(9).

[15] Coutaz, J. (1987) PAC, an Object-Oriented Model for Dialog Design. In *Proceedings of IFIP INTERACT'87*.

[16] Damm, C.H., Hansen, K.M., Thomsen, M. (1997). *Issues from the GCSS Prototyping Project – Experiences and Thoughts on Practice*. Technical Report, Department of Computer Science, University of Aarhus.

[17] Engelbart, D. (1984). Authorship Provisions in AUGMENT. In *Proceedings of the 1984 COMPCON Conference*.

[18] Fowler, M. (1997). *Analysis Patterns*, Addison-Wesley.

[19] Fowler, M. (1999). *Application Facades*. [Online]: `http://www.awl.com/cseng/titles/0-201-89542-0/apsupp/appfacades.pdf`.

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

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

[22] Grønbæk, K., Hem, A.H., Madsen, O.L., Sloth, L. (1994). Designing Dexter-based and Cooperative Hypermedia Systems. In *Communications of the ACM, 37*(2).

[23] Grønbæk, K., Sloth, L. (1999). Ørbæk, P., Webvise: Browser and Proxy Support for Open Hypermedia Structuring Mechanisms on the WWW. In *Proceedings of The Eight International World Wide Web Conference (WWW8)*. Toronto, Canada, May 11-14, 1999.

[24] Hansen, K.M., Yndigegn, C, Grønbæk, K. (1999). Dynamic Use of Digital Library Material - Supporting Users with Typed Links in Open Hypermedia. In *Proceedings of The Third European Conference on Digital Libraries (ECDL'99)*.

[25] Madsen, O.L., Møller-Pedersen, B., Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley.

[26] Krasner, G.E., Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-8. In *Journal of Object-Oriented Programming, 1*(3).

[27] Meyer, B. (1998). *Object-Oriented Software Construction, Second Edition*. Prentice Hall.

[28] Microsoft (1999). Microsoft Visual C++ Integrated Development Environment. [Online]: `http://msdn.microsoft.com/vstudio/`.

[29] Nanard, J., Nanard, M. (1991). Using Structured Types to Incorporate Knowledge in Hypertext. In *Proceedings of Hypertext 91*.

[30] Pfaff, G.E. (1985). *User Interface Systems. Eurographics Seminars*. Springer-Verlag.

[31] Rumbuagh, J., Jacobson, I., Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Addison-Wesley.

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

[33] Tsichiritzis, D.C., Klug., A. (1978). *The Ansi/X3/SPARC DBMS Framework: Report of the Study Group on Dabase Management Systems.* Information Systems, 3.

[34] Waldén, J., Nerson, J.-M. (1994). *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice Hall.

[35] Wiil, U.K., Nürnberg, P.J. (1998). Collaboration in Open Hypermedia Environments. In *Proceedings of the Fourth Workshop on Open Hypermedia Systems (OHS 4)*.

In Proceedings of TOOLS Pacific'1999

Design and Evolution of Software Architecture in Practice

Michael Christensen, Christian Heide Damm, Klaus Marius Hansen,
Elmer Sandvad & Michael Thomsen
Department of Computer Science, University of Aarhus,
Aabogade 34, DK-8200 Aarhus C, Denmark.
{toby, damm, marius, ess, miksen} @ daimi.au.dk

Abstract

With special focus on software architectural issues, we report from the first two major phases of a software development project. Our experience suggests that explicit focus on software architecture in these phases was an important key to success. More specifically: Demands for stability, flexibility and proper work organisation in an initial prototyping phase of a project are facilitated by having an explicit architecture. However, the architecture should also allow for certain degrees of freedom for experimentation. Furthermore, in a following evolutionary development phase, architectural redesign is necessary and should be firmly based on experience gained from working within the prototype architecture. Finally, to get it right, the architecture needs to be prototyped, or iterated upon, throughout evolutionary development cycles. In this architectural prototyping process, we address the difficult issue of identifying and evolving functional components in the architecture and point to an architectural strategy - a set of architectures, their context and evolution - that was helpful in this respect.

1. Introduction

Iterative development processes that take an evolutionary and incremental approach to application development are becoming standard in most object-oriented development processes. However, the development of applications that use these iterative approaches, lead to many new problems. Where the traditional waterfall processes provides an (assumed) precise and definite description of what is to be built, there is no equivalent firm foundation on which to base the initial development when using iterative processes. Furthermore, in doing iterative development, as the development continues, new and/or changing requirements must be addressed.

In the last couple of years the authors of this experience paper have been involved in a large project, that used an iterative and evolutionary approach to development. The project involved a group of university researchers and a global container shipping company and spanned a 14-month development period. During this period, the university team initially went through a number of iterations on a prototype of a global customer service system. When the prototype was approved, the development continued in a number of development cycles in which the prototype was extended both horizontally, to include all the important areas of business, and vertically, to contain a more substantial functionality. The research group consisted of one ethnographer, one cooperative designer, and six OO-developers. The roles of the different team members may, somewhat simplistically, be described as follows: the ethnographer focused on current practice within customer service and related areas, the cooperative designer focused on the design of future practice and technological support together with users. The OO developers

developed the problem domain model and implemented the prototype. For a more thorough discussion of the process and its multi-perspective application development character see [7].

This paper focuses on the software engineering process, more specifically on software architecture. In our experience an explicit focus on the software architecture of the system that was being built, can be seen as one of the key explanations as to how we, technically, were able to accomplish the task set out in this project. We will try to reflect on our concrete experience and present two central lessons:

An *explicit architecture* is essential; even in initial prototyping cycles. It is also important in order to provide a well-defined structure in which, e.g., functionality, user-interface, and a problem domain model can be created and recreated when evolving a prototype in response to new requirements. It is important as it provides the necessary flexibility to experiment with alternative solutions in areas that are not well understood and it can also facilitate parallel work thereby allowing faster development.

It is rarely 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.

2. Software Architecture

No definition of software architecture is commonly agreed upon [1], [5], [11], [21]. However, it is commonly agreed that software architecture is concerned with components of the system and their interrelationships. We will, therefore, in our discussion of architecture, focus on the *significant software structures, and their components and relationships*. As several authors remind us, e.g. [15], [5], [1], a software architecture can be observed from several specific views or viewpoints, each revealing different aspects. Examples include the module view, the process view, and the conceptual view. The module view describes what the important modules and sub-modules are, and it is useful for e.g. assigning work and defining encapsulation. The process view describes the running processes and their relation, and it is useful for e.g. performance analysis, and analysis of multi-user aspects. The conceptual view describes the major components in the architecture as perceived by the developers. This view is useful for e.g. understanding the problem domain and the current system in relation to that.

Then, to describe the software architecture of the system discussed in this article a slight variation of the notation described in [1] will be used. Generally, solid lines denote control and processing, whereas dashed lines denote data. For our purposes, we have added the symbol "Emerging structure" to be able to describe the process of architectural creation and evolution.

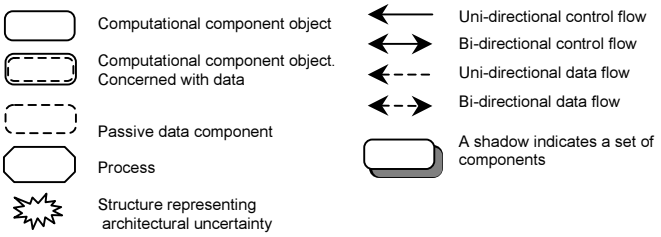


Figure 1. Architecture notation

2.1. Evolution and Architectural Refactoring

We make a clear distinction between the terms 'refactoring' and 'architectural refactoring'. Whereas refactoring is considered with semantic-preserving transformations of objects and classes [19], architectural refactoring is considered with function-preserving transformations of architectural structures. This means that a software system, after an architectural refactoring, will compute the same functions with respect to some problem domain. Pushing it a little, the

reasons for using refactoring are concerned with code-technical transformations, whereas architectural refactoring needs to be seen in a broader software engineering perspective: technical as well as social considerations must be taken into account in order to understand the reasons for doing this.

Evolution, as opposed to architectural refactoring, is concerned with change in functions. Evolution of functions presupposes both flexibility and stable structures. The flexibility provides the ability to change within the stable structure, while the stable structure at the same time helps in achieving important non-functional requirements.

By using specific examples of the evolution and architectural refactorings that the developed system has undergone so far, this experience paper aims at explaining how a focus on architectural issues may support an experimental and evolutionary development process, that, although experimental and evolutionary, still aims at producing a sound system as seen from the software engineering perspective. Moreover, a concrete architectural strategy to support this process is pointed to.

3. The Development Process

The development of the initial prototypes and later the concrete evolutionary development was done for a large, global shipping company. Our work on this development cannot be regarded as an isolated piece of work though: it was part of a larger project within the company that concerned the development of a uniform and globally shared way of providing container transportation. The entire project can be described in a number of streams as illustrated in Figure 2 below.

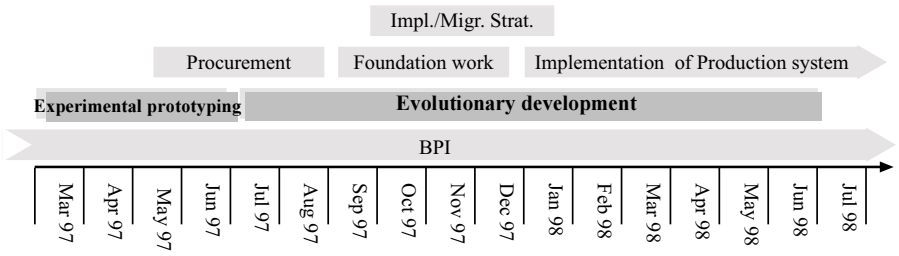


Figure 2. The streams of work within the overall project

Our work – referred to as the *Dragon Project* – is here seen as one stream (boldfaced), that consists of two major development phases. The shipping company, in cooperation with external consultants, carried out other streams of work. Examples of these include a global Business Process Improvement (BPI) project and various types of foundation work regarding network issues, migration issues, and overview of existing databases. These different streams interacted and, of course, influenced each other. That is, the overall process of the Dragon Project is something that should be seen as being related to and influenced by the other streams of work, with the common goal of developing a global customer service system for the company.

3.1. The Dragon Project Process

The process within our project can be divided into two separate phases: An experimental prototyping phase and an evolutionary development phase.

Experimental Prototyping. The first phase was primarily concerned with obtaining an understanding of the problem domain that the system should support. To this end, a number of software prototypes were constructed and evaluated in collaboration with the actual end-users.

The rationale, that ought to be well known, is that building computer system from paper descriptions alone, without any intermediate embodiments, is a problematic endeavour [13].

Concretely, and initially, in this project, a number of prototype versions were developed. These should illustrate and support the implementation of the global improvements suggested by the BPI stream. The concrete requirements resulting from the BPI stream were, however, quite vague. The whole first phase of the project was therefore very much occupied by narrowing down *what* a final system could be and by helping to determine the feasibility of the actual development.

To clarify and following [9], we characterise our approach as *experimental prototyping,* as our prototype versions were used to determine whether our ideas were adequate. This is only one characteristic of the initial phase, though. Other characteristics include: An *exploratory* style of prototyping (ibid.) to increase our limited understanding of the complex business of shipping; an *iterative* approach with short development cycles (14 days on average between reviews) to facilitate rapid development; a combination of *horizontal* and *vertical* prototyping (ibid.) to achieve both depth and width; and, finally, in general, usage of a wide variety of techniques from cooperative design [2], [12].

Evolutionary Development. The result after the first phase was that the company decided to embark on the implementation of the final system, their decision being based on the prototype. This lead to the initiation of the second phase of the Dragon Project. Compared to the experimental prototyping phase, the character of the work in this phase naturally changed focus towards the development of the production version of the final system. Apart from the Dragon Project, the company, as mentioned previously, initiated different types of foundation work.

In this setting, the role of the Dragon Project was twofold. On one hand, the prototype served as specification (in a broad sense, encompassing e.g. division into components, client-side architecture and problem domain model, i.e., not a *requirement specification*) for the production version, and on the other, it served as "the big picture" for demonstration/teaching/design purposes. This meant that, in the second phase, emphasis had switched more towards elaborating the components/concerns previously identified, with particular focus on the client-side of the coming system. We characterise this part of the process as the *evolutionary development* phase. Naturally, the functional requirements were not all fully elaborated after the experimental prototyping phase, so many of the same techniques and approaches were applied in this phase. Nevertheless, a much clearer understanding of what the constituent parts of the system should be and how they should interact had been obtained. This made way for a more steady, evolutionary character of the work, with longer development cycles (reviews about every two months) and room for necessary software architectural considerations.

Work within the Phases. Within each of the phases, our work was organised in a number of development cycles that had a duration of approximately 14 days in the first phase and approximately two months in the second phase. Figure 3 below gives a schematic overview.

As the figure illustrates, a number of concerns were addressed within each cycle (i.e., between two reviews). The left side of the dashed vertical line shows the first phase, in which focus was on producing functional, but not very elaborate, implementations of major concerns. In the second phase, shown to the right, our focus shifted towards evolving and elaborating these.

In each cycle and with each of these concerns, a mixture of analysis, design, and implementation was made depending on the current understanding and possible experiences from an earlier cycle. This always involved continuous workshops with business representatives. Furthermore, each concern would be addressed by several members of the development team in parallel and seen from different perspectives, and several concerns were often treated concurrently.

To "control" the development, each cycle ended with a review, during which the current work was presented to some of the future end-users and management from the company. At these reviews, possible problems with the current solutions would be brought forward and a plan for what was to be done in the next cycle would be made.

A much more in-depth discussion of the development process, and the lessons learnt from it, can be found in [7].

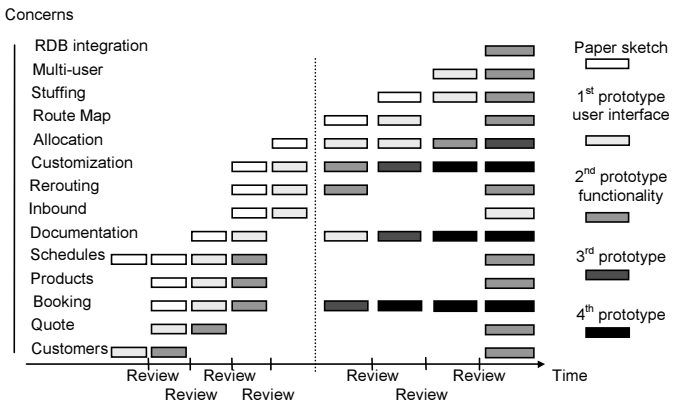


Figure 3. The evolution of central concerns

3.2. Tools and Code

The concrete results achieved is a number of prototype versions that has evolved into a rather large application consisting of over 300 files containing over 100,000 lines of code, and having well over 50 screens. The development platform was Windows NT, and the main software engineering tools used were: a CASE tool, a graphical user-interface builder, a code editor, a persistent store, a relational database (IBM DB2), and concurrent versioning system [8]. The first four tools are part of the Mjølner System [14], [6], [18] and the programming language used was BETA [16].

3.3. Related processes

In many ways, Barry Boehm's *Spiral Software Development Model* [3] resembles our process. The spiral model is both iterative and evolutionary. It is also based on the notion of development cycles, and these cycles also end with a review, in which plans for the next cycle are made. Furthermore, Boehm thoroughly describes how "risk management" controls the process. There is no explicit focus on software architecture in Boehm's article, but through his description of risk management, an indirect advice can be found: If the current architecture is considered a risk or an uncertainty as each cycle is planned, it should be given priority on the following cycle.

The *Unified Software Development Process* [10] describes a development process divided into a number of overall phases of which the first two, *Inception* and *Elaboration*, to some extent, are similar to our *Experimental Prototyping* and *Evolutionary Development* phases. The Inception phase is concerned with the definition of scope and should include a description of a *candidate architecture* and demonstrate the proposed system via prototypes. The Elaboration phase more thoroughly analyses the problem at hand and should result in an *executable architecture baseline*, a *software architecture description*, and an *80% complete domain analysis model.* The Unified Software Development Process is furthermore described as an *architecture-centric process*, We agree with this focus on architecture. Among others, we believe that our project provides empirical evidence on the importance of architectural focus and that a combination of rapid prototyping and architectural focus is feasible.

Rapid, iterative prototyping is in the industry often manifested in the form of RAD (Rapid Application Development). In RAD the development consists of a number of iterative development cycles, and there is also much focus on the use of prototypes. It seems though that RAD assigns no special role to architecture. In the standardised RAD described by DSDM [22] eight principles underlying RAD are described; none of these relate to software architecture.

4. Phase One: Experimental Prototyping

In section 3, it was described how the development was partitioned into two major phases. In this section, the role played by our software architecture in the first phase is described. More specifically, we look into the way, in which the concrete architecture provided a stable foundation for the experimental prototyping, while, at the same time, it allowed for the necessary room for experimentation. Furthermore, we describe the important criteria of having an efficient work organisation.

4.1. Initiating Development

The development was scheduled to begin with a two-week cycle. There was no concrete, specific goal of this first cycle, and we quickly became aware that our understanding of the business of shipping was quite limited, so two strands of work were initiated within the first cycle. One group worked on creating graphical user- interface screens, and another group worked on creating an initial problem domain model. The two groups worked in parallel, and there was little coordination between the groups. This was mainly a practical choice due to the strict time constraints, but it also proved to be a sensible one: In this way, many and diverse pieces of knowledge about the business domain were quickly gained.

4.2. Elements of an Architecture

At the end of the first cycle, two artefacts were delivered: A first design of the graphical user-interface, comprising of a few screens, and a problem domain model covering a subset of the problem domain (called 'quoting'). The user-interface design was illustrated by means of a horizontal prototype, i.e. a running application with virtually no functionality, except for functioning user-interface controls. The problem domain model was presented as a UML class diagram, and was, at this point in time, not a part of the running prototype. Nevertheless, it was already more than just a diagram: by using a CASE tool, code had been generated and updated incrementally, as elements were added to the model.

From an architectural/software-engineering point of view, we could, at this early point in time, – only two weeks into the development – identify the first two architectural elements: one component, consisting of the initial user-interface screens, and one component, consisting of the problem domain model (see figure 4).

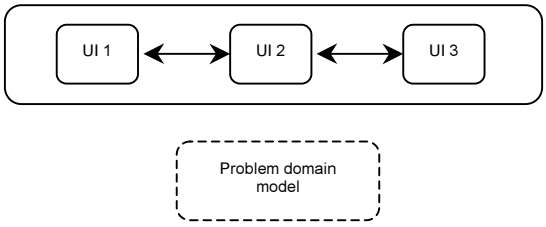


Figure 4. Elements of initial architecture

Retrospectively, we can ask ourselves why these two components were chosen. Much of the reason for this can be found in the developers' background. First of all, the development team had a common background in the Scandinavian approach to object-oriented development. In this approach, object-orientation is not seen as just providing a pleasing technical environment. It more importantly also provides a conceptual framework [16], [17] that underlies development. The creation of a problem domain model therefore becomes an essential part of development, since the problem domain model illustrates our understanding of the most important concepts in the problem domain.

Furthermore, part of the development team has a background in cooperative design [2], [12]. As the essence of cooperative design is the creative involvement of the user in the design of the computer system, the user-interface naturally becomes important. That lies in the simple fact that most users are not concerned with the code and other technicalities; they are simply

concerned with the way in which the system is operated, and how the system performs the tasks they expect.

4.3. Designing an Architecture

The review after the two weeks concluded that the development was on the right track. Together with suggestions for changes and next steps, it was decided that we should add functionality to what was covered so far and extend both the user-interface and the problem domain model to cover another business domain – booking.

As an implication of the decision to start adding functionality, it was considered how the existing parts (user-interface and problem domain model) could be combined. That is, the task was to design a software architecture that could contain both the existing elements and the functionality to be added. This architecture was outlined by two senior developers under consideration of several factors:

- 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.

Figure 5 illustrates the initial architecture. The figure uses the notation introduced in the architecture section and is drawn based on a "conceptual view": The figure illustrates the conceptual components – the major components as perceived by the developers – in the architecture, and the data and control flow between them.

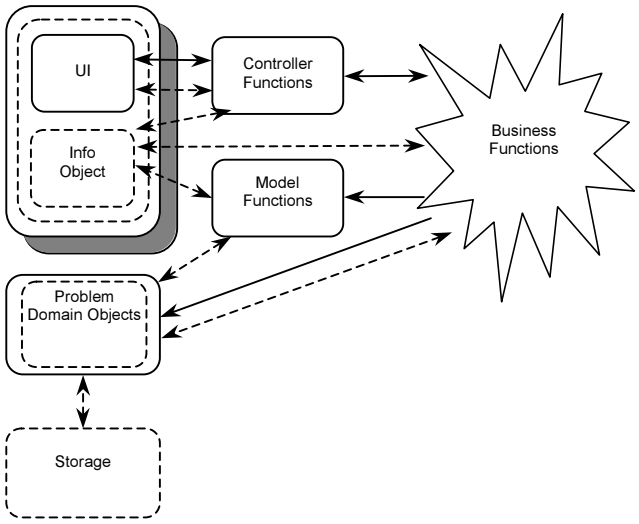


Figure 5. Initial architecture

The user-interface components (UI) hold the basic user-interface functionality. They are divided into several components according to divisions found in the problem domain. Between the Problem domain objects (or just Domain objects) and the user-interface components, the "Info objects" are found. These act as an interface between the Domain objects and the user-interface components. The Info objects are data structures that "mirror" the data in the user interface. Introducing this layer between the UI and the Domain objects de-couples the direct dependence between them. The Controller functions and Model functions components shown in the middle also act as interfaces; the Controller functions mediate between the user-interface components, the Info objects and the Business functions, and the Model functions mediate between the Domain objects and the Info objects. The business functions are the functions that are directly visible to the user; they implement functionality in the problem domain.

Ideally the Info objects layer is used as follows: when a Business function is activated via the UI (onEvent), the relevant content of the UI is mapped into the Info objects (getView). The corresponding Domain objects may need to be updated (updateModel) and the relevant parts of

the Info objects are used in the Business function. After execution of the Business function the Info objects are updated (presentModel) followed by an update of the UI (setView). An elaborated discussion of this architecture can be found in [23].

The left and middle part of the architecture diagram shows the well-defined parts of the architecture. Please recall that the user-interface and problem domain model units were constructed in the first cycle, and note that they remain elements in the new architecture (the user-interface component is transferred directly, and the classes of the problem domain model are instances in the 'problem domain objects' component).

The right part of the diagram, containing the business functions, is represented by the "emerging structure" type. The intention is to express architectural uncertainty. Since the business functions were unknown from the start, they could not be put in a well-defined box with well-defined interfaces to the other components. We needed a space in the architecture where we could experiment with the business functions in an unrestricted and flexible way. This architectural uncertainty will be further discussed in the following section on flexibility.

A Stable Structure for Evolution. 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. By defining a detailed structure within which the coding must be done, this complexity is reduced: E.g., defining the "communication protocol" between the user interface and the problem domain objects via info objects gives a clear and simple way of doing this communication.

In this way, the architecture provides an overview of the quickly growing prototype and constitutes a consensus about "how to do things". Furthermore, it ensures a "uniformness" as to how the functionality is implemented, and e.g. prevents developers pressured of time from "quick-and-dirty" solutions, like implementing the functionality directly in the code for the user interface.

Furthermore, the architecture serves as a vehicle for communication and explanation. This was for example 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.

It should, however, be noted that the well-defined architecture illustrated in the diagram was not achieved in full detail instantly after its definition. Although principal decisions about the overall structure were made immediately, some smaller decisions were not made, that allowed each developer room enough to find what he considered the best solution. As further understanding was achieved, these variations would be discussed and the developers would agree on a shared solution. In this way as the prototype evolved, even some overall structural decisions evolved.

Flexibility for Experimentation. Another criterion in the design of the architecture was flexibility, as it should allow evolutionary development of and experimentation with all parts of the prototype. This requirement is reflected in the large number of interfaces between the components in the architecture. These interfaces provided independence between e.g. the user-interface and the problem domain objects, that made it possible to experiment with each of these components without affecting the other components.

When it comes to the business functions, it was impossible to define an independent component that contained these, since the business functions were not well known. Furthermore, since the business functions are bridging between the user interface and the problem domain objects, they need to be able to access the user-interface (via the info objects) as well as the domain objects. And from the starting point and until very late in the prototyping process, it is not known exactly which part of the user-interface and which domain objects are involved.

Consequently, in order to allow for flexible experimentation with business functions, business functions need to have easy access to all components: user-interface, controller functions, info objects, model functions, and domain objects. The solution is an open architecture with

white-box components. Black-box components can not be introduced before the business functions have been found and developed. To indicate that the architecture has room for flexible experimentation with functionality, the business functions component is shown as an emerging structure in the diagram. As explained above, the business functions were intentionally not very structurally elaborated in the architecture in the prototyping phase. Instead the functions emerged in the boundary of the controller functions in the event handlers of the user interface. This structure was indeed very flexible but as will be described later, its use should be limited to the prototyping phase.

Another issue that enabled experimentation was the fact that we did not have to spend much time on handling storage in the prototyping phase. Making data persistent was an important aspect to be dealt with even in early design and implementation of the prototype. The persistence in the Mjølner system is orthogonal and transparent. This means that any object can be stored and that given a *persistent root* all reachable objects will be made persistent transparently. In this way, achieving persistence in the first versions of the prototype was straightforward. However, despite that, we did spend some time on storage: it was an explicit requirement in the first phase that the Dragon Project at some point should experiment with multi-user functionality, and abstractions preparing for this were made at the beginning of phase one. These abstrations, however, were not unproblematic. They were made prematurely, and in this way, constrained the initial architecture unnecessarily.

Efficient work organisation. As mentioned previously, another important criteria for designing the architecture was that is should facilitate an efficient work organisation. In our context, it should support roughly the following work activities in phase one:

- creation of the user-interfaces by the cooperative designer,
- creation of programming interfaces (info objects and controller functions) to the user-interface modules,
- programming of advanced graphical user-interface elements that could not be created directly using the graphical user-interface builder,
- programming of functionality (business functions and model functions),
- programming of the problem domain model,
- programming of components simulating legacy systems.

These activities had to be performed concurrently in order not to slow down the development. The architecture facilitated this. By making a strict and well-defined division of the user-interface, the business functions, and the problem domain model, with interfaces, the dependencies were reduced in a way such that problems were seldomly experienced. As an example, the info objects interface prevented most changes in the user-interface from having an effect on the domain objects and vice versa.

People had areas of responsibility according to major functional areas of the problem domain. One developer was, for instance, responsible for the customer-related functionality, another for the booking-related functionality and yet another for creating programming interfaces.

The user-interface was often used as a means of organising and/or coordinating activities between the cooperative designer and the OO developers. The cooperative designer mostly made the initial user-interface design in the graphical user-interface builder, and then in collaboration with the OO developers further elaborated it. Due to reverse engineering capabilities of the graphical user-interface builder, it was possible for the cooperative designer to make changes to the user-interface throughout the process, even after the generated code had been changed by the other developers.

Finally, the problem domain model served as a constant common frame of reference between members of the development group and to some extent also between developers and

members of the business. It forms a separate entity of its own right in the prototype and it was kept *purely* as a model of the problem domain isolated from implementation-specific classes. Also, the model evolved and since diagrams were used for discussion purposes, the reverse engineering capabilities of the CASE tool became important in the development process.

5. Phase Two: Evolutionary Development

We go beyond the discussion of architecture as a means of providing flexibility, stability, and proper work assignments in the prototyping phase and discuss why and how the prototype was used in the second phase of the project. In doing so, we will discuss the concrete architectural refactorings made, and the reasons for undertaking such restructurings.

5.1. The Transition to Evolutionary Development

After the first phase it was decided to extend the project for a year. This naturally changed the scope and focus of development. Although the prototype should, preferably, be used for continued experiments with business functionality, other areas of the system were to be explored and developed. These areas included multi-user functionality, database issues, and the general identification and creation of "black-box" components and their topology. Thus, the requirements of the architecture changed.

During the transition we evaluated whether the prototype, resulting from phase one, should be transferred into the evolutionary development phase. We had several options, including: To continue development of the prototype from the first phase; throw away the existing code, only transferring functional requirements; or totally discard the results from the first phase. Given that we were to use the same programming language and tools, that were used in the prototyping phase; that the code and architecture produced generally was of reasonable quality; and that the prototype had been well-received by the business, we decided to continue from the results of the first phase. This was not without problems though.

The Problems. The initial architectures enabled us to focus on experimentation with business functions. However, the prototype grew in size and complexity: From containing some over 150 files with around 50,000 lines of code after the experimental prototyping phase, the prototype evolved to contain over 300 files and around 100,000 lines of code at the end of the Dragon Project. This growth introduced a number of problems that forced us to refactor the prototype architecture. Two major classes of problems were:

The prototyping sessions and the reviews produced new requirements to the prototype that "demanded" changes to the architecture, e.g. in order to be able to reuse code better,

Work within the existing architecture had shown annoyances, e.g., unclear separation of responsibilities and inconvenient dependencies between different parts of the prototype that could be rectified with a new architecture.

An example of the first type of problem is that, in the initial architecture, it was difficult to reuse business functionality. Until late in the first phase of the project, the required functionality was usually local to the corresponding part of the prototype: e.g., only the booking part used the booking functionality. The problems became apparent, as a "compound" function was needed that could perform a number of existing functions on a bulk of business objects. This compound function did not bring about any changes to the problem domain model, but only required reuse of a large part of the existing functionality located in different units. However, the existing architecture did not provide the proper abstractions for the reuse (this actually meant that the compound function initially, due to constraints of time, was implemented using copy-n-paste reuse of the existing code).

An example of the second type of problem was that the turn-around time (edit/compile/run cycles) in the initial architecture increased as the prototype grew bigger due to unpractical dependencies between parts of the prototype: certain changes to the model, the controllers or the

business functions caused recompilation of large parts of the prototype. The main program with the controllers and the business functions became a bottleneck, and as the prototype grew, the recompilation time became a limiting factor in the rapid development process.

The most flexible part of the prototype – the emerging business function components – had thus become problematic at this point. However, the *identification of business functions* enabled us to make clearer abstractions on the problem domain. In this way, the flexibility, inherent in the initial architectures, actually became a forcing as well as an enabling factor in the change. In other words, the "white-box" components of the initial architecture allowed for a high flexibility in the development process, but caused problems as the system grew. Moreover, the identification of business functions enabled the construction of "black-box" components [20].

The exact timing of this architectural refactoring depended on a number of (social) factors: in our specific setting, the new scope made it *desirable* to have a new architecture, and the "spare time", in between the two phases, made it *practical* to do the restructuring at that point in time. Another very important fact was that it was now *possible* to make the actual changes based on a greater understanding of both the problem domain and the "solution domain". Of course, the economics played an important role too; it was expected that the restructuring would pay off in the long run.

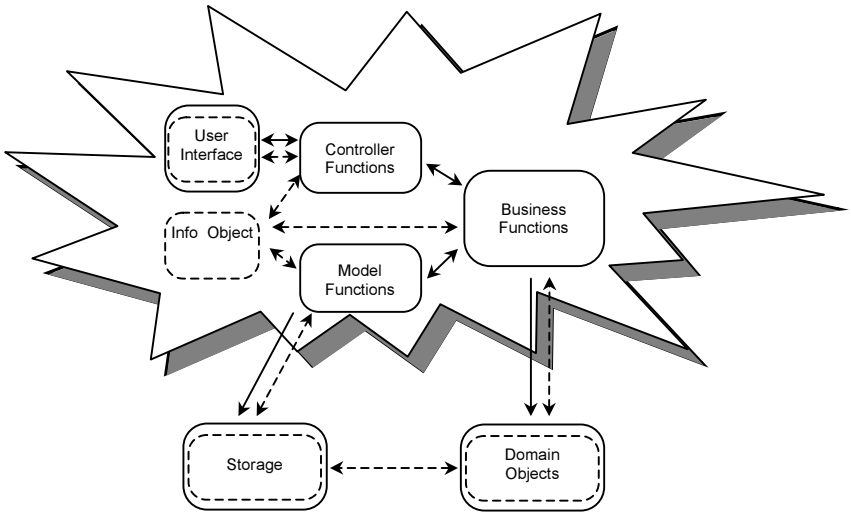


Figure 6. New architecture

The Problems Resolved. The experienced problems were solved with a new architecture, depicted in figure 6.The major change was made in the role of the business functions. From focussing almost entirely on the business functions, the architectural focus changed so that is now included the role of *problem domain area components*. More specifically, the goal was to develop black-box components, corresponding to the major problem domain areas, and to look into the division of work and responsibility between those components. In this way, it can be said that initial architectural work in the evolutionary development phase focused on the component topology.

Thus, the flexibility in the architecture at this point had to be mostly at the level of component boundaries. This is illustrated in figure 6 by now using the Emerging structure at the component level instead of at the Business functions level, because the architectural uncertainty now is on the component level.

The flexibility allowed for concrete experiments with component topologies during the first part of phase two, and led to a reasonable structuring of the software architecture.

As it turned out, this architectural refactoring did pay off in terms of faster turn-around time, higher levels of abstraction, and the ability to identify "black-box" components. During a trip to Asia, e.g., the power of this new architecture was experienced, as it was actually possible to add major components to the system "on the fly". In Singapore the system was modified ac-

cording to the discussions in prototyping sessions. To give an impression of the time frame: the visit to Singapore took three days. On the first day the system was presented and discussed in a number of sessions, with different representatives of the business and at different detail levels. Based on these sessions a number of changes to the prototype were decided. The next day, two developers made the changes in the prototype while the ethnographer and cooperative designer continued their work with the business representatives. On the third day the new version of the prototype was presented. This process was repeated in Malaysia. In this way two major component objects were added: A "query overview" component object providing overview of data related to major objects in the application domain and a "pending tray" component object supporting many collaborative work processes within customer service in the company.

However, this architectural refactoring was not entirely unproblematic. The trade-offs to be considered in such situations will be discussed in the next section.

5.2. Architectural Refactorings in Evolutionary Development

The success of the first architectural refactoring meant that *explicit refactoring phases* were incorporated in the project plan after the first three hectic months of the project. In this way each major review would also contain a review of architecture. Since the focus before a review was mostly on getting the system to work according to the planned changes, and the length of cycles was short (especially in the first phase of the project), we had to relax rules and conventions in order to meet the deadlines, e.g., by employing copy-n-paste reuse of code. This could then be rectified after a review; either by refactoring the architecture or refactoring code.

Please recall from section 2.1 that we distinguish between 'refactoring' and 'architectural refactoring'. In the Dragon Project refactorings could be done as a part of the ongoing development: One person, typically the person who made some shortcuts in the code during the hectic days before a review, did local refactorings. Architectural refactorings, that could be done independently in isolated parts of the prototype, were delegated to all the developers and they could do them when convenient.

The major architectural refactorings concerned the whole prototype, and hence development could not go on as usual. This meant that the role of deciding if, when, and how an architectural refactoring should be made had to be separated from ordinary development. Also, the holder(s) of the architectural role had some time set aside for doing the restructuring without doing ordinary development at the same time. In this way the major restructurings in reality meant temporary interruptions of development. The restructuring usually took between a couple of days and a week.

Not all changes were equally problematic. As the persistent store used provided for transparent, orthogonal persistence it was a relatively straightforward process to retarget the database component of the system. This meant that after that transition a transparent, heterogeneous storage mechanism was available. In this way programming to relational databases and persistent stores was possible at the same time. Also, initial identification of problem domain components and their boundaries were relatively easy after the first phase because of the increasing understanding of the major areas of business functionality during the first phase. Conceptually the candidates for problem domain area components had gradually been formed during the first phase. This architectural refactoring was, due to the use of a syntax directed code editor with semantic browsing and abstract presentation possible within short time. The abstract presentation provided overview and the fact that the editor was syntax directed made syntactically valid transformations of implementation code possible. Semantic "adjustments", however, had to be done using the semantic browsing facilities of the editor. Furthermore the transformation introduced a number of errors that called for renewed testing of the prototype. The architectural refactorings were thus somewhat problematic seen in the context of rapid

system development: A tool for reverse and forward engineering of the architecture or a more direct incorporation of architectural support in the tools used would be very useful.

Several refactorings took place during the evolutionary development phase meaning that the architecture of the system as delivered June 1998 can be depicted as in figure 7. Components and their boundaries have now stabilised in the sense that they represent useful abstractions of the problem and use domain. Notice that the Emergent structure symbols have been substituted by the well-defined architectural component symbols, because the architectural uncertainty has been resolved.

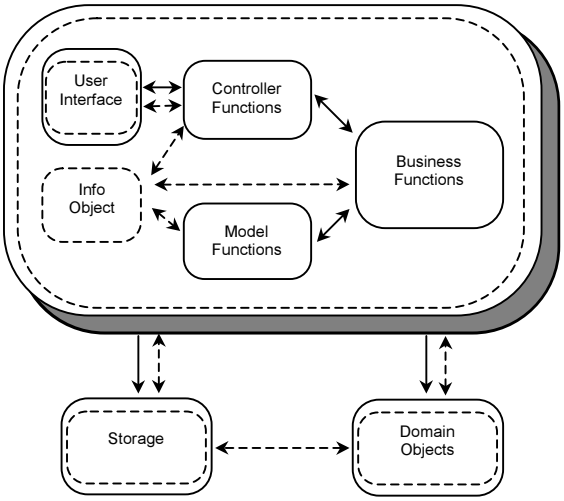


Figure 7. Architecture of June 1998

6. Conclusions

Aspects of the changes in architecture that have occurred during the first two major phases of the Dragon Project have been discussed. Figure 8 should convey two messages: *Firstly*, architectural focus in an experimental prototyping phase should very much be placed on the procurement of flexibility, stability and well-defined foundations for discovering functionality, adding new user-interface screens, etc. *Secondly*, evolutionary development may either discard or retain a successfully developed prototype. In either case, focus is much more placed on architectural prototyping: discovering and experimenting with a proper component topology, experimenting with legacy systems, distribution issues, etc. Still, both architectural and functional insights, gained in an experimental prototyping phase, provide rich and important input to the evolutionary development phase. Moreover, still being able to experiment with user functionality is important for several reasons: Although a proof of concept, a first prototype may not cover the whole scope of the final system in width; politically, organisations may need a continually evolving prototype to ensure user buy-in, etc. Continued experiments may also provide important insights, that also have architectural consequences.

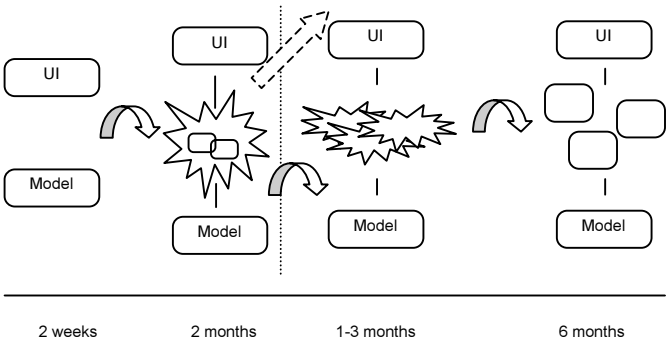


Figure 8. Evolution of architecture

Our project was characterised by dealing with complex human work practice, an unknown problem domain, and the need for rapid work, a context that we do not believe to be unique for our experience. Thus, we have suggested an *architectural strategy* - a set of architectures, their context and evolution - that supports experimental prototyping and evolutionary development in a context as ours. This strategy focuses on *stability* and *flexibility* in design of a computer system. Stability must e.g. provide for efficient parallel work in experimental prototyping and provide for efficient prototyping of a final architecture during evolutionary development. Flexibility must e.g. provide for the experiments with business functions in experimental prototyping and for experi-

ments with component topology during evolutionary development. The Emergent structure symbol indicates where the experiments take place.

An inherent dilemma of experimental prototyping and evolutionary development lies in the mixing of user involvement and software engineering. User involvement – and cooperative design in particular – provides for efficient analysis and design of functions pertaining to a problem domain. Object-orientation is then concerned with the implementation of emerging ideas of future work practices. The dilemma lies in allowing for flexibility and experiments with future work practice, while at the same time preserving the benefits of object-orientation and traditional software engineering. We address just that: The reconciliation of experimental development practice and software engineering practice goes through a strong focus on software architecture throughout the entire development process.

7. Acknowledgements

This work was supported by the Danish National Centre for IT-Research (CIT, `http://www.cit.dk`), research grant COT 74.4.

We would also like to thank Michael E. Caspersen, Henrik Bærbak Christensen, Jørgen Lindskov Knudsen, and Janne Christensen for valuable comments that improved this paper.

8. References

[1] Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice*. Addison Wesley Longman,1998.

[2] Bjerknes, G., Ehn, P., & Kyng, M. *Computers and Democracy: A Scandinavian Challenge*, Aldershot: Avebury, 1997.

[3] Boehm, Barry W. *A Spiral Model of Software Development and Enhancement*. COMPUTER. Vol. 21, No. 5, May 1988.

[4] Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition*. Addison-Wesley, 1995.

[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture. A System of Patterns*. Jon Wiley and Sons, 1996.

[6] Christensen, M. Sandvad, E. Integrated Tool Support for Design and Implementation. Nordic Workshop on Programming Environment Research (NWPER'96), Aalborg, Denmark.

[7] Christensen, M., Crabtree, A., Damm, C.H., Hansen, Klaus. M.H., Madsen, O.L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M. *The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping*. Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP '98*)*, Brussels. Springer Verlag.

[8] Gnu, *Concurrent Version System*. `ftp://archive.eu.net/gnu/`.

[9] Floyd, C. *A Systematic Look of Prototyping*. In R. Budde, K. Kuhlenkamp, L. Mathiassen, & H. Züllighoven (eds.), *Approaches to Prototyping*. Springer Verlag, 1984.

[10] Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, 1999.

[11] Garlan, D., Shaw, M. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[12] Greenbaum, J., & Kyng, M. *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, 1991.

[13] Grønbæk, K. *Prototyping and Active User Involvement in Systems Development: Towards a Cooperative Prototyping Approach*. Ph.D. Thesis, Aarhus University, Denmark, 1991.

[14] Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (Eds.) *Object-Oriented Environments. The Mjølner Approach.* Prentice Hall, 1993.

[15] Kruchten, P. *A 4+1 View Model of Architecture*. IEEE Software 12 (6) , 1995.

[16] Madsen, O.L., Møller-Pedersen, B., Nygaard, K. *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley, 1993.

[17] Madsen, O.L. Open Issues in Object-Oriented Programming – A Scandinavian Perspective. SOFTWARE – Practice and Experience, vol. 25, no. 54, 1995.

[18] `http://www.mjolner.com`

[19] Opdyke, W. *Refactoring Object-Oriented Frameworks*. Ph.D: Thesis, University of Illinois at Urbana-Champaign, 1992.

[20] Parnas, D.L. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, vol. 15, no. 12, December, 1972.

[21] `http://www.sei.cmu.edu/architecture/definitions.html`.

[22] Stapleton, J. *The Dynamic Systems Development Method Manual v2.0*. The DSDM Consortium, 1995, `http://www.dsdm.org`.

[23] Hansen, K.M., Thomsen, M. The 'Domain Model Concealer' and 'Application Moderator' Patterns: Addressing Architectural Uncertainty in Interactive Systems. Proceedings of the 31th Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia '99*)*

In Proceedings of CHI'2000

Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Aarhus N, Denmark
{damm, marius, miksen}@daimi.au.dk

ABSTRACT
Modeling is important in object-oriented software development. Although a number of Computer Aided Software Engineering (CASE) tools are available, and even though some are technically advanced, few developers use them. This paper describes our attempt to examine the requirements needed to provide tool support for the development process, and describes and evaluates a tool, Knight, which has been developed based on these requirements. The tool is based on a direct, whiteboard-like interaction achieved using gesture input on a large electronic whiteboard. So far the evaluations have been successful and the tool shows the potential of greatly enhancing current support for object-oriented modeling.

KEYWORDS
Gesture input, electronic whiteboards, cooperative design, object-oriented modeling, user study, CASE tools.

INTRODUCTION
Software developers use models to develop object-oriented software. In the early stages of a software development project, developers focus on understanding the part of the world that the computer system should support. Throughout the project, they represent their understanding in the form of models. The models are not only used in order to understand and discuss the world, but are also implemented in code and thus form an important part of the final software system.

A variety of Computer Aided Software Engineering (CASE) tools have been created to support the developers' work throughout the development process [12]. However, in practice these tools are supplemented with whiteboards, especially in creative phases of development.

The most appealing aspects of whiteboards are their ease of use and their flexibility. Whiteboards require no special skills, they do not hamper the creativity of the user, and they can be used for a variety of tasks. Their many advantages aside, for most development projects whiteboards are not enough. Capturing diagrams electronically with CASE tools facilitates code generation, documentation, and allows developers much more flexibility in editing and changing the diagrams. The conflicting advantages and disadvantages of whiteboards and CASE tools can lead to frustrating and time consuming switches between the two technologies. Our goal is to design a tool that offers the best of both worlds.

Paper Structure
The next section presents the motivation for our design. We then discuss two user studies from which we derive a set of design implications. We describe the Knight tool we developed based on these observations, and present an evaluation of the tool. Finally, we discuss directions for future research and draw our conclusions.

BACKGROUND
Use of whiteboards has been studied in various contexts including meeting rooms [9][15], classrooms [1], and personal offices [17]. Whiteboards are very simple to use and many activities are ideally suited for this simple interaction. Computational augmentation [24] can potentially solve problems with whiteboards such as lack of space and efficient editing facilities. We are concerned with the use of whiteboards in a specific work practice, cooperative object-oriented modeling, and the potential use of augmentation in that setting.

CASE tools seek to support software development techniques such as diagramming, code generation, and documentation. Nevertheless, adoption of CASE tools in organizations is slow and partial [5][8]. A main reason is that current CASE tools are targeted at technically-oriented methods rather than user-oriented processes. In particular, CASE tools are weak in supporting creativity, idea generation, and problem solving [7].

The *Tivoli* system [20][16] inspired us and was a starting point for our work. It is designed to support small, informal meetings on an electronic whiteboard. The similarity of user interaction to that on ordinary whiteboards is stressed. In order to be able to support specific meeting practices, Tivoli introduced domain objects that allow customizations of the tool to support, e.g., brainstorming sessions and decision-making meetings. We focus on the creation and

manipulation of a certain kind of domain objects and the integration of these into a computational environment.

OBSERVING DESIGN IN PRACTICE

We conducted two field studies of software developers using CASE tools and whiteboards in order to understand the current practice of object-oriented modeling. In both studies, we observed a group of developers with mixed competencies and then interviewed them. The developers used the Unified Modeling Language (UML [22]), which is a formal graphical notation containing several different diagram types. We concentrate on UML class diagrams, which are used to model central concepts and relationships found in the real world (or an imagined world).

Each study focused on three aspects of the design activity: *cooperation, action,* and *use. Cooperation* includes the communicative, coordinative, and collaborative aspects of design. *Action* and *use* are akin to the categories Bly et al. used in their observations of shared drawings [3]. *Action* involves the physical interaction of the designers with tools. *Use* involves the semantics of the result of actions.

User Study 1: Building a New System

Research setting. COT [29] is a technology transfer project between Danish universities and private companies. As part of COT, a university research group and developers from a private company cooperatively designed an object-oriented control system for a flow meter.

Participants. The developers from the private company had no previous knowledge of object-oriented development, whereas the university research group consisted of experienced object-oriented developers. The developers from the private company acted as domain experts in initial phases of development, while the university researchers were object-oriented software designers and, to some extent, mentors for the developers from the private company. During the two-week period in which the project was studied, the number of people attending design meetings varied. Typical sessions involved 2-4 developers from the private company and 2-4 university people.

Procedure. The sessions took place in meeting rooms equipped with multiple ordinary whiteboards, an overhead projector, and a computer. We observed three sessions in detail. In each of these an observer took notes.

Observation 1: Alternating Between Tools

Tom, Mike and Peter[1] *are about to discuss a new area of the problem domain. To brainstorm an initial design, Tom and Mike draw initial models on a whiteboard. Just before lunch they run out of whiteboard space and Peter captures the work so far using a digital camera. After lunch Tom and Mike continue on a freshly-wiped whiteboard, while Peter redraws the diagrams from before lunch in a CASE tool using the photos as a reference.*

Developers alternated between whiteboards and CASE tools. A typical work sequence involved sketching a model and then transferring it to a CASE tool, which could then use the formal model to generate code.

The next morning, Mike uses the CASE tool to generate diagrams from existing code. These illustrate details he and Tom discussed the previous day. Mike places printouts of the diagrams on the overhead projector (Figure 1) and Tom uses whiteboard markers to make amendments.

Work on an area of the problem domain involved several cycles of drawing and redrawing both on whiteboards and in CASE tools.

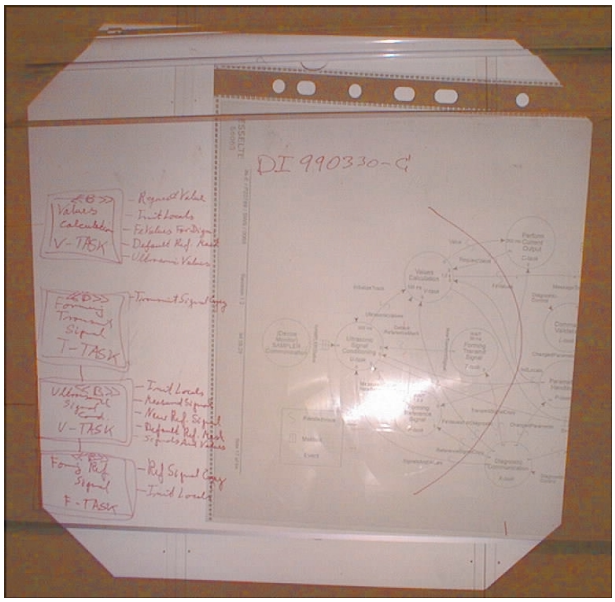


Figure 1. Using transparencies on a whiteboard

Observation 2: Working with a Formal Notation

Tom explains how a flow meter works conceptually. As he explains, he tries to model this using a UML diagram containing classes and relationships. He stops several times in order to ask Mike and Peter how to draw UML elements correctly. Moreover, to understand the semantics of the drawings, Mike and Peter often interrupt Tom.

The formal UML notation was hard to learn for the in-experienced developers. These syntactic problems caused a number of interruptions and breakdowns during modeling.

Peter and Tom are modeling how a number of objects interact with each other in the system. They want to show how these are synchronized. However, the UML is in-capable of describing these aspects. Mike suggests a new notational element for this. Peter and Tom pick this up and use it in subsequent design problems.

The semantics of the notation was extended in order to make it support the work process and to add expressive

[1] The names of participating designers have been changed throughout this paper.

power. In this way, the developers effectively extended the UML notation on the fly.

Observation 3: Combining Informal and Formal Drawings
Tom sketches the physical appearance of a flow meter on the whiteboard. He uses this drawing while explaining a diagram of the flow meter's electrical circuits. Following this, Mark models the interface to the circuits. He connects the elements to Tom's sketches.

The domain experts used illustrations, in connection with diagram elements, to explain important problem domain concepts. New ideas were often sketched informally, just before the introduction of UML notation.

Peter photographs the sketches of the circuits and flow meter before he continues to elaborate on Mark's diagram. Later, he realizes that he needs the sketches as a reminder. He consults the digital photo and redraws a part of the circuits before he continues modeling.

The informal drawings were temporary and were usually erased after the corresponding formal diagram was drawn. Central drawings were, however, redrawn on paper or photographed to make them persistent.

User Study 2: Restructuring an Existing System
Research settings. Mjølner Informatics [30] is a small company that makes compilers and other software development tools for the object-oriented language BETA [13]. A design meeting was held to design a new tool that integrated several separately-developed tools.

Participants. Six developers attended the meeting. They had varying experience in object-orientation and varying knowledge of the separate tools. Four of the developers had been previously responsible for a separate tool each. The last two developers had limited knowledge of the tools.

Procedure. We videotaped the ongoing discussion. In addition, we took notes with special emphasis on what was drawn on the blackboard.

Observation 1: Filtering of UML Drawings
John and Michael have each drawn a model of the tool they have developed. They now focus on how to integrate the two tools. Michael erases the parts of the tools that are irrelevant for the integration. He groups several associations into one in order to show a relationship even though he has erased the intermediate classes.

Often, the developers filtered information to handle large models (Figure 2). The developers idealized the model when it improved the understandability, reduced the interfaces of classes whenever full classes were too detailed, and kept transitive relations between classes to a minimum.

Observation 2: Editing Diagrams
Eric draws a model of a tool. To explain this, he adds details to the classes John and Michael have drawn. After a while this clutters the diagram. Eric erases some of the extended classes and redraws them further apart. For most of the "moved" classes he only redraws the box and name.

Participants frequently changed the diagrams. Such changes were time-consuming and annoyed the developers.

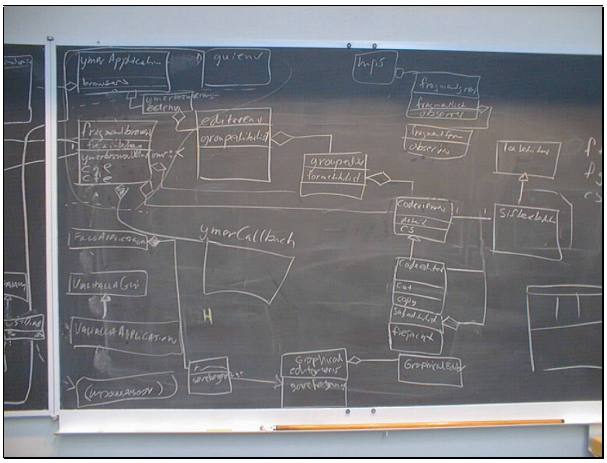


Figure 2. Blackboard snapshot

Observation 3: Drawing Informal and Incomplete elements
John and Eric are modeling a part of the integration. In order to show the lack of knowledge of this area, they only draw a partial diagram. Several times they draw relationships that are only connected to one class.

Approximately 25% of the meeting was spent on actual drawing on the blackboard. Of this, about 80% was spent drawing formal UML diagrams, and the remaining 20% on informal and incomplete drawings (or about 5% of the total meeting time). UML elements were drawn in incomplete variants, such as classes without names, incomplete inheritance trees, or associations connected to only one class.

Observation 4: Cooperation Between Developers
John starts to draw a model of a tool on the blackboard. The other developers sit at a table and ask questions. Following this, Eric starts to draw the tool that he has developed. Soon he discovers relations between the two tools and asks John to elaborate on his drawing. After John finishes, Eric continues on his drawing.

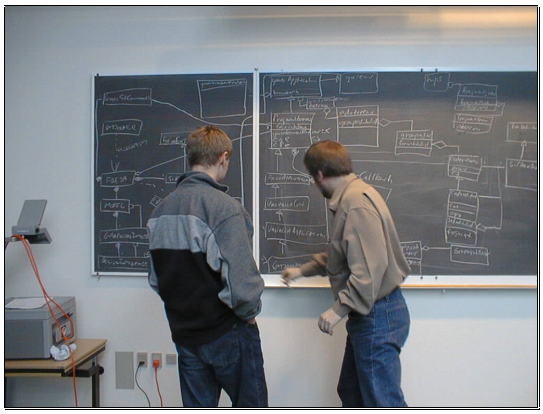


Figure 3. Turn-taking at the blackboard

The developers frequently cooperated by taking turns at the blackboard. Figure 3 shows two developers standing at the blackboard, taking turns adding, deleting or changing the model. Abrupt interruptions were rare and only one developer drew on the model at any time. However, other discussion often took place while another person was drawing.

Design Implications

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 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 led to the following design criteria for a tool to support object-oriented modeling:

- *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.

DESIGN OF THE KNIGHT TOOL

Based on the user studies, we have designed and implemented the *Knight* tool. The Knight tool uses an electronic whiteboard, currently a SMART Board [26], as its input medium. The SMART Board is a 72-inch touch-sensitive computer screen mounted in a cabinet to resemble a traditional whiteboard. Users can draw on the surface using a number of pens (or just using their fingers). In contrast to other electronic whiteboards, such as, e.g., the Xerox Liveboard [20], the SMART Board unfortunately only allows input from one pen at a time.

The prototype is implemented in Tcl/Tk [19] with the [incr Tcl] extension [14], runs on the Windows and Unix platforms and is available for download from the Knight homepage [25]. We kept the interface very simple (Figure 4). A large workspace, resembling an ordinary whiteboard, is the central part of the user interface. The interaction is based on pen-strokes made directly on the workspace.

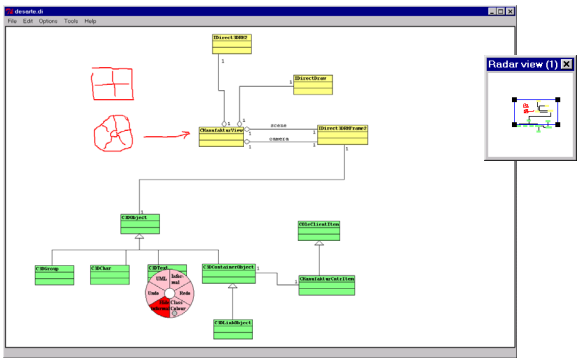


Figure 4. Knight user interface

Formality, Informality, and Directness

We wanted the tool to support a continuum of drawing formality, ranging from informal sketching elements over incomplete UML elements to formal UML elements. To allow this, the tool currently operates in one of two modes: Freehand mode or UML mode. We recognize that the use of modes is potentially problematic. However, our studies indicate that users already naturally operate in these two different modes, in different phases of the design. Freehand mode supports idea generation and UML mode supports design formulation. Two different background colors indicate the different modes.

In freehand mode, the pen strokes are not interpreted. Instead, they are simply transferred directly to the drawing surface. This allows the users to make arbitrary sketches and annotations just as on an ordinary whiteboard. Unlike on whiteboards, these can be moved around or hidden. Each freehand session creates a connected drawing element that can be manipulated as a single whole.

In UML mode, pen strokes are interpreted as gestural commands that create UML elements. If, e.g., a user draws a box, the tool will immediately interpret this as the gesture for a UML class and replace the pen stroke by a UML class symbol (Figure 5).

Figure 5. Recognition of the gesture for a UML class

The diagrams need not adhere to the UML semantics completely, in that incomplete diagram elements are allowed. Figure 6 shows how the user can input a relationship between two classes with only one of the two classes specified. The relationship can later be fully specified.

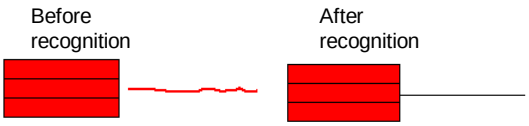


Figure 6. A relationship with only one class specified

The gestures for creating UML elements have been chosen so as to resemble what developers draw on ordinary whiteboards. This makes the gestures direct and easy to learn.

Another set of short directional gesture strokes chooses operations from a number of marking menus [10] illustrated in Figure 7.

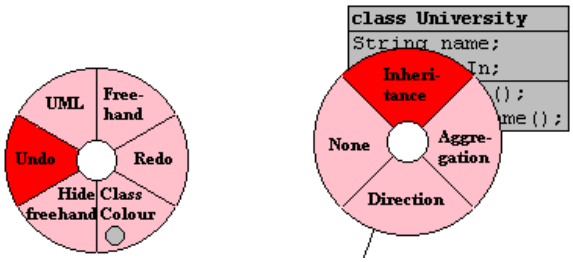


Figure 7. Context-dependent pie menus

For example, in order to undo or redo, the user may either make a short left or right stroke, or press and hold the pen and choose the corresponding field in a popup pie menu. The marking menus are also used to switch between UML and freehand mode. Marking menus support the interaction on a large surface well, because they are always ready at hand, unlike usual buttons and menus [20]. Apart from supporting a transition from initial to expert use, the marking menus also conveniently provide an alternative way of creating certain diagram elements (Figure 7 right). The marking menus are context-dependent. Depending on the immediate context in which a stroke or press was made, it will be determined whether it should be interpreted as a gesture command or as a marking menu shortcut. In the latter case a context-specific menu will be shown.

Use of Gestures

We use Rubine's algorithm [21] to recognize the gestures. This algorithm has the advantage of being relatively easy to train: To add a new gesture command, one simply draws a number of gesture examples. Potential problems with the algorithm, and gesture recognition in general, include that only a limited number of gestures can be recognized and that no feedback is given while gestures are drawn. To address these problems, we use *compound gestures* [11] and *eager recognition* [21], respectively.

Compound gestures combine gestures that are either close in time or space to a diagram element. For example, a user can change an association relationship (represented by an undecorated line) to a unidirectional association (represented by a line with an arrowhead) by drawing an arrowhead at the appropriate end. In this way, users can gradually build up a diagram, refining it step-by-step.

With eager recognition, the tool continuously tries to classify gestures as they are being drawn. Whenever the gesture can be classified with a high confidence, feedback is given to show that the gesture was recognized, and the rest of the gesture is used as parameters to the recognized gesture's command. For example, when a move gesture is recognized, the elements located at the starting point of the gesture will follow the pen while it is pressed down.

Support for Large Models

The workspace is potentially infinite, allowing users to draw very large models. It also supports zooming, as in zoomable interfaces [2]. In order to provide overview and context awareness, one or more floating "radar" windows can be opened (Figure 8; see also Figure 4 upper right).

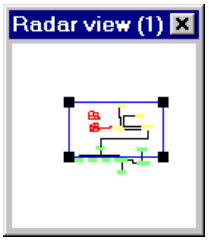


Figure 8. Radar windows provide context awareness

These radar windows 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. By opening more radar windows, multiple users can have convenient access to pan and zoom, even though the physical drawing space is large.

Filtering is in a preliminary stage. Currently, it is possible to suppress details of the formal UML model and toggle the visibility of informal elements.

Tool Integration

The Knight tool must be integrated with existing CASE tools, to facilitate code generation from the models. Although the Knight tool is currently only able to exchange data with the WithClass CASE tool [27], we are currently working on making it a plug-in front-end to a variety of tools. In this way it is possible to use the CASE tool capabilities to create or edit models outside a cooperative modeling situation.

EVALUATION OF THE KNIGHT TOOL

We evaluated the current design of the Knight tool in two sessions. Both sessions were actual design sessions in which Knight was the primary tool. The purpose was to evaluate the usability of the tool in a realistic work setting.

First, a facilitator introduced the Knight tool to the participating designers and taught them the basic use of the tool. During the evaluation, he also helped if the designers had problems and asked for help.

We videotaped the sessions and took notes. As in our user studies, we focused on three aspects of design: cooperation, action, and use. Following the design sessions, we conducted qualitative interviews.

Both sessions were encouraging. Each lasted approximately one and a half hours, and the participants were able to maintain focus on their job, rather than on the tool or the evaluation.

Next we discuss the results of the evaluations with respect to the design criteria identified and summarized in "Design Implications" above.

Evaluation 1: Designing a New System Using Knight

Research setting. The CPN2000 project [6][31] 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. We observed the meeting in which these three models were integrated into one model using the Knight tool.

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

Results

The resulting diagram is shown in Figure 9. This rather large model was constructed with few problems and mishaps.

Provision for a direct and fluid interaction. After the short introduction, the participants were able to use the tool for long periods without help: the tool had a low threshold for initial use.

The use of gestures was mostly unproblematic. However, some participants had trouble drawing certain gestures. This may be due in part to too little training, but the gesture set can also be improved. For example, people draw differently with respect to size, orientation, and speed, and the gesture examples used to train the recognizer must encompass such variations.

Support for cooperative work. The electronic whiteboard worked well as a center of cooperation. The only problem participants reported was that only one person could draw

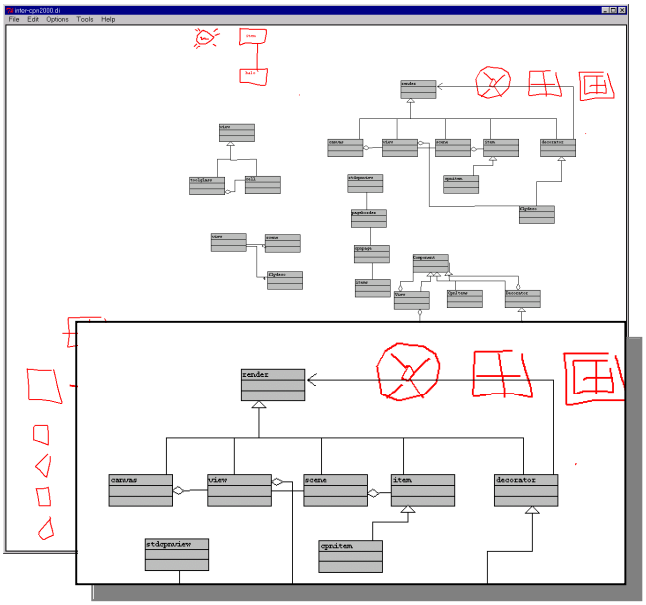


Figure 9. Diagram produced in the first session (with a blow-up of the upper right part)

at a time. Nevertheless, each developer was able to hold his or her own pen, and they all coordinated their actions when necessary.

Integration of formal, informal and incomplete elements. Freehand drawings were widely used and appreciated. The low resolution of the actual 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 quickly.

Support for large models. As the size of the model grew, the radar window was used to pan and zoom efficiently. However, the fact that the radar window was the only way to move around the workspace was problematic. In order to move a class from one corner of the large diagram to another, one participant had to move, pan, and then move again. This suggests the need for some other means of scrolling.

Evaluation 2: Restructuring a System Using Knight

Research setting. The DESARTE project [32] is concerned with designing an electronic support environment for architects. As part of this environment, a 3D replacement of the workstation desktop is 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 modeling.

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.

Provision for a direct and fluid interaction. After a short time, the participants were able to use the tool without many problems. 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.

One of the participants initially had many problems 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.

Support for cooperative work. The two participants had no trouble cooperating around the tool.

Integration of formal, informal, and incomplete elements. The participants 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. Incomplete UML elements were considered useful, but were not widely used.

Support for large models. In this evaluation, the focus was on restructuring an existing diagram of moderate size, and the radar window was mostly used for zooming.

DESIGN IMPLICATIONS & FUTURE WORK

The observations and subsequent interviews showed that the Knight tool is a valuable tool for modeling in practice. However, as the above results point out, improvements are needed.

A number of physical problems with the actual electronic whiteboard hindered cooperation. Only one person at a time could draw on the whiteboard and informal drawing was only slowly rendered. The latter problem may be handled in part by the Knight tool, whereas the former is intrinsic to the specific electronic whiteboard. However, the lack of support for synchronous drawings was not construed as a major problem in the two evaluations. Since design processes are becoming increasingly distributed, we are currently investigating distributed cooperative design using the Knight tool. Integration with a mediaspace [4] may provide a non-intrusive way of supporting distributed communication in relation to this.

Problems with gestures caused a number of breakdowns. Several possibilities exist for alleviating this. First, more appropriate feedback can be given when a gesture has been drawn. Second, personalized gestures may be necessary, e.g., in the form of a *personal pen*, as in Tivoli [20]. For each personal pen there could be a separate gesture set, a separate mode, separate colors, or other personal settings.

The integration of formal, informal, and incomplete elements is not complete. It is not, e.g., possible to connect formal and informal elements. A dynamic extension of the formal notation is a step towards this integration. The environment, with gesture recognition based on examples and an interpreted programming environment, makes such extensions technically feasible. *Flatland* [18] defines non-overlapping *segments* with different behaviors. Such segments may be used to group formal and informal elements separately. A notion of overlapping groups may be used to link these different segments.

Many of our observations of object-oriented modeling seem to be true for other types of formal modeling such as task modeling. A natural step would be to implement support for these as well, especially if combined with the idea of overlapping groups. This would facilitate combination of informal elements with formal elements, as well as handling types of formal elements together.

Filtering should also be considered in depth. Especially in evaluation 1, after the diagram had reached a certain size, navigation in the workspace became time-consuming. It should be possible to selectively hide parts of a model and give drawing elements temporality so elements may exist only for a certain period of time.

An important future area of research is the use of the Knight tool as a plug-in interface for different tools, which we are currently working on. This will involve longitudinal studies of the use of the Knight tool in development projects.

CONCLUSION

We have developed a tool for object-oriented development: Knight. The design of the tool is based on user studies of software developers creating object-oriented models. These show that important design criteria for a usable tool are (1) a direct and fluid interaction, (2) support for collaborative work, (3) an integration of both formal and informal drawing elements, (4) support for modeling in the large and (5) integration with existing development tools.

The Knight tool was designed to meet these criteria by using a large electronic whiteboard as input medium and by using an interaction style similar to that of traditional whiteboards. Input is done using gestures that resemble what is drawn on whiteboards. Both formal and more informal elements are supported and several developers can easily cooperate at the electronic whiteboard. Knight thus maintains the advantages of whiteboards and additionally adds features only possible in a computer based tool: Models can be easily modified, diagrams can be exported and imported to and from CASE tools, elements can be hidden and later restored, and a much larger workspace is provided.

ACKNOWLEDGEMENTS

We thank Michael Tyrsted who participates in the Knight project. We also thank Wendy Mackay for many discussions and for critique and help in writing this paper. Furthermore, we thank Michel Beaudouin-Lafon as well as the people from Danfoss Instruments, Mjølner Informatics, the DESARTE project, and the CPN/2000 project.

The Knight Project is carried out in the Centre for Object Technology that has been partially funded by the Danish National Centre for IT Research [28].

REFERENCES

1. Abowd, G., Atkeson, C., Feinstein, A., Hmelo, C., Kooper, R., Long, S., Sawhney, N., Tani, M.: Teaching and Learning as Multimedia: The Classroom 2000 Project. *Proceedings of Multimedia'96,* 1996, 187-198.
2. Bederson, B.B., Hollan, J.D. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. *Proceedings of UIST*, 1994, 17-26.
3. Bly, S.A, Minneman, S.L. Commune: A Shared Drawing Surface. *Proceedings of the Conference on Office Information Systems,* 1990, 184-192
4. Bly, S.A., Harrison, S.R., Irwin, S. Mediaspaces: Bringing people together in a video, audio and computing environment. *Communications of the ACM*, 36(1), January 1993.
5. Iivari, J. Why Are CASE Tools Not Used? In *Communications of the ACM*, 39 (10), 1996.
6. 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,* 1990, 119-133.
7. Jarzabek, S., and Huang, R. The Case for User-Centered CASE Tools. *Communications of the ACM*, 41 (8), 1998.
8. Kemerer, C.F. Now the learning curve affects CASE tool adoption. In *IEEE Software,* 9 (3), 1992.
9. Kraut, R., Fish, R., Root, R., Chalfonte, B. Informal Communication in Organizations: Form, Function and Technology. *Groupware and Computer-Supported Cooperative Work.* 1993, 287-314.
10. Kurtenbach, G. *The Design and Evaluation of Marking Menus*. Ph.D. Thesis, University of Toronto, 1993.
11. Landay, J.A., and Myers, B.A. Interactive Sketching for the Early Stages of User Interface Design. *Proceedings of CHI'95*, 45-50.
12. Lyytinen, K., Tahvanainen; V.-P. *Next Generation CASE Tools*. IOS Press, 1992.
13. Madsen, O.L., Møller-Pedersen, B., Nygaard, K. *Object-Oriented Programming in the BETA Programming Language.* ACM Press, Addison Wesley, 1993.
14. McLennan, M.J. [incr Tcl]: Object-Oriented Programming. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.
15. Moran, T.P., Chiu, P., Harrison, S., Kurtenbach, G., Minneman, S., van Melle, W. Evolutionary Engagement in an Ongoing Collaborative Work Process: A Case Study. *Proceedings of CSCW'96,* 150-159.
16. Moran, T.P., van Melle, W., and Chiu, P. Tailorable Domain Objects as Meeting Tools for an Electronic Whiteboard. *Proceedings of CSCW'98*, 295-304.
17. Mynatt, E.D. The Writing on the Wall. *Proceedings of INTERACT'99,* 1999, 196-204.
18. Mynatt, E.D., Igarashi, T., Edwards, W.K., and LaMarca, A. Flatland: New Dimensions in Office Whiteboards. *Proceedings of CHI'99*, 346-353.
19. Ousterhout, J.K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
20. Pedersen, E.R., McCall, K., Moran, T.P., and Halasz, F.G. Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. INTERCHI*'93*, 391-398.
21. Rubine, D. Specifying gestures by example. *Proceedings of SIGGRAPH'91*, 329-337.
22. Rumbaugh, J., Jacobson, I., Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
23. Russell, F. The case for CASE. *Software Engineering: A European Perspective.* Thayer, R., McGettrick, A. (Eds.) IEEE Computer Society Press, 1993, 531-547.
24. Wellner, P., Mackay, W., Gold, R.: Guest Editors' Introduction to the Special Issue on Computer-Augmented Environments: Back to the Real World. In *Communications of the ACM*, 36(7), 1993.

ONLINE REFERENCES

25. http://www.daimi.au.dk/~knight
26. http://www.smarttech.com
27. http://www.microgold.com
28. http://www.cit.dk
29. http://www.cit.dk/COT
30. http://www.mjolner.com
31. http://www.daimi.au.dk/CPnets/CPN2000
32. http://desarte.tuwien.ac.at/

In Proceedings of TOOLS Europe'2000

Tool Integration: Experiences and Issues in Using XMI and Component Technology

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted
Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark
Email: {damm, marius, miksen, tyrsted}@daimi.au.dk

Abstract

It is impossible to implement one tool that supports all activities in software development. Thus, it is important to focus on integration of different tools, ideally giving developers the possibility to freely combine individual tools. We discuss how tools can be integrated even in the context of conflicting data models, and provide an architecture for doing so, based on component technology and XML Metadata Interchange. As an example, we discuss the implementation of an electronic whiteboard tool, Knight, which adds support for creative and collaborative object-oriented modelling to existing Computer-Aided Software Engineering through integration using our proposed architecture.

1. Introduction

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 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 shifting between tools hard.

One tool will never be appropriate for *every* activity in software development. Thus, integration of tools is important. In this paper we report our experience with the integration of two commercially available CASE tools with a tool, *Knight*, that supports collaborative object-oriented modelling. The integration highlights concrete experiences with two current integration strategies: using a standardised interchange format in the form of XML Metadata Interchange (XMI, [24]) and using component technology in the form of Microsoft COM [19]. Our experiences with integrating CASE tools with different, although overlapping, data models also apply to other kinds of tools, and the discussions in this paper will thus cover general aspects of tool integration.

The rest of this paper is structured as follows: section 2 discusses tool integration in general and related work. Section 3 gives a brief overview of the Knight tool, i.e., its use, implementation, and software architecture. Section 4 describes and discusses our two experiments with tool integration and proposes an architecture for integration. Finally, section 5 concludes.

2. Tool 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 [7].

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

Table 1. Component collaboration taxonomy and typical applications

Our case study (described in section 3) focuses on the situation in which the processing components operate on data based on the UML metamodel. The cooperation may be performed in a number of ways, e.g.:

- as separate tools using a common interchange format (*asynchronous, shared*),
- as separate tools working on different aspects of the separate data (*asynchronous, separate*),
- as components collaborating through a well-defined interface to the common, logical core of data (*synchronous, shared*), or
- as interaction between separate tools via component technology (*synchronous, separate*).

Our case study concentrates on asynchronous integration with shared data and synchronous integration with separate data.

2.1. Related Work

Viewed as a development process, integration of components or applications has three important aspects: *designing architecture*, *creating components*, and *using components*. Designing architecture is concerned with choosing an appropriate architectural style, creating components is concerned with assigning functionality to components and relating this functionality to the environment of the components, and using components is concerned with finding, adapting, and assembling components.

In choosing an appropriate architectural style, several possibilities exist [17], including

- *File Level*. In this case the file is the common data abstraction. Using a common interchange format for CASE tools, as we do for asynchronous integration with shared data, builds on this architectural style. This is also a common way of integrating UNIX tools, typically using the Pipes and Filters [1] architectural pattern.
- *Single System*. Many programming environments build on a common interface and common data representation in a monolithic way. This integration is synchronous, uses shared data, and is in general hard to maintain and extend.

- *Program Database*. Using the Blackboard/Repository [1] architectural pattern, a loose coupling between integrated tools may be obtained. Given a common application programming interface to UML data, a synchronous integration with shared data could be implemented using this architectural style.
- *Message Facility*. Several integration efforts, such as the Field environment [17], builds on a Data Abstraction [21] architectural style with an emphasis on message passing. Our integration of separate tools via component technology essentially falls into this category.
- *Hierarchical*. One of the most ambitious efforts to create a reference architecture for tool integration [2] was built on hierarchical Layers [1] with layers for repository services, data-integration services, adding tools, process-management services, and user-interface services all built on a layer of message services. One of the major problems with this architecture was that it was complex and constraining: it fixed the integration of tools instead of allowing for open-ended lightweight integration. Rather than developing a fixed reference architecture for integration, we concur with Tichelaar et. al [22] in that emphasis should rather be put on how to coordinate independent components in flexible ways.

Components need to be designed with adaptation and integration in mind. This process is not well understood. However, guidelines for creating coordination components [22] and tailorable, reusable frameworks [6] may be useful in this.

Using components consists of four phases: *component qualification, component adaptation, component assembly,* and *system evolution* [10]. *Component qualification* tries do determine fitness for use of components using discovery and evaluation techniques. *Component adaptation* is concerned with modifying components for new contexts. The actual adaptation technique is determined by the nature of components (white box, grey box, or black box components), and wrapping, bridging, and mediating are well-known techniques for this [9]. *Component assembly* presupposes a well-defined software architecture in which components can be integrated. *System evolution* has the component as unit of evolution meaning that replacement of components may be problematic. Our integration efforts use a mixture of such techniques for adaptation and discuss experiences with the process of integration. Furthermore, we propose an architecture for synchronous integration based on separate data.

3. The Knight Tool

As an example of tool integration we use the Knight tool. The Knight tool extends existing CASE tools by providing an alternative user interface, which has support for creative, flexible, and collaborative modelling. The tool is implemented in Itcl [14], which is an object-oriented extension of Tcl/Tk [16], and it uses Microsoft COM [19] for tool integration. More information can be found at `http://www.daimi.au.dk/~knight`.

3.1. Use of Knight

Knight uses a touch-sensitive electronic whiteboard (currently a SMART Board – `http://www.smarttech.com`) as input medium (Figure 1a),. Since a major design goal of the Knight tool is to make the interaction with the tool similar to that of an ordinary whiteboard, the user interface is very simple: it is a white surface (Figure 1b), on which users can draw UML diagrams with dry pens.

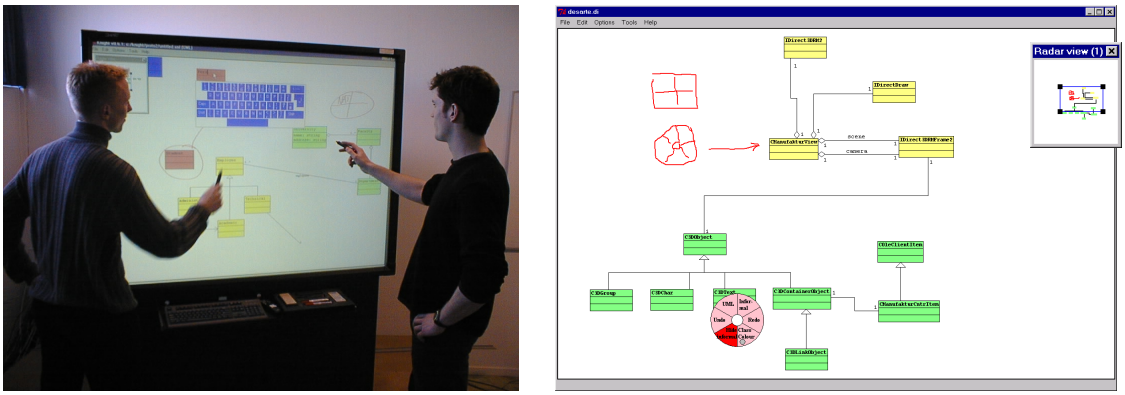


Figure 1. a) Use of Knight on an electronic whiteboard, b) Knight user interface with radar window

The Knight tool operates in two modes: an informal freehand mode and a formal UML mode. In freehand mode, the user may add arbitrary annotations to the diagram. In UML mode, the strokes of the user are interpreted as UML elements. As an example, to create a new class, a user can draw a rectangle with a pen on the workspace drawing surface, which the tool will then interpret as a class (Figure 2). This results in an interaction that is direct and intuitive [3].

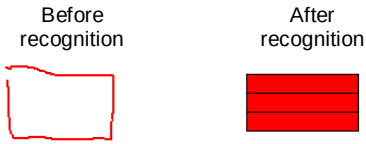


Figure 2. Recognition of the gesture for a class

The same type of interaction is used when the user wants to create other types of UML model elements. If the user wants an association between two classes, the user just draws a line between the classes; if the user wants to change the association to an aggregation, the user draws a diamond at one end of the association. Moreover, there are gestures for common operations such as deletion and movement of elements. To enable easy access to less common operations, a context-dependent pie menu [13] is provided (Figure 3a). The user may either press the pen against the drawing surface for a short while in order for the menu to pop up, or make a short stroke in the direction of the desired command.

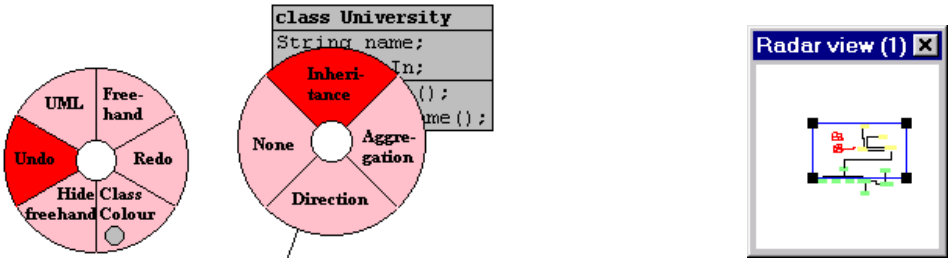


Figure 3. a) Context-dependent pie menus, b) Radar window

A small radar window (Figure 1b, Figure 3b) is used to provide context awareness and a potentially infinite workspace. To pan, the user drags the rectangle in the window. To zoom, the user uses the black handles to resize the rectangle.

3.2. Software Architecture

The software architecture of Knight follows the Repository architectural pattern [21], the Repository in Knight being a set of UML diagrams. The architecture is shown in Figure 4.

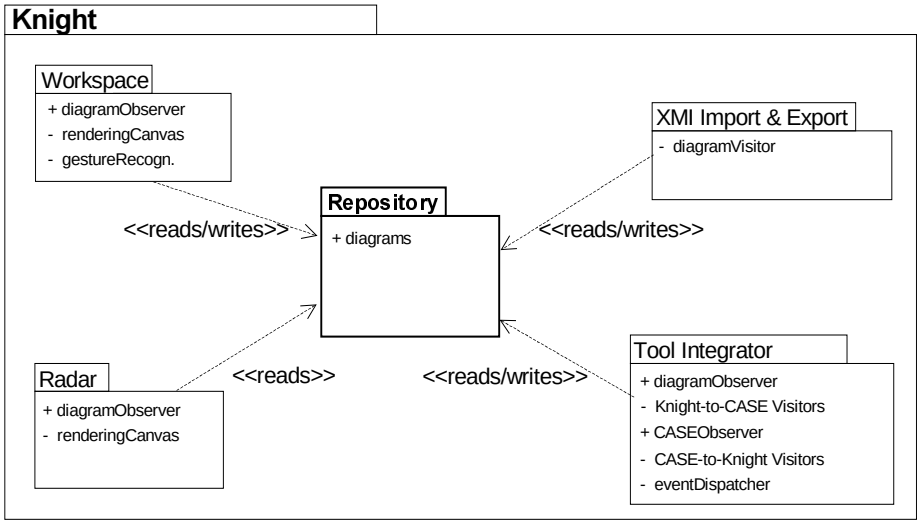


Figure 4. Logical view of the software architecture of Knight

A class diagram in Knight is basically a subset of the UML metamodel for class diagrams, extended with layout information (position, size etc.). The model is shown in Figure 5.

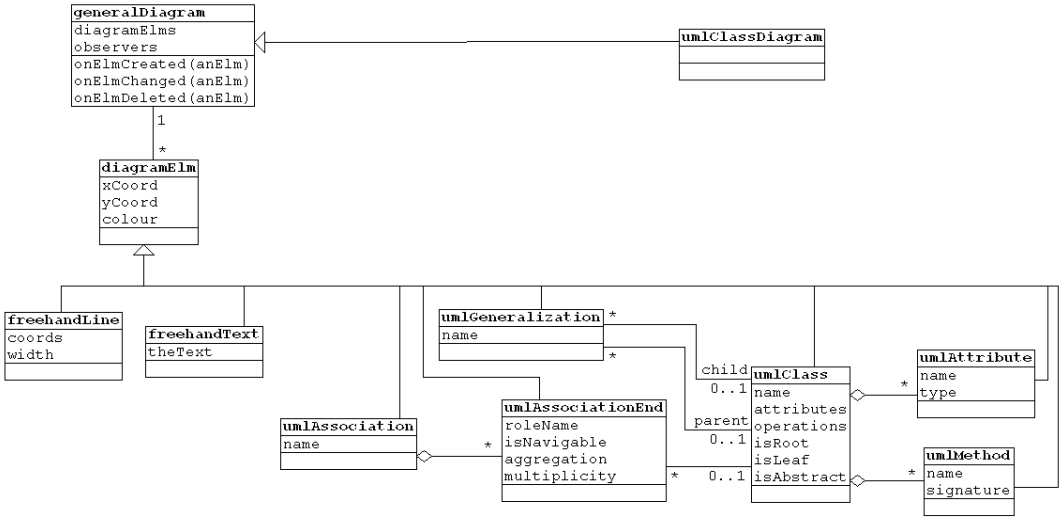


Figure 5. Simplified object model

It is possible to attach an Observer [9] to a diagram, which will be notified when elements have been created, changed, or deleted. Observers are used for many purposes in Knight. For example, the workspace uses an Observer in its implementation: when a diagram element is created, the workspace creates a corresponding visual representation of the new diagram element, etc. The small radar window is also observing a diagram. Observers are used for various other things, such as debugging and integrating with CASE tools.

In addition to observing, the workspace also writes to the Repository, namely when the user interacts with the workspace. Writing in the Repository is implemented by using a combination of the Composite and the Command patterns [9], providing fine-grained multi-level undo/redo functionality.

4. Tool Integration

Having introduced the subject of our case study, we now discuss the details of the integration. As mentioned, the discussion will focus on asynchronous integration with shared data and synchronous integration with separate data. In order to motivate our choice of these two strategies, we introduce three typical use scenarios that will be referred to in the discussions. In all three scenarios, Mike and Sarah are part of a development team that develops an administrative system for a university. Mike is a domain expert, and Sarah is an object-oriented developer. They use the Knight tool for collaborative modelling activities, and Sarah uses a traditional CASE tool for refinement and detailed design.

Use scenario 1: creating a new diagram in Knight. Mike and Sarah are about to model the payroll of the administrative system. To do this they gather other relevant domain experts for a modelling session. At the beginning of the session, Mike creates a blank diagram in Knight. During the session, the topics of the discussion are recorded using a combination of UML class diagram elements and informal sketches on the electronic whiteboard. After the meeting, Sarah transfers the diagram to the CASE tool, in which she elaborates the diagram while making use of the possibilities the CASE tool provides.

Use scenario 2: comments on a model created in Knight. Mike and Sarah have had several modelling sessions during the preceding weeks, but they have a tricky problem they cannot solve. Sarah consults a colleague, Peter, in the company she works for. Peter discusses the model at the company site with a number of modelling experts, and they analyse the model and make a number of changes to it before sending it back to Sarah.

Use scenario 3: working on an aspect of a diagram in Knight. The administrative system has now reached a state in which it is running as a prototype communicating with a central, relational university database. Sarah has used the CASE tool to generate code for accessing the database. A conceptual problem discovered during prototyping leads to a remodelling session. Sarah creates a live link to the Knight tool, and Mike and she then discuss the conceptual problem and try to fix it using Knight. As they make changes, Sarah now and then checks the effects of the changes on the database access code in the CASE tool.

4.1. Asynchronous Integration with Shared Data

Use scenario 1 describes a situation in which there is a need for transferring a model from one tool to another. An interchange format supported by both tools is an effective and simple way of achieving this. The interchange data file represents a shared artifact that the two tools can read asynchronously, or sequentially, in order to collaborate.

Use scenario 2 also describes a situation in which an interchange format is useful, but in contrast to use scenario 1, it also illustrates a situation in which synchronous integration is problematic. Sarah needs to send the model to her colleague Peter, and as Peter does not have the Knight tool, in which the model was created, at the company site, the only option is to exchange the model asynchronously.

XMI. To implement asynchronous integration using shared data, we have implemented support for XML Metadata Interchange (XMI, [24]) in Knight. 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 6. Description of a part of a bank

Consider the simple diagram in Figure 6, 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, [15]). 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.

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[1]

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 7 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.

```
<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 7. Part of the bank model encoded in XML conforming to the UML DTD

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

To specify a grammar, the XMI standard uses XML (eXtensible Markup Language, [8]) 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 (`http://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 (`http://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 [12], there is currently no standardised way of encoding presentational information in a UML XMI file.

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 practise 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, has 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 ([18], `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 [25] 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.

4.2. Synchronous Integration with Separate Data

Use scenario 1 and 2 could be well supported by the XMI integration. Use scenario 3 above justifies the need for a synchronous form of integration: different tools may work on different aspects of the same data, and quick alternation between the tools may be needed.

Synchronous integration is typically component based. The most widespread component technologies are CORBA [**Error! Reference source not found.**] and Microsoft COM [19]. The choice of component technology is not essential for our purpose, but since most CASE tools today only support COM, we chose COM as well. Our implementation of synchronous integration with separate data is thus based on runtime COM connections between the Knight tool and the CASE tools. In this set-up, there are two phases to be considered: the initial synchronisation, i.e., a batch-transfer of data, and, subsequently, a continuous incremental synchronisation between the tools.

Initial synchronisation. If a user wants to work on an existing diagram made in a CASE tool, the initial synchronisation includes transferring the diagram to the Knight tool. It may also be the case that both tools contain diagrams that should be used in a modelling session. In this case the synchronisation is a merge of the two diagrams.

One Visitor for each direction implements the initial synchronisation. The Visitor that traverses the Knight diagram is a trivial application of the design pattern, whereas the Visitor that traverses the object graph of the CASE tool needs to be constructed externally to the CASE tool.

First, the ‘Knight-to-CASE Create Visitor’ traverses the Knight diagram. During the traversal, it builds a similar diagram in the CASE tool: when it visits a class in Knight, it creates a copy of the class in the CASE tool, etc. Second, the ‘CASE-to-Knight Create Visitor’ traverses the CASE tool diagram and copies it to Knight (this time ignoring the elements just

copied *from* Knight). A logical view of the architecture for the initial synchronisation is shown in Figure 8.

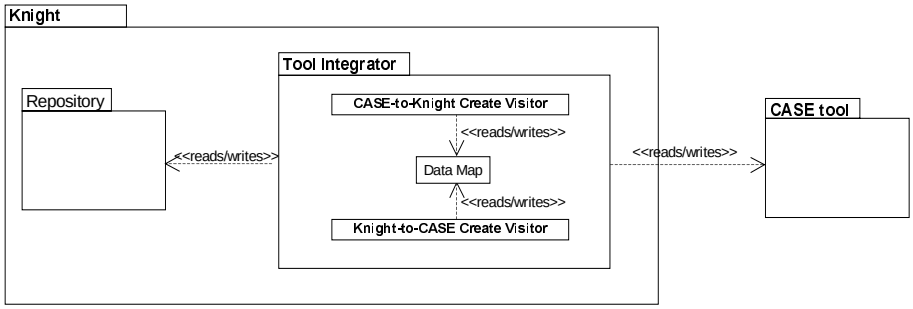


Figure 8. Initial synchronisation

While synchronising the two tools, the Create Visitors build a mapping between the data in the tools. The mapping is used in the incremental synchronisation, allowing, e.g., a change to a class in the Knight tool to be propagated to the corresponding class in the CASE tool. The Create Visitors of the initial synchronisation may also be used to support scenario 1 above: if the following continuous, incremental synchronisation is not invoked they in effect implement a traditional import and export of diagrams.

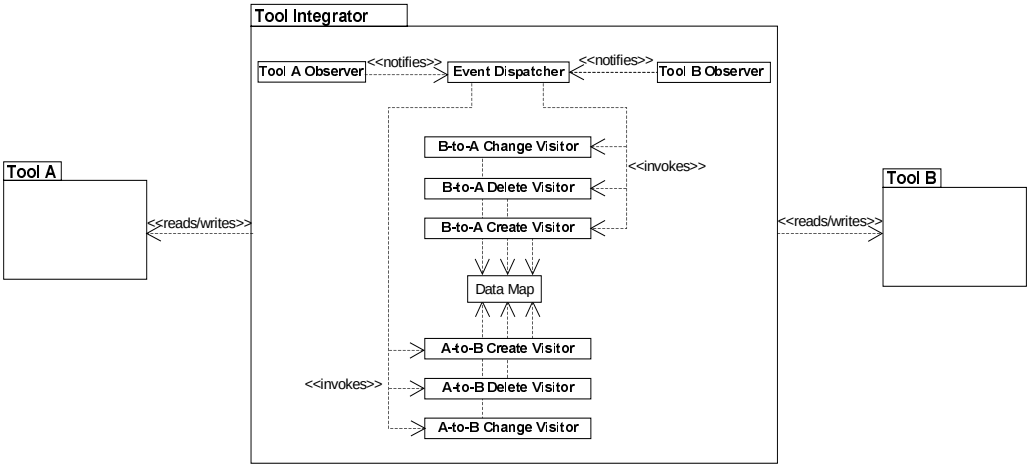


Figure 9. Architecture for synchronous integration with separate data

Continuous incremental synchronisation. In order to keep the diagrams in the two tools synchronised after the initial synchronisation, changes in one diagram should propagate to the other diagram. In our implementation, two integration-specific Observers observe the diagrams. The Observer of the CASE tool relies on COM events. The overall architecture for the continuous incremental synchronisation is depicted in Figure 9. Note that the architecture has been generalised and shows the integration of two generic tools A and B. The architecture can be used for integrating tools that have "compatible" data models, such as the UML metamodel in our case.

If, e.g., a new class is created in tool A, the A Observer will get notified, and it will then make the 'A-to-B Create Visitor' visit the new class. The 'A-to-B Create Visitor' will, as always, create a copy of the class in tool B. In a similar way, changes to, or deletion of, existing elements in tool A will be propagated to tool B, implemented by two other Visitors.

Updates in tool B are propagated to tool A in a similar way. The Event Dispatcher mediates the notifications between the tools based on subscriptions.

In Figure 9, the 'Tool Integrator' is shown as a separate component, which is not *part of* any of the tools (in contrast to Figure 8). The integrator component is symmetrical and thus requires the same handles on both tools. The reason that it is a part of the Knight tool in our implementation is that Knight currently has no COM interface.

Component Technology Experiences. The architecture for integrating Knight with CASE tools is simple and effective. It has proved to be adequate for the two CASE tools we have experimented with, namely Rational Rose and WithClass (`http://www.microgold.com`), and we believe it will work for other tools as well. When integrating existing tools, there will, however, often be specific issues to consider as well as problems to overcome. This was also the case for Rational Rose and WithClass, and next we will present our experiences with these problems.

In Knight, a diagram may contain freehand drawings, which are not supported in Rational Rose and WithClass. To ensure that these drawings are not lost, we store them in the CASE tool in a special note element, which we hide from the user. In general, the solution to this problem is to find *somewhere* to store arbitrary information – and it may vary for different tools.

There are also examples of Rational Rose and WithClass diagrams containing more information than Knight diagrams, but because the Knight tool operates directly on the Rational Rose and WithClass diagrams, we chose not to store the extra information in the Knight diagrams.

There are situations, however, in which extra information in the Rational Rose or WithClass diagrams should be handled in a special way. When performing, e.g., a 'delete' followed by an 'undo', the state of the diagrams in both Knight and the CASE tool should be exactly the same as before the deletion. If the diagram in the CASE tool contains extra information, then this information must also be restored in some way. If the CASE tool supports multi-level undo, as does Knight, this is trivial. If not, a reasonable alternative is to serialise objects in CASE tools when a delete operation is performed, using their COM interface, and to de-serialise objects when an undo operation is performed. Failing this, other actions may be taken, such as hiding and showing diagram elements, and removing these elements completely when disconnecting. In Knight we solve the problem by a combination of serialising and hiding: presentational data is serialised and logical data is hidden. Notice that the same problems may occur in other situations, e.g., if elements are copied.

4.3. Discussion

Cooperative work has three aspects: *coordination, collaboration*, and *communication*. *Coordination* is realised through rules, delegation, sign-offs etc. *Collaboration* realises cooperative creation of artifacts. *Communication* is the organisational distribution of information. These three aspects of cooperative work are also relevant in tool cooperation.

Asynchronous integration. In connection to asynchronous integration with shared data, *coordination* is partly on the discretion of the developers using the integrated tools. The XMI file *is* the common artifact worked on by both CASE tools and Knight that is being *collaborated* on. The UML DTD defines the *communication* syntax and the semantics is partly

decided by the semantics of the UML metamodel. The communication is open-ended, and tools may add arbitrary extensions using the XMI extension tag.

A standard interchange format is important in enabling different tools to collaborate on the same data. With XMI, a UML model can be stored in a standard way. On the other hand, many tools have needs that are not covered by the standard. These tools will have to add extensions to the standard format, thereby making full integration with similar tools hard. The evolution of HTML and browsers nicely illustrates the problem: different Web browsers and editors have created idiosyncratic mark-up, some of which has later become standard (such as frames) and some of which have remained implemented for one tool only (such as positioning mark-up). Similarly, the UML metamodel cannot describe all types of relationships between entities in class diagrams. Descriptions of method call and attribute access, e.g., are not supported. Several tools, including some program visualisation and refactoring tools, need this kind of information. Thus, it is problematic to use UML as basis for tool integration in general [5]. Just as the UML metamodel currently is being extended to cover diagram specification, the inclusion of code information may be considered. It is obvious, though, that the agreement on a standard will always lag behind the needs for cooperation between tools.

Synchronous integration. In connection to our implementation of synchronous integration with separate data, *coordination* is mainly handled by the Event Dispatcher component, which, among other things, ensures that an update in one tool will not lead to an infinite number of update signals between the integrated tools. The CASE tool and Knight *collaborate* on a common artifact, namely the model that the developers are designing. The used component technology and the interfaces of the tools decide the *communication* protocol; the Observers influence the communication pattern of the integration.

There is currently no standard programming interface that has been adopted by CASE tools. Rational Rose and WithClass have completely different COM interfaces. We argued above, through the use scenarios, that there is a need for both asynchronous and synchronous cooperation between tools. Hence, there is also a need for a standardised programming interface for operating on the data that has already been standardised in the UML. Just as a standardised data interchange format has been created from the UML metamodel, we believe that a standardised programming interface for operating on data conforming to this metamodel should be created. In the case of the UML Metamodel, there should be methods for, e.g., creating and deleting classes, and adding and deleting attributes. Of course, there should also be methods for attaching arbitrary information these elements, just like it is possible to add arbitrary data in the extension elements of the XMI format. A proposal for a standard interface is provided as part of the UML 1.3 specification [23], in the form of a CORBA IDL description. However, we are aware of no tools that implement this specification and thus have no experiences as to whether it is actually usable. While having a standard programming interface for operating on the data itself is important, it is not sufficient. There is still a need for tool-specific functionality, e.g., opening and closing another tool or scrolling a diagram. The situation is thus the same as for common interchange formats: only part of the interaction will be standardised.

Other integration types. In this paper, we have mainly considered the two categories "asynchronous integration with shared data" and "synchronous integration with separate data" of Table 1. One of the other categories, “synchronous integration with shared data”, covers components working together synchronously on the same data. This model is also very common in existing tools, e.g., in integrated development environments with editors, compilers, debuggers etc. working on the same data. It would be easy to modify the Knight

tool to work on, e.g., a Rational Rose data structure, instead of having its own data, which, conceptually, is just a copy of the Rational Rose data. However, by having its own data, Knight can run as a stand-alone tool without Rational Rose. This situation is analogous to exchanging data between Knight and Rational Rose (section 4.1): having to run both tools synchronously is impractical (sometimes even impossible) compared to exchanging the data in a file. Also, if Knight uses Rational Rose's data format, it cannot run with WithClass or any other tool. Of course, having standard programming interfaces would remedy the problem.

The final category in Table 1 is "asynchronous integration with separate data". We have not considered this category in detail in our integration work, but just to complete the table we give an example: a diagram versioning system with a central repository. Each client makes changes to its own copy of the shared diagram. Once in a while, a client commits the changes to the central repository, and the changes are merged into the shared diagram.

5. Conclusion

As an example of tool integration, we described the integration between the Knight tool and existing CASE tools. Two different types of integration were investigated and implemented: asynchronous integration with shared data implemented using a standard interchange format, and synchronous integration with separate data implemented using component technology. The two integration approaches complement each other: asynchronous integration with shared data via a standard interchange format is simple, robust, and lightweight. Synchronous integration with separate data via component technology supports a tight integration between tools.

Several problems and issues to be considered were highlighted: even though the asynchronous integration with shared data was realised using a standard interchange format, only part of the necessary information to be interchanged was standardised. For the synchronous integration with separate data, the situation was even worse. No standard way of defining interfaces for components dealing with UML has been implemented. Thus, even though both integration efforts have been successful, the way to flexible, interoperable, and integratable tools goes through even further standardisation efforts. Irrespective of standardisation, however, tools will need to store arbitrary information that must be handled in specific ways. This caused problems for both types of integration considered.

Nevertheless, we were successful in integrating the tools. We believe that our integration can be used not only as a source of inspiration, but also that the concrete design and the software architectures described are of general applicability, and that they can help developers in integrating other tools.

6. Acknowledgements

The work described in this paper was carried out at the Centre for Object Technology (`http://www.cit.dk/COT`), which has been partially funded by the Danish National Centre for IT Research (CIT).

7. References

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

2. Chen, M. & Norman, R.J. (1992). A Framework for Integrated CASE. *IEEE Software,* 9(2):18-22, March.

3. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction (CHI'2000)*, The Hague, The Netherlands.
4. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000) Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'2000)*, Sophia Antipolis and Cannes, France.
5. Demeyer, S., Ducasse, S., & Tichelaar, S. (1999). Why FAMIX and not UML? UML Shortcomings for Coping with Round-trip Engineering. In *Proceedings of <<UML'99>>,* Fort Collins, CO, USA, October 28-30.
6. Demeyer, S., Meijler, T.D., Nierstrasz, O., & Steyaert, P. (1997). Design Guidelines for Tailorable Frameworks, *Communications of the ACM*, pp. 60-64. October, vol. 40, no.10.
7. Ellis, C. A., Gibbs, S. J., & Rein, G. L. (1991). Groupware: Some Issues and Experiences. *Communications of the ACM*, pp. 38-58, 34(1).
8. W3C (1998). *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`.
9. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman.
10. Haines, G., Carney, D., & Foreman, J. (1997). *Component-Based Software Development / COTS Integration.* SRI Software Technology Review.
11. Jarzabek, S. & Huang, R. (1998) The Case for User-Centered CASE Tools. *Communications of the ACM*, 41 (8).
12. Kobryn, C. (1999) UML 2001: A Standardization Odyssey. *Communications of the ACM,* pp. 29-37, 42(10).
13. Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus.* Unpublished Ph.D. Thesis, University of Toronto.
14. McLennan, M.J. (1993) [incr Tcl]: Object-Oriented Programming in Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
15. MOF Revision Task Force. (1999) *Meta Object Facility Specification v. 1.3.* Document ad/99-06-05, Object Management Group.
16. Ousterhout, J. (1990). Tcl: An Embeddable Command Language, in *Proceedings of the Winter 1990 USENIX Conference*, January 22--26, Washington, DC, USA.
17. Reiss, S.P. (1990) Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July.
18. Robbins, J.E. & Redmiles, D.F. (2000). Cognitive support, UML adherence, and XMI interchange in Argo/UML. In *Information and Software Technology*, 42(2), 79-89, 2000
19. Rogerson, D. (1997). *Inside COM: Microsoft's Component Object Model*. Microsoft Press.
20. Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.
21. Shaw, M. (1996) Some Patterns for Software Architectures. In Vlissides, Coplien, Kerth (Eds.), *Patterns Languages of Program Design*, volume 2, Addison Wesley.
22. Tichelaar, S., Cruz, J.C., & Demeyer, S. (2000). Design Guidelines for Coordination Components. To appear in *Proceedings of ACM SAC 2000 - Track on Coordination*, ACM Press, March.
23. UML Revision Task Force (1999). *OMG UML v. 1.3: Revisions and Recommendations*. Document ad/99-06-10, Object Management Group, June.
24. XMI Partners (1998). *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`.
25. XMI Partners (1999). *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`.

In Proceedings of ECOOP'2000

Creative Object-Oriented Modelling:
Support for Intuition, Flexibility, and Collaboration in CASE Tools

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted

Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark

{damm,marius,miksen,tyrsted}@daimi.au.dk

Abstract. A major strength in object-oriented development is the direct support for domain modelling offered by the conceptual framework underlying object-orientation. In this framework, domains and systems can be analysed and understood using models at a high level of abstraction. To support the construction of such models, a large number of Computer-Aided Software Engineering tools are available. These tools excel in supporting design and implementation, but have little support for elements such as creativity, flexibility, and collaboration. We believe that this lack of support partly explains the low adoption of CASE tools. Based on this, we have developed a tool, Knight, which supports intuition, flexibility, and collaboration by implementing gesture based UML modelling on a large electronic whiteboard. Such support improves CASE tools, and can thus potentially lead to increased adoption of CASE tools and thus ultimately help improving the overall quality of development projects.

1. Introduction

Object-oriented programming languages provide many technical qualities but more importantly object-oriented languages and object-oriented development in general also provide a conceptual framework for understanding and modelling [19] [21]. This conceptual framework provides abstraction mechanisms for modelling such as concepts (classes), phenomena (objects), and relations between these (inheritance, association, composition) that allow developers to describe *what their system is all about* and to formulate solutions on a higher level of abstraction than program code.

In practice, developers build models at several abstraction levels. The program code can be seen as an executable model, but for purposes such as analysis, specification, documentation, and communication, models visualised graphically in the form of diagrams are often used. The Unified Modeling Language (UML [31]) is a prominent example of a graphical notation that is used for such purposes.

The so-called Computer-Aided Software Engineering tools (CASE tools) provide tool support for modelling. These can among other things help users to:

- create, edit, and layout diagrams,
- perform syntactic and semantic checks of diagrams and simulate and test models,
- share diagrams between and combine diagrams from several users,
- generate code or code skeletons from diagrams (forward engineering) and generate diagram or diagram sketches from code (reverse and round-trip engineering), and
- produce documentation based on diagrams and models.

But even though CASE tools offer these many attractive features, they are in practise not widely and frequently used [1][15][18]. CASE tools clearly support design and implementation phases, but have less support for the initial phases, when the focus is on understanding the problem domain and on modelling the system supporting the problem domain. To rectify this problem, we propose inclusion of support for intuition, flexibility, and collaboration, and that CASE tools should have more direct, less complex user-interfaces, while they should still preserve the current support for more technical aspects such as implementation, testing, and general software engineering issues.

We have implemented a tool, Knight, which acts as an addition to existing CASE tools. It uses a large electronic whiteboard (Figure 1) to facilitate co-operation and a mixture of formal and informal elements to enable creative problem solving. This paper discusses the support of the Knight tool for modelling through gesture based creation of UML diagrams.

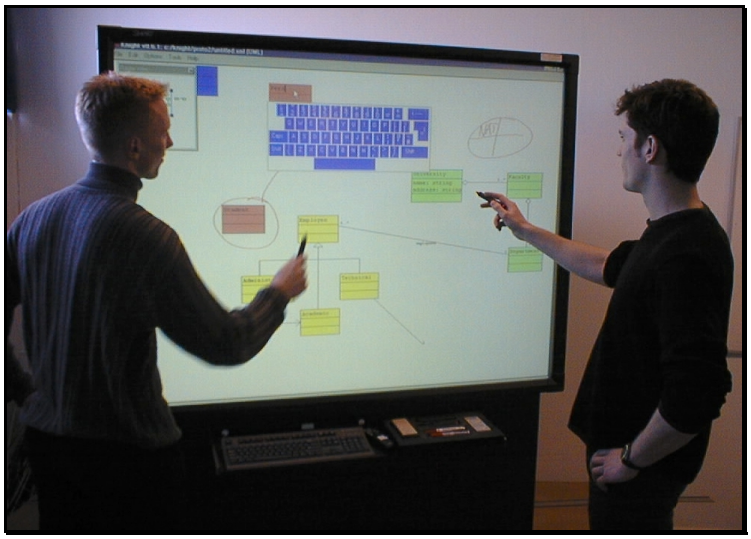


Figure 1. Use of Knight on an electronic whiteboard

Section 2 gives a theoretical overview of modelling, and section 3 discusses related work on current tool support for object-oriented modelling and the adoption of CASE tools. Section 4 presents studies of modelling in practice and section 5 discusses the Knight tool. Finally, section 6 discusses future work and section 7 concludes.

2. Modelling and Interpretation

Simplistically, object-oriented software development can be viewed as mapping a set of real-world phenomena and concepts to corresponding objects and classes [19][5]. The set of real-world phenomena and concepts is called the *referent system* and the corresponding computerised system is called the *model system* [19] (Figure 2).

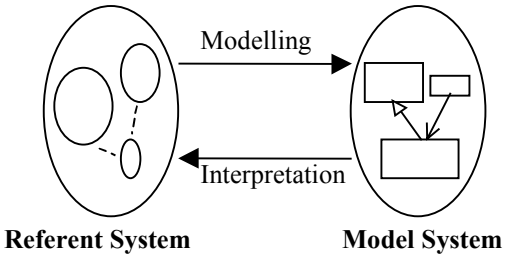


Figure 2. Modelling and interpretation

Mapping a referent system to a model system is called *modelling*. Modelling is often iterative and it is thus important to be able to discuss a model system in terms of the referent system. We call this reverse process *interpretation* [5]. Modelling is concerned with expressing an understanding of a referent system in a fluent, formal, and complete way, for which usable and formal notations (such as diagramming techniques and programming languages) are crucial. Interpretation is concerned with understanding a model system in terms of the referent system. For doing this, it is important that central concepts in a model system are understandable in the referent system.

Neither understanding the referent system nor building a model system is unproblematic. Understanding the referent system is within the domain of user-oriented disciplines such as ethnography [12] and participatory design [2][11]. The ethnographic perspective on system development is concerned with basing system design on actual work performed within the referent system. The rationale is to avoid a major cause of system failure, namely that the developed system does not fit the work practice within which it is to be used [12]. The participatory design perspective on system design is concerned with designing systems in active collaboration with potential users. This is done for both a moral and a practical reason: Users have to work with the future system, and users are competent practitioners who are knowledgeable of the referent system. Building the model system is within the domain of object-oriented software engineering. Software engineering is concerned with building robust, reliable, and correct systems. In iterative development, then, co-ordination, communication, and collaboration between different perspectives and

individual developers is crucial [5]. Thus, when considering tool support for a system development process, tool support for the modelling and the interpretation processes is important.

In iterative development, modelling and interpretation are interleaved. When different competencies work together, it is crucial that both processes are, to a certain extent, understood by all involved parties. For example, when doing user involvement it is both important that the developers understand the work of the users (i.e., the referent system) and that the users are able to react on and help design the application (i.e., the model system.)

3. Related Work on Tool Support

A large variety of tools that support software development, and modelling in particular, exist. In [28], Reiss presents an overview of 12 categories of software tools. CASE tools, or *Design Editors* as they are also referred to, are tools that let a user design a system using a graphical design notation, and that usually generate at least code skeletons or a code framework.

A large body of literature discusses the lack of adoption of CASE tool. In their study of CASE tool adoption [1], Aaen et al. showed that less than one fourth of the analysts in the companies studied used CASE tools and that about half of the respondents had used the tools for two projects or less. In [15], Kemerer reported that one year after the introduction of a CASE tool, 70 % of them were never used again.

In [13], Iivari confirms, and explains, the low adoption and use of CASE tools. Two of the main conclusions of the study are that high CASE tool usage does indeed increase productivity, but that a high degree of voluntariness in using the CASE tool tends to decrease the use of the tool. Important for our concern, the study also disclosed that many developers found it hard to appreciate CASE tools as they often were perceived as having a high complexity. From this, Iivari concludes that any intelligent means of reducing the perceived complexity might be a very profitable investment. Thus, the study to some extent supports our claim that the user-interface of CASE tools needs to be less complex and more direct.

A recent study by Lending and Chervany [18] examines a number of aspects of use of CASE tools. It examines whether CASE tools are perceived as being useful, and if developers using CASE tools use the same methodologies and perform the same activities as developers who do not. Finally, the study points to the features of CASE tools that are being used. Surprisingly, the participants who did use CASE tools, were quite neutral when it came to the perceived usability of CASE tools. Also, the study showed that users of CASE tools on average spent more than twice as much of their time doing analysis, and considerable less time on programming, test, and maintenance, than non-users did. Unfortunately, the study did not show whether this increased time spent on analysis actually resulted in a better analysis model, or whether it was due to overhead caused by the CASE tool.

The respondents indicated that only a small amount of the functionality offered by the CASE tool was actually used. They all used the facilities for creating and editing models and diagrams but used little else. The only other functionality that was used at

least “sometimes” was functionality to detect inconsistencies in diagrams, and to create documentation. The study in this way highlights that a focus on increasing the support for creating and editing diagrams in CASE tools is appropriate, and perhaps even the most important support.

In [14], Jarzabek and Huang present the view that current CASE tools are far too focused on hard aspects of software development such as software engineering. In their opinion, softer aspects such as support for creativity and idea generation are needed if CASE tools are to gain a wide acceptance. Concretely, they believe that CASE tools should allow the developer more freedom in expressing ideas and that their environment should be more tolerant, and allow the developers to think and work “at the level of application end users”. Also the authors argue that current CASE tools are too method-oriented, and that they should provide a more natural process-oriented framework. We concur with Jarzabek and Huang, and try to support creativity, through enhanced and more flexible support for modelling.

Usage of whiteboards as a support in creative, problem-solving meetings has been studied in several contexts [3][23]. Electronic whiteboards have consequently been used as a computational extension of traditional whiteboards. A goal in systems running on electronic whiteboards has often been to preserve desirable characteristics of whiteboards such as light-weight interaction and informality of drawings [26]. To our knowledge no one has investigated this use in the setting of Computer-Aided Software Engineering. We try to extend the box of tools for electronic whiteboards with a tool supporting object-oriented modelling.

4. Modelling and Interpretation in Practice

We have empirically studied the practice of modelling – and thus interpretation – in three distinct situations with three different user groups. The main results of each of the studies are given below. In each situation, the groups contained a mixture of competencies such as ethnographers, participatory designers, experienced and inexperienced developers, and end users. For a more detailed account, refer to [6].

4.1 The Dragon Project: Designing a New System

This study was carried out informally during the Dragon Project [5]. The project involved participants from a university research group and a major shipping company. Development in the Dragon project took place in active collaboration with users in the sense that at least one user was co-located with the development group at any given time. Whenever modelling of major conceptual areas of the shipping domain took place the users participated actively in this. Actual modelling in these sessions almost always took place on a whiteboard. Although most information from the users was in the form of domain knowledge as verbal or written accounts, drawings were often made by the end user on, or in connection to, the diagrams on the whiteboard. User drawings were informal, in contrast to formal UML models, which described key concepts or relationships from the shipping domain. Eventually, the users

nevertheless picked up parts of the UML notation and commented on, e.g., multiplicities on an association.

4.2 COT: Reengineering an Existing Application

We have studied a technology transfer project (COT, `http://www.cit.dk/COT`) involving a university research group and an industrial partner. The project reengineered an existing industrial application for control of a flow meter, while the inexperienced developers from the industrial partner learned object-oriented analysis and design. For the modelling, this project used a mixture of CASE tools, projectors, and whiteboards.

Simplistically, the inexperienced developers explained the existing implementation, whereas the experienced developers modelled a reengineered version. Eventually, this balance shifted as the inexperienced developers picked up larger parts of the UML and participated in the modelling. This introduced a certain number of syntactical errors in the use of the UML. These errors were either repaired, if they disturbed the shared understanding of what was modelled, or ignored, if the meaning was clear from the context. After each modelling session, photographs of the diagrams were taken for future record or for manual entry into a CASE tool. Programming was then done in an ordinary editor and the CASE tool was used to reengineer code into diagrams.

4.3 Mjølner: Restructuring an Existing Application

This study investigated the redesign of the Mjølner integrated development environment (`http://www.mjolner.com`). The group performing the redesign consisted of six developers with different experience in the domain of the integrated development environment. All developers had experience in object-orientation and a fair understanding of UML. Two of the developers had an in-depth knowledge of the development environment, other two developers had a knowledge of the part of the tool that they developed, and the last two developers were introduced to the software architecture of the environment while participating in the redesign.

During the redesign session, the most used artefacts were a whiteboard and a laptop. The whiteboard was used to draw the software architecture of the existing environment and to edit these drawings. The laptop was used whenever a developer needed to look at code in order to remember the actual architecture. This use took place whenever another person was at the whiteboard.

Although almost everything they drew was in actual UML notation, the notation was tweaked in three ways. First, UML did not suffice to explain certain aspects of the architecture leading to informal drawings of this. Second, the language used to implement the environment is BETA [19], which has a number of language and modelling constructs not supported by the UML. Two of these constructs, inner and virtual classes, were used heavily in the implementation, and thus the developers invented new notational elements on the fly. Third, the information drawn was

filtered in the sense that often only important attributes, operations, and classes were shown.

4.4 Key Insights

From our analysis of the user studies a number of lessons on the co-ordinative, communicative, and collaborative aspects of object-oriented modelling can be learned. We group the insights into the three categories 'tool usage', 'use of drawings', and 'collaboration'.

Tool Usage. Typically, a diagram existed persistently in a CASE tool as well as transiently on a whiteboard. CASE tools were primarily used for code generation, reverse engineering, and documentation whereas whiteboards were used for collaborative modelling and idea generation. This mix caused a number of problems: Whereas whiteboards are ideal for quickly expressing ideas collaboratively and individually, they are far from ideal for editing diagrams etc. This means that in all user studies, drawings have been transferred from whiteboards to CASE tools and from CASE tools back to whiteboards.

Use of Drawings. Most of the drawing elements were in the form of elements from UML diagrams such as class, sequence, and use case diagrams. However, these elements were combined with non-UML elements in two forms. Either as rich "freehand" elements that explained part of the problem domain or as formal additions to the UML such as notations for inner classes or grouping. Timings from one of the user studies show that approximately 25% of the meeting was spent on actual drawing on the whiteboard. The drawing time was divided into 80% for formal UML diagrams and 20% for incomplete or informal drawings.

Another key observation is the use of filtering. Filtering was used for several reasons. First, even whiteboard real estate is limited. Second, not all parts of a diagram are interesting at all times. Third, users may employ a specific semantic filtering to decide the important elements of a diagram. Such a filtering could, e.g., be that only the name of a class is shown, or that modelling is restricted to the part of an application related to the user interface.

Collaboration. It has been a striking fact in our user studies that all collaborative construction of models has been co-ordinated as turn-taking. This is somewhat in contrast to other observations on shared drawing [3], but we believe it to be general for the kind of work that object-oriented modelling is about.

What was, however, not co-ordinated via turn-taking, was verbal communication and the use of other artefacts. The people engaged in the meetings, e.g., discussed among themselves while another person was drawing at the whiteboard, or they used other artefacts concurrently.

4.5 Implications for Tool Support for Modelling

The user studies show that computerised support for collaboration and communication in modelling is beneficial. This includes support for turn-taking and not hindering communication. Moreover, context switches in turn-taking need to be fast and transparent to users. Also, possible tool support needs to integrate with a computational environment, i.e., provide functionality such as editing of diagrams, code generation, and reverse engineering. Integration of this functionality should not hinder collaboration.

Aspects of the UML notation were too restraining for initial modelling. A tool may try to help in several ways, including supporting semi-formal, incomplete drawings and integrated informal "freehand" annotations. Moreover, a formal notation is often not completely adequate for the problem at hand. Thus, it should be possible to tweak the notation on the fly, in such a way that the notation becomes more appropriate for the problem, while still preserving the original properties of the notation.

Filtering is needed in a wide sense. A CASE tool provides a potentially unlimited workspace, whiteboards do not. CASE tools often provide filtering mechanisms such as zooming, panning, and showing/hiding attributes on diagrams. On a whiteboard, on the other hand, it is possible to, e.g., contract several relationships into a single relationship. Both kinds of filtering, visual and semantic, are useful and should be integrated.

5. The KNIGHT Tool

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 1) as input and output device. This naturally enables collaboration via turn-taking among developers and users. The interaction with and functionality of the tool is discussed in the next sections.

5.1 Functionality and User-Interface

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 3) is very simple: it is a plain white surface, where users draw UML diagrams using non-marking pens.

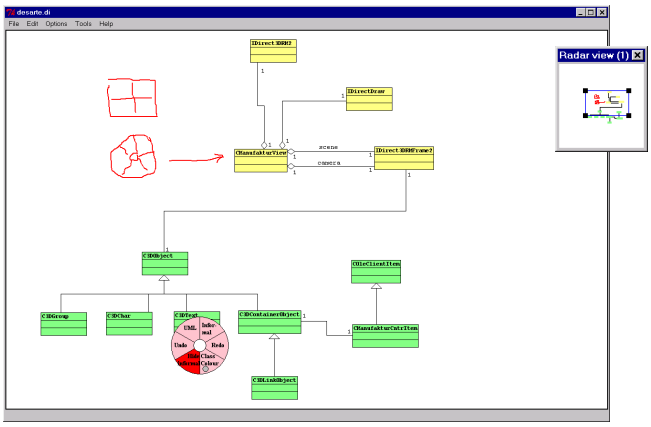


Figure 3. Knight user interface

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 4). Gestures have several advantages over existing toolbars: They allow input directly on the workspace, are fast to draw, and have a low cognitive overhead.

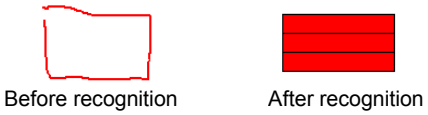


Figure 4. Recognition of the gesture for a class

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 so as to resemble what developers draw on ordinary whiteboards. This directness makes the gestures easier to learn and use.

Another way of achieving an intuitive interaction is by using compound gestures [17] and eager recognition [30]. Compound gestures combine gestures that are either close in time or space to one drawing element. So for example, when users draw an inheritance relationship on an ordinary whiteboard, they normally draw a line and then an overlaying triangle. Analogously, in the Knight tool, a user first draws a line between two classes and then a triangle at the appropriate line end.

With eager recognition, the tool continuously tries to classify gestures while they are being drawn. This is used for moving: To move an element, the user draws a *squiggle* gesture on the item to be moved. When the squiggle gesture can be classified with a high confidence, feedback is given in order to show that the gesture was recognised: the item follows the pen.

The technical aspects of the gesture recognition are discussed in the "Design & Implementation" section.

Informality vs. formality. A continuum from informality to formality is supported in two ways. First, users may draw incomplete diagrams, such as relationships only

belonging to one class (Figure 5). The incomplete elements can later be "completed", e.g., by attaching another class to the relationship.

Figure 5. A relationship with only one class specified

Second, a separate *freehand* mode is provided. In freehand mode, the pen strokes 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 3 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 the Knight tool, any number of floating radar windows may be opened (Figure 6). 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 *pie menus* that are opened when the pen is pressed down for a short while (see Figure 7). 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 [16]. Furthermore, the menus are context-dependent. The left side of Figure 7 shows the default menu, whereas the right side shows a more specialised menu that is opened when close to the end of a relationship.

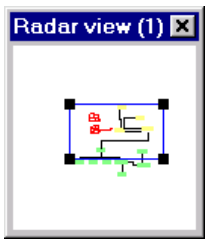


Figure 6. Radar windows provide context awareness

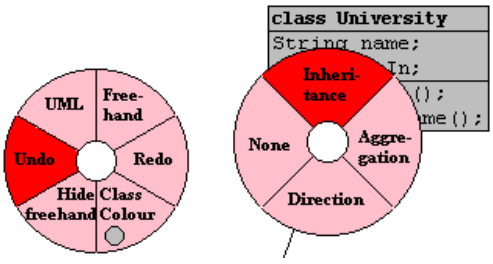


Figure 7. Context-dependent pie menus

Inputting text. The tool offers five different ways of inputting text (Figure 8). Apart from normal keyboard input these are: *Virtual keyboard* (a), *Stylus-based Gestures* (as on PDA's) (b), *Cirrin* (c) [20], and *Quikwrite* (d) [27]. They are shown in increasing order with respect to speed of text-entry and difficulty to learn. There is thus support for both casual users without any knowledge of the more specialised ways of inputting text, and advanced users who wish to enter text quickly.

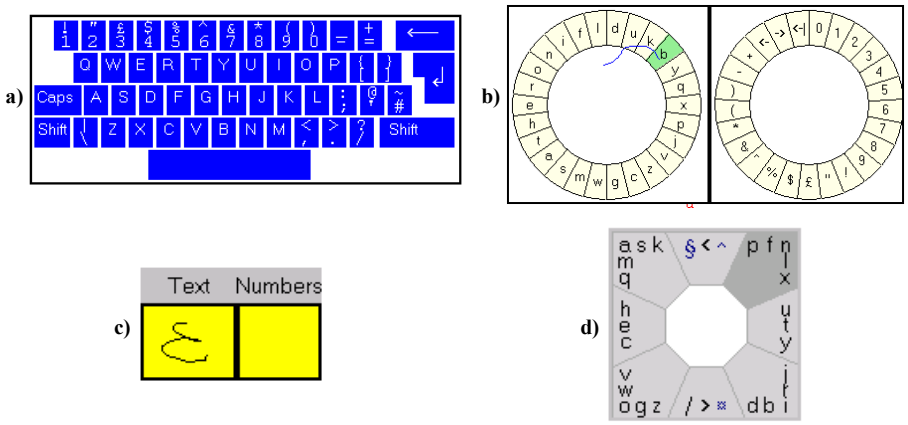


Figure 8. Text input possibilities in Knight

5.2 Design & Implementation

The Knight tool is implemented in the Itcl [22] object-oriented extension of Tcl/Tk [25]. Since the implementation uses Microsoft COM [29] for tool integration, it currently only runs on the Microsoft Windows platform.

Software Architecture. The software architecture of the Knight tool is shown in Figure 9 using a UML package diagram. The Knight tool and CASE tools that Knight is integrated with are separate processes. The connectors between these processes are, as discussed further below, currently implemented using Microsoft COM.

Internally in Knight, connectors are either event broadcasts or direct method calls. The structure of Knight follows the *Repository* architectural pattern [32]. The packages surrounding the Repository are *readers* and *readers/writers*. They work independently on the repository.

The *Repository* is a repository of UML diagrams. Basically, the class structure is a subset of the UML metamodel. When changes occur in the diagram, Observers [10] on the diagram are notified.

The *Radar* is observing the Repository and it gives an overview by reading and representing the state found in the diagrams.

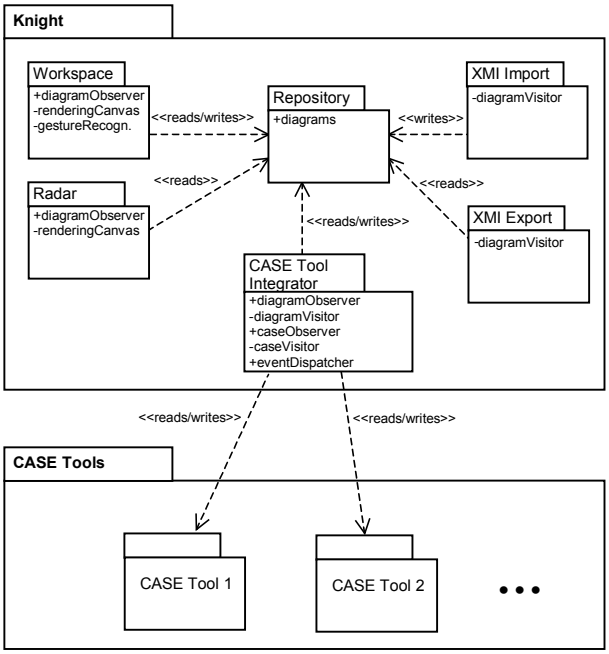


Figure 9. Software architecture of the Knight tool

The *Workspace* both reads and writes from and to the diagrams to display and edit these. Writing in the Repository is implemented using a combination of the Composite and the Command patterns [10], providing undo/redo functionality.

The *CASE Tool Integrator* is a Mediator [10] between the Knight tool and multiple CASE tools (see below).

Gesture Recognition. Rubine's algorithm [30] is used for gesture recognition. The main advantage of this algorithm is that it is relatively easy to train: For each gesture to be recognised, it must simply be provided with a number of prototypical examples of the gesture. Based on these examples, the algorithm then computes a representative vector of features. Examples of features are the total length of a gesture and the total angle traversed by a gesture. Subsequently, these representative vectors can be compared to a vector computed from a user's input, and the most resembling gesture can be chosen based on a statistical analysis.

A few circumstances complicate the gesture recognition a little. First, different types of input devices have different physical characteristics. Thus, the recogniser should ideally be trained once per type of input device. Second, Rubine's algorithm requires that different types of gestures are distinguishable with respect to the feature vector. We handle this by using compound gestures, e.g., when drawing different types of relationships, thus reducing the number of gestures to be recognised. Third, all users do not draw, e.g., a rectangle starting in the same corner and in the same direction. In order to preserve the intuitiveness of and the resemblance to ordinary whiteboards, the gesture recogniser must thus be able to recognise a rectangle starting

in all four corners going both clockwise and counter-clockwise. To lessen the burden of training the recogniser with all these variations, we have implemented a script that takes, e.g., a set of rectangle examples all drawn starting in one corner and going in one direction and then permutes these by rotating and mirroring to obtain eight combinations. In this way the full gesture set of 49 different gestures can be achieved from only 9 sets of examples.

Integration with other CASE tools. There is a need to provide common CASE tool functionality such as code generation and reverse engineering. Different tools support different tasks well and users choose to use different kinds of tools based on the work at hand. Thus, we have chosen an integration strategy instead of building a full-featured CASE tool.

We are implementing both batch and incremental integration with CASE tools. Batch integration is currently achieved in two ways: using *XML Metadata Interchange* format (XMI [24]) and using component technology. To implement the XMI exchange, we used a DTD based on the UML metamodel. Given this DTD, we implemented support for importing models from and exporting models to XML files following its grammar. Since the DTD is generated as specified in the XMI standard, it enables us to share models with other CASE tools that support UML and XMI.

Incremental integration is done for two reasons. First, batch integration may lose information and rely on other tools' interpretation of information. Second, we wish to integrate with CASE tools that support incremental round-trip engineering [4]. For this to be useful, incremental updates are needed so that code is always available and changes in code are immediately reflected in diagrams. The incremental integration is currently done via Microsoft COM [29].

The details of the integration are described in a companion paper [7].

6. Evaluation and Future Work

6.1 Evaluation

We have performed qualitative evaluations of the Knight tool in use. The primary objective of the evaluations has been to evaluate Knight in real work settings. Typically, a facilitator introduced the tool briefly. Following this, each subject was given a chance to try the interaction of Knight and learn the gestures for manipulating diagrams. The facilitator also helped if the subjects had problems using the tool in their work.

After the introduction, the subjects used Knight to work on their current project. While the subjects worked on their project, the use was videotaped and notes were taken. After the sessions, the subjects were interviewed.

The evaluations were all positive and showed that the tool was useful in the design situation. Also, the subjects considered the tool to be a better enabler for both collaboration and creativity than traditional CASE tools. They especially liked the large electronic whiteboard’s collaboration support and the tool’s interaction style, with its combination of informal and formal elements.

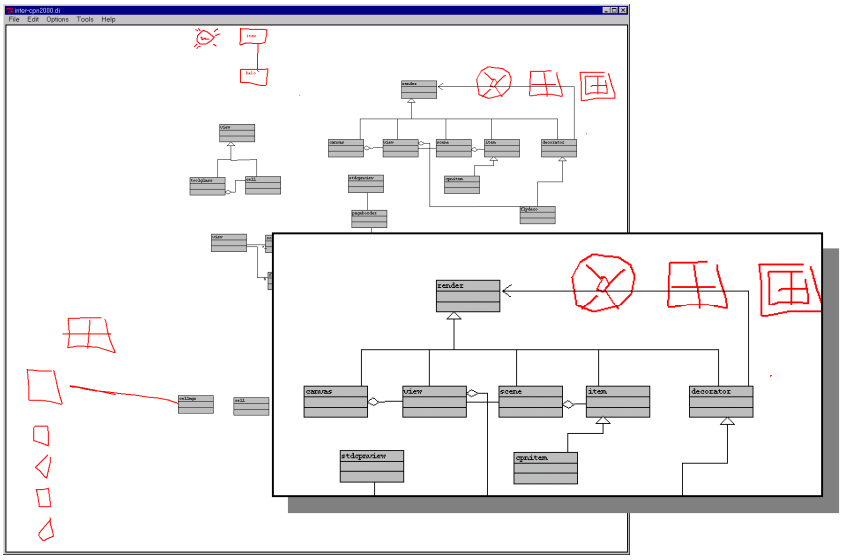


Figure 10. Diagram produced during an evaluation session (with part of the diagram emphasised)

A number of minor problems surfaced. A few of the gestures were hard to learn by some of the participants, although they seemed to learn them as they used the tool further. Also the integration of freehand, formal, and informal elements was generally construed as useful (see Figure 10), but further functionality was desired, e.g., to associate a freehand element and a formal element to each other.

In summary, the evaluations validate our general approach to supporting collaborative modelling. The issues raised as results of the evaluations were mostly smaller interaction problems, and these were outweighed by the tool's advantages. The users especially found that the tool's lighter, more intuitive interaction allowed for more creativity and that the tool less frequently than ordinary CASE tools caused breakdowns in the modelling.

6.2 Future Work

Longitudinal Studies of Actual Use. The tool should be evaluated in real settings over a longer period of time, preferably for the duration of a whole project. Further evaluation could also include a comparative, quantitative study of the Knight tool's impact on the quality of the resulting design and the amount of time spent using the tool.

Distributed Use. Considering distributed use of the Knight tool suggests some interesting possibilities. The obvious possibility is to connect two electronic whiteboards in different locations and to have designers work on the same model

synchronously. But other combinations may also be possible. For example, one might choose to have a different view on a model via an ordinary PC located in the same meeting room as the electronic whiteboard.

End User Customisation. An area of research that we so far have left more or less untouched is the ability for the user to customise the tool in a number of ways. Currently, we are working with the idea of a *personal pen* that holds several settings for the user. In this way, a user who chooses a specific pen will use the settings of the pen, such as gesture set, freehand/UML mode, or current colour, when interacting with the electronic whiteboard. A bit further along the way is the possibility for the user to customise the notation, ultimately "on the fly". Simple customisations of this type include changing the appearance of drawings or adding user-defined stereotypes to diagrams. A more elaborate tailoring scheme in which the user actively models on the metamodel of the tool may be possible. Such a tool could be considered a light-weight meta-CASE tool [8][9].

Generalisation. Obviously, full support for all the diagram types of the UML is desirable. Moreover, many of the observations that we have made of object-oriented modelling seem to be true of other kinds of formal modelling as well. This suggests that the Knight tool could support other kinds of diagramming with the same basic interaction.

Other generalisations are also possible: Even though the gesture based interaction has been designed for use on an electronic whiteboard, the interaction seems direct and natural and it would be worthwhile investigating its use on an ordinary PC with other input devices, e.g., tablets, mice, or trackballs. We would also like to explore other more exotic input/output devices such as tablets with built in LCD displays, connected PDAs communicating with a large screen, or electronic extensions of traditional whiteboards such as Mimio [33].

7. Conclusion

Support for modelling is one of the major advantages of object-oriented development. Through the conceptual framework underlying object-orientation, both the problem domain of the system and the system to be built can be understood and formulated.

It is often advantageous to have tool support for modelling. Tool support can, e.g., aid in creating and editing models, in checking the syntax and semantics of models and in generating code from the models. Studies have shown, however, that existing *CASE tools* are rarely used. We believe that part of the explanation for this lies in the poor support for intuition, flexibility, and collaboration in CASE tools. To experiment with and to prove the advantages of such support, we have designed and implemented a tool, Knight, which complements existing CASE tools. The Knight tool provides a direct and fluid interaction that in many ways resemble the interaction with regular whiteboards, it provides support for collaboration in its large shared workspace, and it allows for more flexibility by supporting models with both informal and incomplete elements.

The tool has so far been successfully evaluated in qualitative experiments that validated the basic design. Further evaluation and studies of use are to be performed, but we believe that an extension of current CASE tools with support for creativity, flexibility and collaboration as found in Knight can ultimately help in improving the overall quality of development projects.

Acknowledgements. We thank Ole Lehrmann Madsen for comments that improved this paper. This project has been partly sponsored by the Centre for Object Technology (COT, `http://www.cit.dk/COT`), which is a joint research project between Danish industry and universities. COT is sponsored by The Danish National Centre for IT Research (CIT, `http://www.cit.dk`), the Danish Ministry of Industry and University of Aarhus.

References

1. Aaen, I., Siltanen, A., Sørensen, C., & Tahvanainen, V.-P. (1992). A Tale of two Countries: CASE Experiences and Expectations. In Kendall, K.E., Lyytinen, K., & DeGross, J. (Eds.), *The impact of Computer Supported Technologies on Information Systems Development* (pp 61-93). IFIP Transactions A (Computer Science and Technology), A-8.
2. Blomberg, J., Suchman, L., & Trigg, R. (1994). Reflections on a Work-Oriented Design Project. In *Proceedings of PDC '94*, pp. 99–109, Chapel Hill, North Carolina: ACM Press.
3. Bly, S.A. & Minneman, S.L. (1990). Commune: A Shared Drawing Surface. In *Proceedings of the Conference on Office Information Systems* (pp. 184-192). ACM Press.
4. Christensen, M. & Sandvad, E. (1996). Integrated Tool support for design and implementation. In *Proceedings of the Nordic Workshop on Programming Environment Research*, Aalborg, May 29-31.
5. 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 *Proceedings of ECOOP'98*, Bruxelles, Belgium, July, Springer-Verlag, LNCS series, volume 1445.
6. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction (CHI'2000)*. Haag, The Netherlands, 2000.
7. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Tool Integration: Experiences and Issues in Using XMI and Component Technology. In *Proceedings of TOOLS Europe'2000*. Brittany, France.
8. Ebert, J., Süttenbach, S. & Uhe, I. (1997). Meta-CASE in Practice: a Case for KOGGE. In Olive, A. & Pastor, J. A. (Eds.): *Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97*, pp. 203-216, Barcelona, Catalonia, Spain, June 16-20, LNCS 1250, Berlin: Springer.
9. Englebert, V. & Hainaut, J.-L. (1999). DB-MAIN. A Next Generation Meta-CASE. In *Information Systems*, pp. 99-112, 24 (*2*).
10. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object/Oriented Software.* Addison-Wesley.
11. Greenbaum, J. & Kyng, M. (1991). *Design at Work: Cooperative Design of Computer Systems*. Hillsdale New Jersey: Lawrence Erlbaum Associates.

12. Hughes, J., King, V., Rodden, T., & Andersen, H. (1994). Moving Out of the Control Room: Ethnography in System Design. In *Proceedings of CSCW'94* (pp. 429-439). Chapel Hill: ACM Press.
13. Iivari, J. (1996). Why Are CASE Tools Not Used? In *Communications of the ACM*, 39(10).
14. Jarzabek, S. & Huang, R. (1998) The Case for User-Centered CASE Tools. In *Communications of the ACM*, 41(8).
15. Kemerer, C.F. (1992). How the Learning Curve Affects CASE Tool Adoption. In *IEEE Software,* 9(3).
16. Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus.* Unpublished Ph.D. Thesis, University of Toronto.
17. Landay, J.A. & Myers, B.A. (1995). Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of CHI'95*, 45-50.
18. Lending, D. & Chervany, N.L. (1998). The Use of CASE Tools. In Agarwal, R. (Eds.), *Proceedings of the 1998 ACM SIGCPR Conference*, ACM.
19. Madsen, O.L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley.
20. Mankoff, J. & Abowd, G.D. (1998). Cirrin: A Word-Level Unistroke Keyboard for Pen Input. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*.
21. Martin, J. & Odell, J.J. (1998). *Object-Oriented Methods. A Foundation.* Second Edition. Prentice Hall.
22. McLennan, M.J. (1993). [incr Tcl]: Object-Oriented Programming. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
23. Moran, T.P., Chiu, P., Harrison, S., Kurtenbach, G., Minneman, S., & van Melle, W. (1996). Evolutionary Engagement in an Ongoing Collaborative Work Process: A Case Study. In *Proceedings of CSCW'96,* 150-159.
24. Object Managment Group (1998). *XML Metadata Interchange* (XMI), document ad/98-07-01, July.
25. Ousterhout, J. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
26. Pedersen, E.R., McCall, K., Moran, T.P., & Halasz, F.G. (1993). Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. In Proceedings of *INTERCHI'93*, 391-398.
27. Perlin, K. (1998). Quikwriting: Continuous Stylus-Based Text Entry. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST).*.
28. Reiss, S.P. (1996). Software Tools and Environments. In *ACM Computing Surveys*, 28(1), CRC Press, March 1996.
29. Rogerson, D. (1997). *Inside COM. Microsoft's Component Object Model*. Microsoft Press.
30. Rubine, D. (1991). Specifying Gestures by Example. In *Proceedings of SIGGRAPH'91*, 329-337.
31. Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.
32. Shaw, M. (1996). Some Patterns for Software Architectures. In Vlissides, Coplien, Kerth (Eds.), *Patterns Languages of Program Design 2.* Addison Wesley, 1996.
33. Yates, C. (1999). Mimio: A Whiteboard without the Board. In *PC Computing*, June 28.

In Proceedings of UML'2000

Supporting Several Levels of Restriction in the UML

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted

Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark

{damm,marius,miksen,tyrsted}@daimi.au.dk

Abstract. The emergence of the Unified Modeling Language (UML) has provided software developers with an effective and efficient shared language. However, UML is often too restrictive in initial, informal, and creative modelling, and it is in some cases not restrictive enough, e.g., for code generation. Based on user studies, we propose that tool and meta-level support for several levels of restriction in diagrams and models is needed. We furthermore present a tool, Knight, which supports several levels of restriction as well as ways of transferring models from one level of restriction to another. This approach potentially increases the usability of the UML, and thus ultimately leads to greater quality and adoption of UML models.

1. Introduction

Modelling is an important activity in object-oriented development [18]. For visual modelling, the Unified Modelling Language (UML; [21]) has been widely accepted. Much research has focussed on extending, formalising, or refining the UML [8][9][3]. Another important research question is, however, to what extent the UML supports visual modelling as it is being carried out in actual practice? This paper seeks to answer this question, although not in a general sense, as we will mostly discuss the existing practice of initial, creative, and collaborative problem domain modelling. The question will be addressed from two perspectives: One focussing on the UML itself as a modelling language, and one focussing on the support by current tools and technologies for UML modelling.

The UML is widely used by practitioners, but our previous studies suggest that certain parts of the UML are too restrictive with respect to what can be described [5]. For example, while modelling, practitioners often draw incomplete relationships or represent concepts (classes) by illustrative freehand drawings.

Likewise, tools and technologies for UML-based modelling are in wide use. It is a striking fact, however, that Computer Aided Software Engineering (CASE) tools are not widely used and adopted [1][13][15][17]. Some of the explanations are that current CASE tools focus too much on hard aspects of software engineering, while

support for soft aspects such as collaboration, creativity, and idea generation is lacking [14].

This paper takes a closer look at the practice of initial, creative, and collaborative problem domain modelling through a number of concrete projects. Based on these, we argue for the usefulness of several levels of restriction of the UML metamodel for the usefulness of transferring a model element from one level of restriction to another. Furthermore, we present and discuss a tool, *Knight*, which illustrates the main ideas. The main characteristics of the Knight tool are: Support for a direct and fluid use, support for collaboration, and support for different levels of restriction in both model and presentation, ranging from incomplete, freehand UML diagrams to UML models close to code.

The rest of this paper is structured as follows. First, we analyse current modelling practice and its application of the UML and tools. Second, we present the Knight tool, which tries to overcome some of the problems uncovered in the analysis. Then, related and future work is discussed, followed by conclusions.

2. Current Modelling Practice

We will focus our examination of the UML's support for modelling practice on studies of two large development projects and on our own personal experience as system developers. The first project, Dragon [4], involved a research group and a large, globally distributed shipping company. The goal of the project was to implement a prototype of a global customer service system for the company. This was realised over a one and a half year period by the development of a series of successful prototypes. In this project, three of the authors participated actively and observed ongoing work.

The second project, Danfoss (`http://www.cit.dk/COT/case2-eng.html`), was concerned with implementing an embedded control system for flow meters. The project lasted a year and involved experienced developers from a research group and engineers from a private company. One of the authors participated in this project. This involved formal observations of work and active participation. Thus our approach was a mix of (ethnographic) observations and active involvement in the project studied [11].

Both projects used an iterative object-oriented approach to system development. Throughout development, UML was used on whiteboards and in CASE tools to visualise an emerging understanding of the problem and solution domains.

In order to focus the discussion, the next section presents three representative scenarios distilled from the studies. These scenarios will then subsequently be used as a basis for analysis of current modelling practice.

2.1 Scenarios

Christina and Tom are two developers working on an administrative system for a university. Lisa is a problem domain expert knowledgeable in the domain of university administration.

Scenario One: Using Incomplete and Freehand Elements. Christina and Tom have invited Lisa to a session in which they will start to model the student administration part of the administrative system. Since Christina and Tom have little knowledge of this part of the problem domain, Lisa explains her understanding of student administration. While doing this, she makes heavy use of a whiteboard, and she draws freehand diagrams of the study structure at the university. Christina and Tom, in collaboration with Lisa, then try to transform the verbal and graphical account to a class model, which contains the relevant aspects of the problem domain. Sometimes, Christina or Tom draws an incomplete association or generalisation, since it is not yet clear which classes are involved.

Scenario Two: Shifting Between Levels of Completion. Christina and Tom continue working on the diagram after the initial session with Lisa. Because they work in an iterative manner, they already have a running version of the overall administrative system, and they now want to incorporate the newly discovered student details into the system. Since the previous session was done in a flexible and creative manner, the diagram contains a lot of non-UML elements such as incomplete relationships and freehand drawings. They implement the model in code. In the process they transform the problematic parts in simple ways, e.g., by converting freehand drawings to verbal descriptions in the form of comments, and by either deleting the incomplete relationships or creating stub classes for the dangling ends. During implementation, they discover conceptual errors, which they fix in the code. For later sessions with Lisa, these changes will have to be consistent with the original diagram.

Scenario Three: Using a UML Tool. The development team uses a traditional CASE tool for various software engineering tasks such as code generation, configuration management, and documentation. Christina and Tom thus need to add the student administration model and diagram that they modelled with Lisa to their UML repository. In order to do that, they have taken photos of the whiteboard with the diagram in various stages of completion. Next, they create image files for the important freehand drawings. They then redraw the diagram in their CASE tool by hand, making sure that all parts of the model conform to the UML. References to the photos are then connected to relevant parts of the diagram.

2.2 Analysis

The three scenarios give examples of some of the obvious characteristics of practical UML modelling. This includes the use of different types of information as well as the shifts between these.

Scenario one shows that freehand drawings are used in the early phases of modelling and that sometimes open-ended modelling using, e.g., freehand drawings and incomplete relations are appropriate. At other times, a close connection to code is essential, and this puts some restrictions on the structure of the models.

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, even some of the overall structures, and it is of less importance 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.

At some point, the model is sufficiently mature to be turned into code. It is, however, not satisfactory to have to wait until the model is complete, before turning it into code, and hence the question arises: how to turn a model with incomplete elements into code? The solution adopted in the scenarios was to fill the holes manually with default values, e.g., setting potential attributes' types to "void". By doing this, we lose information about the model, and at a following analysis/design session, the model looks different from the previous session. In the later phase, it is thus necessary to have a more restricted model, and therefore also less expressive modelling abilities.

As a project progresses, the models get more elaborated in some areas, but new areas are also investigated, like the student administration in the scenarios. This means that the same model can be stable and detailed in some areas, while being initial and containing incomplete elements in other areas.

As is evident in the scenarios, modelling is often done in collaboration, with several developers working together and often also involving domain experts. When people with different competencies work together, these people use different means to express their ideas. In scenario one, Lisa was not a computer expert, so she used the whiteboard to draw sketches. Christina and Tom, on the other hand, focused on creating a class diagram and thus used the UML notation on the whiteboard.

Also, scenario three highlighted the large gap between whiteboards and traditional UML tools. The whiteboard contained many elements that had to be modified to fit in the UML tool, and the transfer itself was manual and cumbersome. Moreover, since development was iterative, such shifts will often occur.

3. UML Tool Support for Modelling Practice

The fact that the UML metamodel disallows practices used in modelling suggests that the UML metamodel should be modified. As the analysis in section 2.2 shows, the way of expressing knowledge in a model varies. Sometimes, the flexibility to create freehand drawings and incomplete elements is needed. At other times, the model should be sufficiently restricted to allow for direct code generation.

One solution would be to make the UML metamodel more expressive so as to allow models to contain all kinds of elements needed as well as all kinds of relationships between those elements. However, in order for a model to be useful for,

e.g., code generation, it is necessary that the model is restricted only to contain elements that have a straightforward translation to programming language constructs. As an example, Bunse and Atkinson propose the *Normal Object Form* (NOF) [3], which is a version of the UML metamodel that is closer to object-oriented programming languages and therefore better supports code generation from models. Thus, there *should* be support at the metamodel level for different levels of restriction.

Instead of having several more or less incidental metamodel, the UML metamodel itself could contain different *levels*. The levels of the UML metamodel should cover the range from very expressive models to models close to code. There should, at least, be the following levels:

- A flexible level that supports freehand drawings, incomplete elements, etc.
- A level that allows the models to be analysed (this could be the current UML metamodel)
- A level, in which models are close to object-oriented programming languages, so that code generation is possible (this could be the NOF)

Ideally, the UML should support a continuum of metamodels at different levels of restriction, but, realistically, a fixed number of standardised levels should be incorporated in the UML

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. It is interesting to consider how these varying meta-models and meta-presentations may be coupled. At times, a very restrictive meta-model and a very restrictive meta-presentation will be useful, such as when exchanging models and diagrams between organisations. At other times, there will be a need for a very expressive meta-model coupled to a very expressive meta-presentation, such as when brainstorming. Also, a coupling between a restrictive meta-model and an expressive meta-presentation is beneficial: If domain experts are involved in modelling, a presentation much like that on a whiteboard is useful.

Such a modification of the UML that supports different levels of restriction will help modelling practice. However, there is also a need for tools to support those levels. The next section describes a prototype of a tool, the *Knight tool*, which is designed to support flexible and creative modelling. We have used the Knight tool to experiment with different levels and the transition between those levels.

3.1 The Knight Tool

The Knight tool is a tool for the initial analysis and design in object-oriented software development. With a lightweight interface, it combines freehand drawings with UML diagrams. We have found that this is very important for the modelling in early phases of object-oriented software development [5]. Problems of today's Windows, Icons, Menus, and Pointers (WIMP) interfaces are many and well known [2]. The Knight

tool tries to overcome such problems by basing the interaction on gestures, marking menus, and a whiteboard metaphor.

The Knight Tool is implemented in Itcl [19], an object-oriented extension of Tcl/Tk [21]. For further information on the Knight tool, refer to `http://www-.daimi.au.dk/~knight` or [5][6][7].

Interaction with Few Breakdowns. The whiteboard metaphor is the cornerstone of the Knight tool (Figure 1): The user interface is initially a blank, white surface on which the user draws strokes. These strokes are then recognised as gestures using Rubine's algorithm [24] and interpreted as commands.

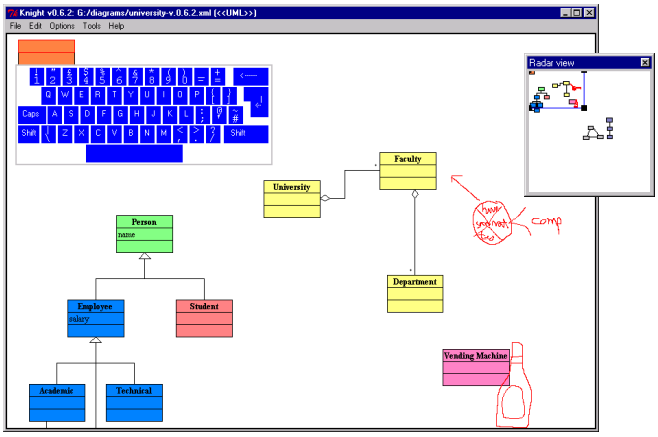


Figure 1. User interface of the Knight tool

The gestures resemble what is typically drawn on an ordinary whiteboard. For example, the gesture for drawing a class is a box, as shown in Figure 2.

Figure 2. Gesture recognition in the Knight tool.

In addition to the gestures for creating UML elements, there are also gestures for moving, editing, and deleting elements. Less common commands may be accessed through a context-sensitive pie menu, which is activated by pressing on the drawing surface for a moment.

The rationale for using gestures is to make the interaction fast and fluid: few breakdowns and focus shifts should be caused by the interaction with the tool. So far, evaluations with users have supported this claim [5].

Collaboration Support. The Knight tool may be used on an ordinary workstation with a mouse or a tablet. If used on an electronic whiteboard, collaboration between a number of developers and domain experts is supported (Figure 3).

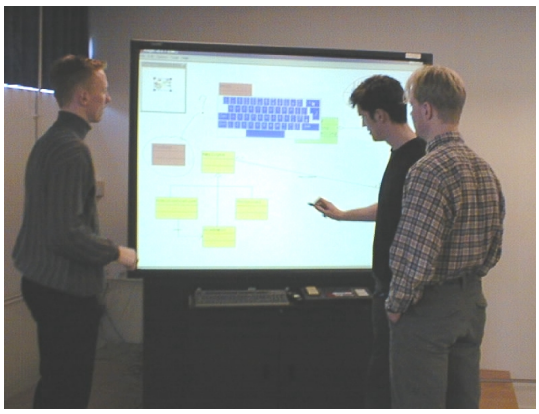


Figure 3. Collaborating using the Knight tool on an electronic whiteboard

We have used the Knight tool on several types of electronic whiteboards, including the SMART Board (`http://www.smarttech.com`) and the Mimio (`http://-www.mimio.com`).. The SMART Board (Figure 3) is a large, touch-sensitive surface, whereas the Mimio is a small device, which can be attached to ordinary whiteboards. Given the Knight tool and a projector, both technologies gives a total interaction much like that on a whiteboard.

3.2 Knight's Support for Varying Levels of Restriction

From the analysis in the previous section, we derive that a UML tool should support metamodels with differing levels of restriction, since modelling is not done on one level of abstraction at all times. Also shifts between levels should be supported. This section discusses the current support for varying levels of restriction in the Knight tool.

Shifting Between Levels of Restriction. As identified above, there is a need for shifting between the levels of expressiveness and restriction. A user may use a tool in three ways to transform a model, so that it conforms to a certain restriction level.

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

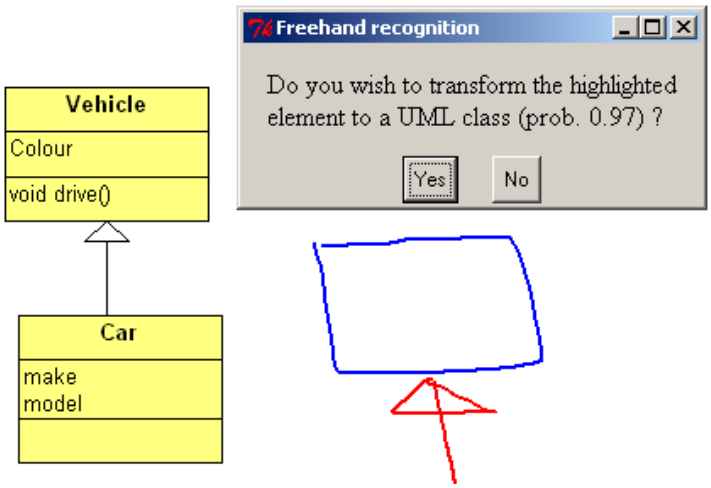


Figure 4. Guided transformation of freehand elements

The Knight tool implements all three kinds of support to varying degrees. Initially, strokes are conceptually elements in an unrestricted meta-model and -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. These are all examples of automated transformations.

Guided transformations are also supported in several ways. The user can 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 turns 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 model has been restricted. Furthermore, the user can ask the tool to guide a transformation where freehand elements are restricted into UML elements. The tool will then iterate through the individual marks in the drawing and present the user with a possible restriction (see Figure 4).

Figure 5 shows examples of manual transformations. In the example, a freehand drawing showing important details of the concept of a “Route” has been drawn. Depending on the context, the user may use 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.

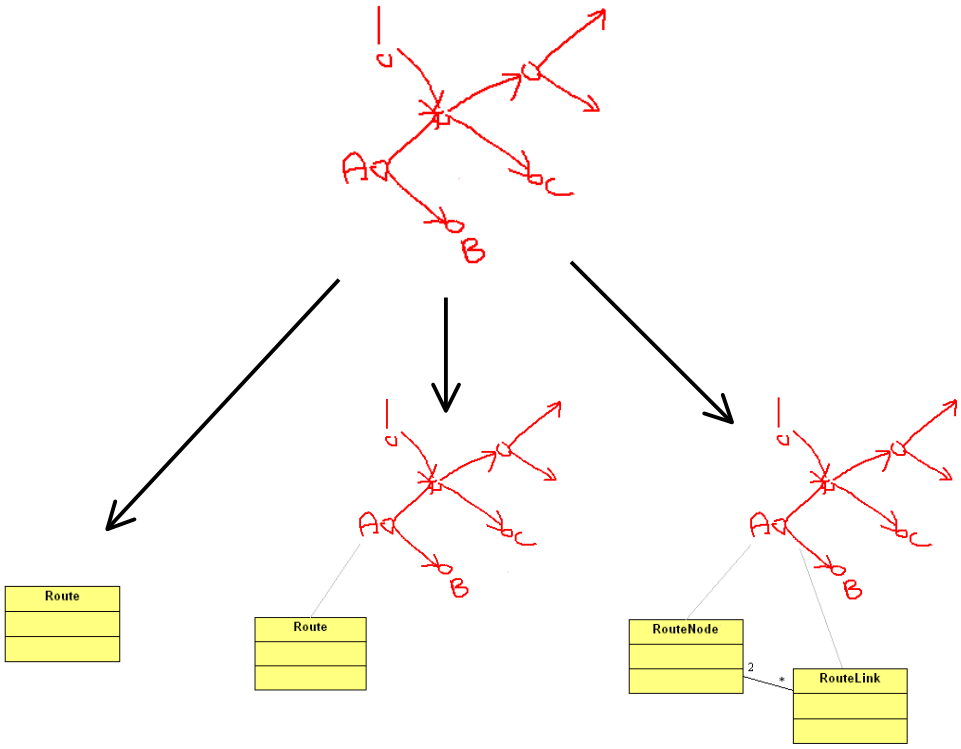


Figure 5. Restrictive transformations of a freehand drawing

Preserving the Look of Diagrams. Knight supports the creative phases of object-oriented modelling, because it allows users to create expressive models. For example, a UML class may not have a name yet, and only some of the attributes may be known and specified. However, in Knight and other UML tools, the elements often *look* finished, even when they are not (Figure 6).

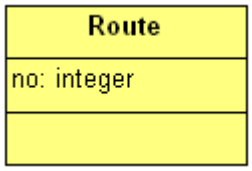


Figure 6. Unfinished elements may look finished

There is evidence that the appearance of, e.g., a diagram or a user interface influences how people perceive it [16]. 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.b

Knight supports preservation of the look of elements. An element can be displayed in three different ways:

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

These three looks are exemplified for a UML class in Figure 7a. In Figure 7b, the Route class from Figure 6 is displayed with an untransformed look, which suggests that it is unfinished.

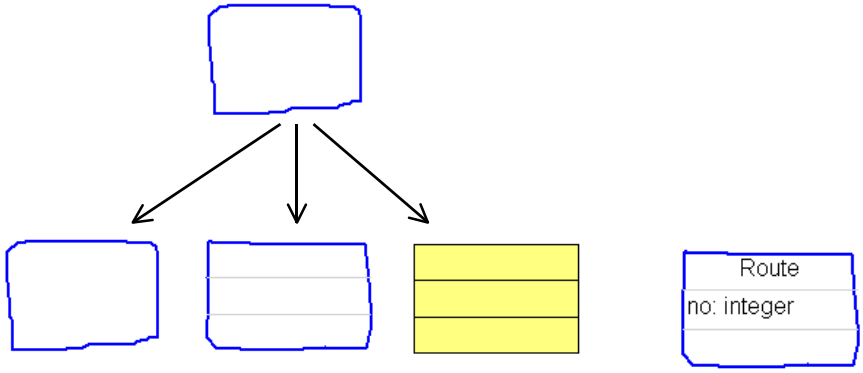


Figure 7. a) A class may be displayed in three ways in Knight, b) the Route class is not finished

It is possible for different elements in the same model to be displayed differently. In fact, as discussed in section 2.2, it is often the case that certain parts of a diagram are detailed and finished, while other parts have not yet been elaborated on. Displaying the elements corresponding to whether they are finished or not can show the situation to the users.

3.3 The Knight Tool at Work

The Knight tool was designed to support the modelling practice found in actual software development projects. In order to demonstrate *how* the use of Knight on an electronic whiteboard accomplishes this, this section will describe what the three scenarios from section 2.1 could look like, if the Knight tool was used.

Scenario One. Instead of using a traditional whiteboard, Lisa, Christina, and Tom use Knight. Because Knight supports freehand drawing, Lisa is able to illustrate the study structure directly in Knight. Christina and Tom gradually transform Lisa's freehand drawings to corresponding classes, while retaining the original drawings in the diagram.

Scenario Two. Christina and Tom have to transfer the model created together with Lisa to the overall administrative system. The model contains non-UML elements, but Christina and Tom use Knight's guidance to remedy some of the problematic

elements. The rest of the non-UML elements are handled automatically by Knight, which deletes some incomplete associations and adds default types to some attributes.

Scenario Three. Because the model was created directly in Knight, there is no need to capture the contents using a camera. Freehand elements may, moreover, be connected to UML elements directly. Also, the shifts to and from ordinary whiteboards are eliminated through the use of Knight on an electronic whiteboard.

Even though the scenarios highlight the positive features of Knight, there is room for a lot of improvement. Problems and plans are discussed in the next session.

4. Related & Future Work

As mentioned in the introduction, Jarzabek and Huang [14], among others, explain the low adoption of CASE tools with that they are too concerned with software engineering aspects of development. Instead, aspects such as creativity, flexibility, and idea generation should be focussed on and more widely supported. We concur that focussing on softer aspects of system development in CASE tool development may be a way for CASE tools to gain wider acceptance. This is the very basis for our work on Knight and is what we will continue working on also in the context of other formal notations than the UML. A very important means of ensuring that Knight supports this kind of work will be to set up longitudinal studies of Knight in use. We are currently trying to do this at several Danish companies. Although many of the basic mechanisms for creative modelling are present in Knight, we suspect that further refinements are needed. For example, it should be possible to incorporate other types of media than drawing such as photographs, videos, and sound in a seamless manner. Also, the use of and transition between different levels of restrictions will have to be refined.

In [12], Haake et al. present a taxonomy for categorising information-structuring systems based on the perspectives of the user and the system. The taxonomy has two dimensions. The first dimension describes how explicitly the user specifies the types of objects. Does the user, e.g., state that a specified object is of a certain type or not? The second dimension describes how the system represents the data. Does the system, e.g., retain or discard the type information? In our default transformation when users draw UML elements by gesturing, the user does not specify that it is, e.g., a class that is drawn. However, the system infers and represents the class via type information. In this sense there is no difference between selecting a class icon from a palette and drawing a gesture for a class. In the case in which the user draws a freehand drawing and then later transforms it manually, the situation changes from one in which the user has a typed representation of an element, whereas the system does not, to a situation in which both user and system has a typed representation. We concur with Haake et al. that this provides the users with a flexibility to support the creation of objects without the overhead of deciding types immediately. The focus of this paper has been on how presentation and model in UML has been represented. This is in

many ways system-centric, and one may envision a conceptualisation of levels of restriction in which cognitive aspects of users are taken into account.

In [3], Bunse and Atkinson propose the *Normal Object Form* (NOF), which is closer than UML to object-oriented programming languages. The NOF is basically a subset of UML with certain predefined extensions, and with some additional constraints on the structure of models. The authors make a distinction between *refinement* and *translation* of a UML model. The former involves adding details to the model, while remaining within the UML, and the latter is a transformation of the UML model to an object-oriented programming language. Thus, a model should be gradually *refined* until it satisfies the NOF criteria, and then it should be *translated* to code. The NOF is an example of a useful version of the UML with a specific purpose, and thus it supports our claim that there should be multiple versions of the UML. Also, models should be gradually transformed from one form (current UML, expressive) to another (NOF, restrictive), and this process could benefit from guidance or automation of a tool. The *refinement patterns* mentioned in the paper could play a role in the process, be it guidance or automation. Whereas NOF is a very restrictive version of the UML metamodel, what we propose is that the default, or lowest, level of restriction of the UML metamodel should be such that the flexible modelling described in this paper can be directly supported.

Another strand of future work is to experiment with the transportable Mimio technology, which is more lightweight than the SMART Board. A starting configuration for Knight using Mimio may be a mode, in which Mimio is used with marking pens but without projector. Strokes drawn may then either be interpreted during a session, yielding a possibility to give feedback, or be transformed to UML elements after the session using the techniques described in this paper. In-between this scenario and the one in which Mimio acts as an ordinary electronic whiteboard, a number of possibilities can be imagined. For example, UML elements may be drawn with a non-marking pen while freehand elements may be drawn with marking pens. One of our current goals is to offer a smooth transition between such scenarios.

Finally, the tool is being commercialised by Ideogramic (`http://www.ideogramic.com`). Part of this is, naturally, to support all diagram types in the UML.

5. Conclusion

The UML enables the construction of object-oriented models in a usable, visual way. However, in an object-oriented software development project – and in the process of creating a UML model – the UML is *not* sufficiently flexible, and hence it does not support all phases of object-oriented modelling. Likewise, existing tools do not support initial modelling phases in a satisfactory way.

Bunse and Atkinson [3] argue that, when coding starts, a restricted version of the UML metamodel is needed. We, on the other hand, argue that UML should support the initial phases of modelling, where incomplete diagram elements and freehand drawing are heavily used. The answer to both requests is to introduce different levels of restriction in the UML metamodel and to split it into meta-model and meta-

presentation. In this way, it is possible to combine expressive or restrictive meta-presentations with expressive or restrictive meta-models.

A change in the UML metamodel should be accompanied by appropriate tool support. Modelling involves going from more or less expressive diagrams to more restricted models, and tools should support this process. This implies that tools should be aware of the different levels of restriction and be able to assist the user in going from one level to another, either automatically or semi-automatically.

In this paper, we have presented the Knight tool, which was designed to support modelling practice. The Knight tool supports collaboration in modelling, fluid modelling, and shifts between levels of restriction in modelling. Using Knight, we have demonstrated that it is in fact possible to make tool support for actual modelling practice.

Acknowledgements. This project has been partly sponsored by the Centre for Object Technology (COT, `http://www.cit.dk/COT`), which is a joint research project between Danish industry and universities. COT is sponsored by The Danish National Centre for IT Research (CIT, `http://www.cit.dk`), the Danish Ministry of Industry and University of Aarhus. Thanks to Ole Lehrmann Madsen for comments that improved this paper.

References

1. Aaen, I., Siltanen, A., Sørensen, C., & Tahvanainen, V.-P. (1992). A Tale of two Countries: CASE Experiences and Expectations. In Kendall, K.E., Lyytinen, K., & DeGross, J. (Eds.), *The impact of Computer Supported Technologies on Information Systems Development* (pp 61-93). IFIP Transactions A (Computer Science and Technology), A-8.
2. Beaduoin-Lafon, M. (2000). Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of Computer Human Interaction (CHI'2000)*. The Hague, The Netherlands.
3. Bunse, C., Atkinson, C. (1999). The Normal Object Form: Bridging the Gap from Models to Code. In *Proceedings of <<UML>>'99 – The Unified Modeling Language*, pp. 675-690, Fort Collins, CO, USA, October.
4. 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 *Proceedings of ECOOP'98*, Bruxelles, Belgium, July, Springer-Verlag, LNCS series, volume 1445.
5. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction (CHI'2000)*. The Hague, The Netherlands.
6. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. To appear in *Proceedings of ECOOP'2000*. Cannes, France.
7. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Tool Integration: Experiences and Issues in Using XMI and Component Technology. To appear in *Proceedings of TOOLS Europe'2000*. Brittany, France.
8. Egyed, A. & Medvidovic, N. (1999). Extending Architectural Representation in UML with View Integration. In France, R. & Rumpe, B. (Eds.) *<<UML>>'99 – The Unified*

Modelling Language. Beyond the Standard. Second International Conference. LNCS 1723, Springer Verlag.
9. Evans, A. & Kent, S. (1999). Core Meta-Modelling Semantics of UML: The pUML Approach. In France, R. & Rumpe, B. (Eds.) *<<UML>>'99 – The Unified Modelling Language. Beyond the Standard.* Second International Conference. LNCS 1723, Springer Verlag.
10. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object/Oriented Software.* Addison-Wesley.
11. Greenbaum, J. & Kyng, M. (1991). *Design at Work: Cooperative Design of Computer Systems*. Hillsdale New Jersey: Lawrence Erlbaum Associates.
12. Haake, J.M., Neuwirth, C.M., Streitz, N.A. (1994). Coexistence and Transformation of Informal and Formal Structures: Requirements for More Flexible Hypermedia Systems. In *Proceedings of the 1994 ACM European conference on Hypermedia technology*, pp. 1-12. Edinburgh, Scotland.
13. Iivari, J. (1996). Why Are CASE Tools Not Used? In *Communications of the ACM*, 39(10).
14. Jarzabek, S. & Huang, R. (1998) The Case for User-Centered CASE Tools. In *Communications of the ACM*, 41(8).
15. Kemerer, C.F. (1992). How the Learning Curve Affects CASE Tool Adoption. In *IEEE Software,* 9(3).
16. Landay, J.A. & Myers, B.A. (1995) Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of Computer Human Interaction (CHI'95).*
17. Lending, D. & Chervany, N.L. (1998). The Use of CASE Tools. In Agarwal, R. (Eds.), *Proceedings of the 1998 ACM SIGCPR Conference*, ACM.
18. Madsen, O.L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley.
19. McLennan, M.J. (1993). [incr Tcl]: Object-Oriented Programming. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
20. Object Managment Group (1998). *XML Metadata Interchange* (XMI), document ad/98-07-01, July.
21. Object Managment Group (2000). *OMG Unified Modeling Language Specification*, document formal/00-03-01, March.
22. Ousterhout, J. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
23. Rogerson, D. (1997). *Inside COM. Microsoft's Component Object Model*. Microsoft Press.
24. Rubine, D. (1991). Specifying Gestures by Example. In *Proceedings of SIGGRAPH'91*, 329-337.

In Proceedings of NWPER'2000

Tool Support for Iterative System Development Activities: Issues and Experiences

Carsten Juel Andersen[*], Klaus Marius Hansen[**],
Elmer S. Sandvad[***], Michael Thomsen[**], Michael Tyrsted

*: Technological Institute, Information Technology
Teknologiparken, 8000 Aarhus C, Denmark
carsten.juel.andersen@teknologisk.dk

**: Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark
{marius, miksen, tyrsted}@daimi.au.dk

***: Presently at Mjølner Informatics, Science Park Aarhus
Gustav Wieds Vej 10, 8000 Aarhus C, Denmark
ess@mjolner.com

Abstract. In the past three years, seven system development projects involving industrial and academic partners have been carried out within Centre for Object Technology. Central to all projects has been an intensive use of a diversity of object-oriented tools and techniques. Based on the experiences from the projects, we discuss tool support for central activities in iterative development and point to directions for future research in development of usable object-oriented tools. Even though a variety of different (CASE) tools can collectively support an iterative, object-oriented system development process, we argue that there is a need for a deeper understanding of, and a better support for, concrete activities in system development practice. Furthermore, there is and cannot be a perfect tool for iterative object-oriented system development: tool developers also need to focus on the art- and skilful interplay and integration of separate tools.

1. Introduction

Centre for Object Technology (COT) is a large, joint research project between universities, approved technological service institutes, and industry in Denmark that carries out research, development, and technology transfer in the area of object-oriented development [COT, 1999]. One of the main research areas of COT is tool support for (cooperative) object-oriented development.

This paper is concerned with evaluating tools used in two cases encompassing seven projects of varying size and scale. See Table 1 for an overview. The largest is the Dragon Project (case 5) [Christensen et al., 1998] that involved more than 20

Projects	Result	Dura–tion	Partners	Artifacts	Tools
Dragon: Object-Oriented Architecture of a Global Customer Service System (Case 5) **References:** [Christensen et al., 998], [Dragon, 2000]					
Dragon	Prototype of a global customer service system	Feb 1997 – Jun 1998	University of Aarhus, Mærsk Line	Whiteboard snapshots, class diagrams	Mjølner System, DESCRIBES
Architecture of Embedded Systems (Case 2) Reference: [Embedded, 1999]					
Danfoss Instruments Pilot Project I (DIPP1)	Evaluation of an existing framework and a domain model for a flow measurer	Dec 1997 – Jan 1998	University of Aarhus, Danfoss Instruments, Technological Institute	Whiteboard snapshots, post-it notes, use cases, class diagrams, sequence diagrams, state charts	Rational Rose
Bang & Olufsen Pilot Project I (BOPP1)	Model and prototype of software for a CD player	Jan 1998 – Mar 1998	University of Aarhus, B&O, Technological Institute	Whiteboard snapshots, CRC cards, use cases, class diagrams, sequence diagrams, state charts, collaboration diagrams	Digital camera, Rational Rose (20%), Visio (80 %), GNU Emacs, SMART Board
Danfoss Drives Pilot Project I (DDPP1)	Model and prototype of software for a VLT frequency transformer	May 1998 – Nov 1998	University of Aarhus, Danfoss Drives, Technological Institute	Whiteboard snapshots, use cases, class diagrams, sequence diagrams, state charts	Rational Rose, Qualiware, Sequence Diagram Generator v0.1, ClearCase
Bang & Olufsen Pilot Project II (BOPP2)	Model and prototype of software for a B&O Audio / Video system equipped with Master Link	Nov 1998 – Feb 1999	University of Aarhus, B&O, Technological Institute	Whiteboard snapshots, use cases, class diagrams, sequence diagrams, state charts	Digital camera, Snapshot Composer, WithClass, Sequence Diagram Generator v0.2, Rhapsody, ClearCase
Danfoss Instruments Pilot Project II (DIPP2)	Model and prototype of software for a SONO flow measurer	Mar 1999 – Aug 1999	University of Aarhus, Danfoss Instruments, Technological Institute	Whiteboard snapshots, use cases, class diagrams, sequence diagrams, state charts	Digital camera, WithClass, Sequence Diagram Generator v0.3
Danfoss Drives Pilot Project II (DDPP2)	Common object model and corresponding prototype for several types of frequency transformers	Apr 1999 – Aug 1999	University of Aarhus, Danfoss Drives, Technological Institute	Whiteboard snapshots, use cases, class diagrams, sequence diagrams, state charts, package diagrams	GDPro, A0 plotter

Table 1. Overview of selected COT projects

people in over a year and the smallest is DIPP1 that involved three persons during a couple of weeks. Case 2 and case 5 have very different problem areas. Case 2 consists of a number of *pilot projects* that are concerned with development of (technically) embedded systems with special requirements for time and space efficiency, including real-time constraints. An incremental approach to introduction of object-oriented technology has been used, typically in projects in which a central part of an existing non-object-oriented application has been substituted by an object-oriented model and legacy wrappers. Case 5, on the other hand, was concerned with the development of a global customer service system, which can be characterized as a socially embedded system with focus on business data and case handling, involving a large number of

people. The development approach was experimental and evolutionary prototyping with participation of a variety of users.

Table 1 also shows for each project the development artifacts that have been produced and the tools used. We strongly emphasise that this paper should not be considered an evaluation of the qualities of specific tools, but rather an experience report on how different tools supported our development processes. Some specific tool facilities are discussed in this paper, but this should not be seen as an evaluation of the tool's overall qualities. For pointers to the tools used refer to the "Tools and Technology" section of the references on the last page.

Common to all the projects is an intensive use of object-oriented techniques. Through the use of these techniques, object-orientation is viewed as a conceptual framework for modelling parts of the real world in terms of classes, objects, and relations rather than a set of technologies [Madsen et al., 1993]. The view of object-oriented system construction as a modelling process is reflected in the nature of the development processes employed in the various projects: all projects have gone through a series of experiments in order to model an increasing understanding of the part of the real world that is of concern.

Iteration was another key characteristic of all projects. The concrete process used in each of the projects varied, but all projects were highly iterative and all used prototypes that were incrementally extended and refined in a number of development cycles. The processes of the projects were thus in accordance with other current iterative system development processes such as Extreme Programming [Beck, 1999] and Dynamic Systems Development Method [Stapleton, 1995].

Development Phases. Traditionally, a development process is seen as involving a set of phases such as *requirements gathering*, *analysis*, *design* and *implementation*. Analysis, e.g., is concerned with understanding the domain that the potential system is to be embedded in, whereas design is concerned with shaping and envisioning the technological future in this domain. Current CASE tools often support one or more of these phases. However, in a concrete system development project they are not concrete activities. Rather, requirement gathering, analysis, design, and implementation are abstract *concerns* and not what is actually carried out but rather something that matters in a project [Grønbæk et. al, 1997].

Activities, on the other hand, are concrete. They are what actually constitute the system development process. A concrete activity, such as a prototyping session with a potential end-user, may contribute to several abstract concerns. The actual purpose of such a meeting may be to design the graphical user interface of the future system, but, on the other hand, the meeting may also help the system developers understand the problem domain. The prototyping activity will have design as its major concern but will also contribute to the analysis concern. In accordance with this, this paper focuses on the experiences in tool support for central, concrete activities in the seven system development projects.

Structure of the Paper. The next section discusses a set of six system development activities that were important in our seven projects. For each activity we present our experiences and discuss both implications for the design of future tools and advice on using current tools. Finally, we conclude.

2. Tool Support for Central System Development Activities

This section presents our experiences with tool support from the projects introduced in the previous section. In presenting the experiences we will not use a taxonomy approach, such as describing tool features using e.g. [ISO, 1995], as our earlier work on using such an approach was unsuccessful. We tried to create a method-independent framework that could be instantiated for different methods, so that tools could be chosen according to the instantiation with a focus on support for concerns. However, this effort was not successful: the taxonomy was too abstract to be useable in selecting tools whereas the projects needed to work concretely from their start.

Our experiences on tool support obtained in the seven projects will instead be presented and discussed under the heading of six development activities: *Problem Domain Modelling*, *Solution Domain Modelling*, *Software Architecting*, *Coding*, *Documenting*, and *Testing*. As mentioned earlier all projects used an iterative approach to development, so it should be recognised that the activities thus were not always carried out in this order and not only once. More, we believe that most of these activities, and most likely others, will be present in other development contexts.

2.1 Problem Domain Modelling

Several forms of *domain modelling* are central to object-oriented modelling. Examples are *problem domain modelling*, often contributing to the analysis concern, and *solution domain modelling*, often contributing to the design concern. This section discusses problem domain modelling – solution domain modelling will be discussed in the next section.

In problem domain modelling, the focus is on understanding the problem domain. Without a proper understanding of the domain that the software system is to support, the system will most likely not match the users' needs. Common to early domain modelling activities is that they tend to be rather *informal.* In order to understand a domain, or to have a first attempt at modelling it, it is necessary to be able to discuss, sketch, and experiment in an open-ended, flexible, collaborative, and creative way. Often, however, the desired output of such activities is formal, e.g., in the form of code or UML diagrams [Booch et al., 1999] since the output is later used to produce other formal artifacts such as code. Thus, in domain modelling, a mixture of formal and informal techniques is used.

Experience. All seven projects have experience with this activity. The perception of the problem domain developed iteratively over time and was captured and expressed in a number of ways ranging from informal paper sketches to structured designs in CASE tools. All projects experienced a need for sketching models in informal and unstructured ways, although the sketches often also contained elements of a more precise character. As the projects progressed and the understanding of the problem domain evolved and was detailed, there was a move towards the use of more structured, formal models such as UML class and sequence diagrams. Thus, both formal and informal diagramming was important.

Use of Whiteboards. To support informal work, all projects used whiteboards intensively in the early stages of the projects. In the Dragon Project, a team of developers used the UML class diagram notation to create the initial versions of the problem domain model on a whiteboard. For the parts of the domain that were not well understood, small freehand notes or annotations were made (Figure 1, below). The other six projects used similar approaches although the exact use and concrete notation varied.

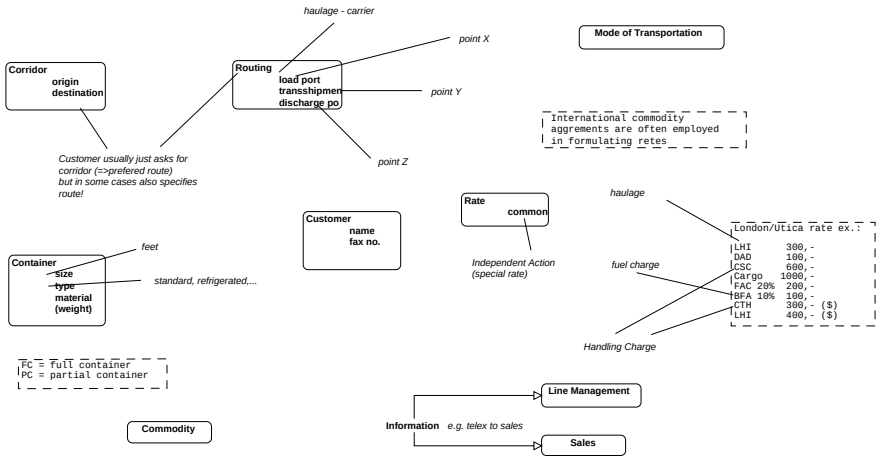


Figure 1. A whiteboard snapshot from the Dragon Project

While the use of whiteboards was open-ended, and thus was well suited for informal work, it also gave a number of problems, especially with respect to the iterative nature of the development. First of all a whiteboard has no support for saving a diagram other than instructing others not to erase it, which meant that some more permanent way of capturing the work was needed. In the Dragon Project, we initially manually captured the work by redrawing it on a piece of paper. As work progressed, we also redrew the diagrams in a CASE tool to obtain a more readable diagram.

Several projects experimented with the use of digital cameras for capturing the contents of whiteboards. Especially the BOPP2 project used a digital camera systematically. These photographs could then be printed and distributed to all developers. The diagrams still had to be manually restored on the whiteboard though. This was not a major problem, as typically only some parts of the diagram was needed for further discussion, and also often not all details were needed. In the cases were the full diagram was needed, we experimented with printing old diagrams on a transparency sheet, which could then be projected onto the whiteboard (Figure 2, below). This was useful when a diagram was needed on the whiteboard for reference, but not in the cases in which changes to the diagram were made, although we experimented with another technique: part of transparency sheets with class diagrams were cut out and the changes were drawn on the whiteboard in connection to the projection of remaining model.

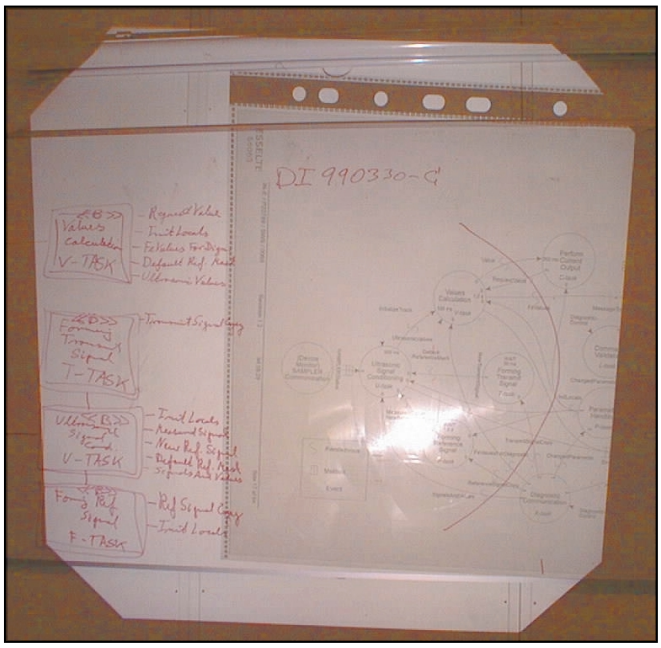


Figure 2. Projection of a previous diagram on a whiteboard

To facilitate easy capture of work, and to assist in preserving the context of the large number of photos taken, the BOPP2 project created a Web page containing a custom-made Java Servlet that facilitated upload of images. Small descriptions of the contents of the pictures could be added, and the Web application could thus help in tracing the evolution of the various artifacts of the project.

Use of CASE tools. As mentioned a large problem with whiteboards is their lack of computational power: whiteboards cannot store and restore diagrams, and they have weak support for diagram editing, which makes restructuring a diagram very laborious. To remedy this, several projects experimented with using both traditional drawing programs and CASE tools for problem domain modelling. These, however, were not well suited for this early, rather informal work. Firstly, working cooperatively at a standard PC is not practical for much more than two developers and in our experience cooperation between several developers is crucial in problem domain modelling. Secondly, interaction problems with the tools often slow down the creative process. Thirdly, the support for informal and incomplete elements is non-existing in most CASE tools.

To combine the natural support for group work at a large whiteboard and the editing power of a computerised tool, the BOPP1 project tried to use the Visio drawing program on a large electronic whiteboard (a SMART Board). While this did provide some of the desired benefits, it was in practice not a solution. The user-interface of Visio was not usable on a large screen since, e.g., the menu bars and palettes were very hard to reach from various positions around the electronic whiteboard. Furthermore, the general interaction with the tool was too impeding on

the work since even simple drawing operations took so long time that the developers felt it continuously disrupted their work. Finally, the electronic whiteboard also suffered from a low resolution, and impractical text input via an ordinary keyboard.

Use of posters. Many projects created large posters of various models. These posters where hung on office walls, and were used as reference points, and more importantly as centres for discussion. Often a couple of developers would gather at a poster hanging on the wall and use the poster as a common reference point in the discussion. Often they would also draw changes on it. In the Dragon Project, the posters were not created that often because they had to be constructed from normal printouts that were glued together. In a couple of the pilot projects, e.g., the DDPP2 project, a large format plotter capable of printing in A0 format was available. This made it very easy to produce new posters, which resulted in a large number of posters illustrating different aspects of the problem domain being available.

Discussion. In iterative development, tool support for modelling is especially important. Surprisingly, though, one of the activities that is not well-supported by CASE tools is diagram drawing. To layout is often difficult, time-consuming, and frustrating, and most tools support only one undo/redo operation at most.

Although diagramming is important, one of our recent studies [Damm et al., 2000a] of a development project shows that only about 25% of a design meeting with focus on diagramming was spent on drawing while the rest was used for discussion. Also, of the time spent diagramming, about 80% was spent on drawing formal UML diagrams while the remaining 20% was used to draw incomplete UML elements and freehand elements. Thus, if diagramming is important, incomplete and freehand drawing is as well. Actually, almost all projects experienced a need for informal modelling not supported by current CASE tools. This was clearly demonstrated by the fact that the projects supplemented CASE tools with an extensive use of whiteboards. From this we conclude that current CASE tools should try to provide better support for informal modelling and idea generation and thus concur with [Jarzabek & Huang, 1998].

One possibility for such a tool is the Knight tool as presented in [Damm et al., 2000a, 2000b, 2000c]. The Knight tool is designed to combine the interaction of the whiteboard with the functionality of a computerised tool. It is developed for use on an electronic whiteboard, and dry pens are used to draw directly on the surface of the electronic whiteboard. The tool recognises gestures resembling the drawings made on an ordinary whiteboard. This means, e.g., that whenever a user draws a rectangle, a class is created, and whenever a user draws a line, an association is created. In addition, Knight gives the developer the possibility of saving, loading, and editing the diagram in an easy manner. Figure 3 illustrates the interface of the tool, which intentionally is kept very simple. Because the interaction is based on gestures, the tool is suitable for use on a big screen. Moreover, the Knight tool integrates with existing CASE tools, adding sophisticated CASE tool capabilities to the tool.

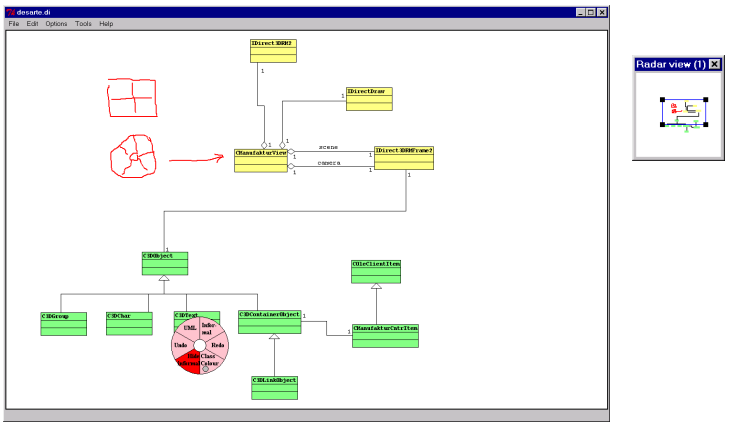


Figure 3. The user interface of the Knight tool

Nevertheless, while support for informal work is not present in CASE tools, several work-arounds exist. The projects were all successful in using whiteboards for the initial modelling, and we recommend the use of whiteboards as a complement to current CASE tools. To achieve support for, e.g., storing the diagrams, we had positive experiences using digital cameras. Another possibility is to use the Mimio technology. Attached to an ordinary whiteboard, this technology acts as a digital recorder of what is drawn on a whiteboard.

It must be recognised, though, that the lack of computational power of whiteboards can result in laborious work in order to transfer models manually, but we believe that this will not harm a project to any significant degree.

2.2 Solution Domain Modelling

As time elapses, the frequency of activities concerned with design and implementation grows increasingly. Whereas the activities in problem domain modelling have a strong informal component, later activities will be more focused on formal models expressed in diagrams, code, and other forms.

In such phases, then, iterative system development involves continual shifts in focus on analysis, design, implementation, and validation concerns. This means that tools, and techniques, for iterative development must support evolution on a number of levels and the ability to move seamlessly and losslessly between levels of abstraction, such as between diagrams and code, becomes crucial.

Experiences. In the Dragon project, functionality of the developed system was presented to users through a graphical user interface. This meant that the actual design of functionality was done by the developers and users by means of sketches and mock-ups of user interfaces. In this way, user interfaces were used to model the dynamics of the Dragon prototype. Also, solution domain models were created in code, but these were not discussed with users.

In the case 2 projects, UML sequence diagrams were often used for modelling dynamic aspects of the developed systems. When a new scenario was considered, the

scenario would be modelled as a sequence, and new classes would be introduced if needed. Following that, the scenario contributed to the class diagram or was implemented directly in code.

This technique illustrates some of the different strategies for handling iterations on models and code employed in the case 2 projects. In BOPP1, the model and code were kept consistent through a repeated time-consuming manual process, whereas DIPP2 handled the consistency problem through periodic reverse engineering of the developed code. In DDPP2, changes had to be done in the diagrams as well as the code, because of inconveniences in the reverse engineering (the class responsibility comments did not survive the reverse engineering).

In BOPP2, the model consisted of a class diagram and a number of sequence diagrams that modeled a number of representative use cases. The object-oriented model was going to substitute the central part of an existing application and wrapper classes to the surrounding, already existing, non-object-oriented code had been designed.

When the model had been developed to a certain stage in a common group, two different groups, BOPP2-A and BOPP2-B, with different strategies were formed. In the BOPP2-A group, legacy wrappers were implemented and integrated with the model that had been translated manually to C++. From time to time a new class diagram was reverse engineered (with WithClass) and the dynamic behaviour was tested using an automatic sequence diagram generator (see "Testing"). BOPP2-A thus primarily had a *focus on code*. Changes to the model were synchronised with the other group, the BOPP2-B group, in which the work continued with *focus on diagrams*. The model was elaborated and tested using the simulation capabilities of Rhapsody (see "Testing"). When the model had matured, the code was moved to the target platform with the legacy wrappers, and the application was practically running immediately. With the model as the common reference point, the two groups succeeded using a combination of two techniques: reverse and forward engineering.

The major benefits of the focus on code approach were that detailed changes to the code were easy and that developers had full influence on the quality of the code. The major liabilities were that structural code changes were cumbersome and that the reverse engineering process was inconvenient in some respects: time was spent on, e.g., improving the layout of the generated diagrams. Conversely, the major benefits of the focus on diagram approach were that the simulation capabilities were strong and that major structural changes were easy. The major liabilities were that the code generation was not completely flawless and that all structural changes, e.g., all declarations of classes, attributes, methods etc. had to be done in the CASE tool.

Discussion. From the projects we can conclude that there is a need for focusing both on diagrams and on code. Each strategy has benefits and liabilities. Diagrams and diagram editors are often an effective way of understanding and changing high-level structures, whereas code and editors often are more suitable for working on detailed implementation.

The fact that both foci are needed combined with the iterative nature of development creates a need for some way of reconciling the changes made in one artefact with the other. Tools should support this alternation between code and diagram, as manual transformation is both laborious and error prone. Not all tools

support effective round-trip engineering well, and those that do often have flaws or produce code of low quality. An important criteria in choosing a CASE tool is thus to evaluate to which extent the tool supports the concrete needs the project has in alternating between these two artifacts. Optimally, a CASE tool should support easy and incremental transitions between the model and implementation levels. During this, the tool should ensure consistency between diagrams and code. Only a few commercially available tools, such as Mjølner, support this.

2.3 Software Architecting

Software architecting is concerned with creating, maintaining, and changing structures of the software system under development in the form of components and connectors [Bass et al., 1998]. A structure may be seen from a number of views such as a logical, a process, or a module view [Kruchten, 1995]. In iterative system development, software architectures are continually created and elaborated. This then involves maintaining and creating large and complex structures with multiple views, which in turn poses challenges for tool support.

Experiences. In the Dragon Project, focus was on creating a well-established software architecture very early. Within two weeks an initial architecture had been designed on paper and implemented. This initial architecture enabled fast iterations, experiments with functionality, parallel development, and general "mud", i.e., areas in which the software structure was underspecified and allowed to evolve as the developers pleased [Foote & Yoder, 1997] [Christensen et al., 1998].

After several iterations, the mud had grown and an architectural restructuring was needed. Ad hoc tools for reverse engineering software architecture such as a UML CASE tool, a module structure viewer, and Grep helped us to understand the evolved software structure as opposed to the planned software structure. Subsequently, the structure editor in Mjølner was useful in syntactically performing the structure transformations necessary to realise the architectural refactorings. The editor was, however, unable to aid semantically, e.g., in resolving renaming problems, when restructuring.

In the pilot projects, the main tools used in architecting were CASE tools. In DIPP2, e.g., the project used a procedural and an object-oriented legacy system as its basis. The procedural system was documented in the form of informal drawings, prose, and data-flow diagrams, while the object-oriented system was documented mainly in the form of prose, sequence diagrams, and class diagrams. Common to these descriptions were their detachment from the systems they described. Thus, the descriptions could neither be reused nor merged into an understanding of a software architectural description of the new system.

In DDPP1, a two-part architectural model was identified for the large class of embedded systems that performs a continuous handling of signals in addition to reacting on asynchronous external events that influence and reconfigure the signal handling [Jepsen et al., 1999]. The motor control unit at Danfoss Drives is an example of such a system. A number of design patterns [Gamma et al., 1995] helped in building the architecture. In DDPP2 the application was generalised into a framework

with specialisations in two dimensions: motor control principle and application area. The ultimate goal is a product line architecture.

In BOPP1, a C++ framework supporting active objects (concurrent processes) and monitors has been developed [Caspersen, 1999]. The framework has been implemented in the operating system used by Bang & Olufsen and resulted in a reduction of the code size of more than 50 % compared to earlier versions of the system.

DDPP1, DDPP2, and BOPP1 all used CASE tools to draw package diagrams of the software architecture.

Discussion. Architectural tool support in iterative system development should not be detached from other tools used in the development. This poses special problems for architectural tools since, on the one hand, software architecture is concerned with the highest – most abstract – level of design of the software artifact and thus the hardest to change and, on the other hand, change is a hallmark of iterative software development. The nature of this problem is indicated by the number of very different architecture description languages that exist [Allen, 1997]. This suggests a lack of understanding of and consensus on what software architecture *really* is as opposed to, e.g., software design. Thus, establishing such connections must be the first step towards proper tool support for software architecting in iterative system development.

Some support is present though. A diagram editor may be used to design or to help reverse engineer an architecture, and a UML editor may be used to draw an architecture structure from a (limited) number of views. Also during restructuring phases, tools such as structure editors [Minör, 1994] were helpful.

2.4 Coding

Coding is the bread-and-butter of system development: no system will ever be built without code being written in one form or another. In iterative system development, coding may run in parallel with other activities so that what coding adds to one concern may be in conflict with what other activities add. In most projects, coding and prototyping have been central: tool support for this activity has thus been emphasised.

Experiences. Coding is a modelling as well as a construction activity: in the Dragon Project, e.g., coding contributed to analysis as well as design when implementing visions from problem and solution domain modelling. Conversely, concepts and phenomena created during modelling are only thoroughly understood when re-represented as classes and objects. That conceptual modelling is closely connected to code and diagrams once again stresses the need for a close connection between source code and CASE tools.

Figure 4. Pair programming in the BOPP2 project

Coding is also a collaborative activity. In many projects the team has been co-located and, especially in DIPP2, pair programming has been a widely used technique because a domain expert and an object-oriented expert could work closely together (see Figure 4). This means that tools for managing configurations and creating awareness have been important. For configuration management, e.g., Dragon used CVS, whereas DDPP-1 and DDPP-2 used ClearCase. In the Dragon Project, configuration management was used for frequent updates and commits, while e-mails with manually written log messages were used to generate awareness of the coding activity of other team members. This was, however, not sufficient: the awareness information was often too late or too detailed. Often, responsibilities of team members were overlapping, leading to conflicts that could not be resolved by the configuration management system. The awareness notifications were on an entity level of detail (see "Documenting"). What was needed was often to know if, or how, module structure, components, responsibilities, and other architecture-related entities had changed.

In BOPP2, subgroups worked in parallel on different packages of the system. When finished, packages could then be merged into a common model using reverse engineering using Rhapsody. Earlier in BOPP2, Qualiware had been used to support cooperative work on the model: a hierarchical, client-server locking mechanism meant that several developers could work concurrently on the same model. However, at least in our set-up, speed of operation was a bottleneck.

Collaboration was also distributed in the Dragon Project: part of the development team went on a prototyping trip to Asia while the rest of the team continued development in Denmark. Even though the merging-based scheme of CVS was used, synchronisation during the trip was problematic: the developers in Asia had to merge their developments manually on-site in order to build the next iteration of the prototype, and since they were not able to connect to the central repository, a manual merge had also to be performed when they returned.

Coding is also a maintenance activity. This was two-fold: developers have to understand as well as restructure existing code. In the Dragon Project, the Mjølner structure editor was used. The editor includes semantic hyperlinks so that, e.g., a

double-click on a name application makes the editor jump to the corresponding declaration. Moreover, the syntactic editing facilities made it possible to work on a high level of abstraction. These two facilities were very useful, perhaps even crucial, in the context of large amounts of prototypical code.

Discussion. Integration of configuration management tools and collaborative awareness was a need not met by current tools. To notify other developers of changes, developers in the Dragon Project manually created emails describing changes, but the fact that they had to be created manually meant that they often were delayed or forgotten.

Sometimes there is a need for synchronous collaboration and immediate updates, and at other times asynchronous collaboration via merging is enough as the experience from the Dragon Project shows. Moreover, configuration and awareness entities should also be architecture-level entities. To create and maintain such entities, tool support is needed [Christensen, 1999]. If distributed, asynchronous coding is performed, a configuration management tool that support distributed workspaces may be needed.

Evolution of code and models was rapid and drastic in the Dragon Project. However, evolution is not well supported by current tools: only experimental tools, e.g., support refactoring [Roberts et al., 1997] and only for specific languages and environments. Even so, semantic analyses made by a good compiler may, perhaps in combination with structure-oriented tools, give many of the same possibilities for changing code in experimental ways without changing the semantics of the system.

2.5 Documenting

A COT documentation activity divides documentation into three types: *tutorial documentation*, i.e. introductory material for novices, *rationale documentation*, i.e. discussions of design decisions, and *reference documentation*, i.e. a complete description of the software for experienced users [Bjerring et al., 2000]. The reference documentation is divided into *entity-based reference documentation* and *architecture documentation*. In the iterative development process this should be interpreted as a complete description of the current system regardless of its degree of completeness, ranging from an early analysis model to a fully completed implementation. The entity-based reference documentation is documentation associated with concrete entities in the system. An entity can be a package, class, attribute, method etc. The architecture documentation describes how groups of logically related entities work together, whereas the entity documentation describes the entities one by one. Examples of the elements in architecture documentation are patterns and a diversity of diagrams, e.g., the UML diagrams drawn as a logical view, process view, implementation view, or deployment view.

Experiences. From the pilot projects, our experience regarding documentation has mostly been with the entity-based reference documentation and the architecture documentation.

As opposed to in more traditional waterfall processes, documentation was continuously written in the iterative and incremental development. There were many iterations on the development artifacts as well as of the documentation. Initially, the iterations concerned the analysis and design models but later the iterations included implementation. The documentation needed to be updated after each iteration often at the end of each day. Also, as code changed, artifacts such as sequence diagrams produced from this code (see "Testing") needed to be updated, and the large number of whiteboard photos taken (see "Problem Domain Modelling") needed to be sorted and inserted into the existing documentation.

It was time-consuming to keep all development artifacts together, i.e. to collect them into one coherent document together with explaining text. In the first pilot projects, we had many pieces of paper in no systematic order. In BOPP2, we started collecting the artifacts in working documents with a running number, but there was still not much structure in the documentation. The report generator of Qualiware was a substantial step forward in this respect, as it provided good help in collecting the pieces, i.e. diagrams, model information, and textual descriptions, into one coherent document. The documentation could be updated just by pressing a button at the end of the day. In Qualiware it is possible to define a fairly elaborate report structure with automatic inclusion of diagrams and the descriptions of the entities in the models such as use cases, actor descriptions, and class responsibility descriptions. The automatic inclusion of development artifacts is important because it reduces inconsistency.

Another experience with a report generator was the generator of Rhapsody that (at least at that time) did not support inclusion of diagrams. The diagrams had to be copy/pasted into the documentation and since many diagrams were produced, this manual work was cumbersome and error prone with a risk of inconsistency.

In the Dragon Project the architecture documentation consisted mainly of text-documents presenting the overall architecture and the domain model in the form of a class diagram. The domain model was initially captured using whiteboard snapshots; later it was periodically recreated using reverse engineering. With respect to rationale documentation, observations made by an ethnographer were stored in the form of a multimedia database (DESCRIBEs), which was used as input in the design process. Also, late in the project when a usability expert observed use of the prototype, she used the database as a means of understanding the work conducted in the problem domain.

An overall experience from the projects is that the entity-based reference documentation is the documentation type that can get the highest degree of tool support since it can be generated automatically either from the source code or from the model information created using CASE tools. Structured reports can be generated presenting the packages, classes, attributes, methods etc. together with descriptive text originating from, e.g., comments in the source code. The architecture documentation cannot in the same way be produced just by pressing a button in a tool. Documenting architecture is a creative process. Although reverse engineering can be used to generate diagrams from code, the different diagrams must be carefully selected and, typically, different views of the system must be produced. In addition, the diagrams need to be accompanied by explanatory text. However, once created the diagrams can be automatically inserted in a document together with the explanatory text.

Discussion. Automatic generation of entity-based documentation was largely successful, although inclusion of diagrams was not widely supported. Examples of commercial tools that generate entity-based reference documentation are JavaDoc, BumbleBee, George, and Doc++. These tools furthermore output HTML allowing for easy accessible online documentation.

The optimal tool support for documentation in the context of iterative development is automatic insertion of selected development artifacts into documents produced in standard text processing tools. Rational SoDA (Software Documentation Automation) that adds document generation to Microsoft Word and Adobe FrameMaker and enables extraction of information sources such as Rational Rose, seems to be a candidate.

There are other ways of integrating text documentation with information from a CASE tool or development environment. Modern document architecture technologies make it possible to embed documents or parts of documents from other applications into a document. If the word processor and the CASE tool support Microsoft ActiveX linking/embedding, the diagrams or the part of the source code can, e.g., be linked into a document through a live link that will be updated whenever the source changes. If the document is written in HTML, different scripting technologies like CGI scripts or server-side JavaScripts can be used to insert diagrams or parts of the source code in a Web page. Another solution in a heterogeneous environment is to use open hypermedia [Grønbæk & Trigg, 1999] as an integration technology. Using open hypermedia, very different applications such as word processors, CAD programs, and spreadsheets have been integrated without changing the integrated applications. The Webvise open hypermedia system, e.g., features an embedded scripting engine [Hansen et al., 1999] that may be used to create so-called transclusion links such as the links made in ActiveX linking/embedding. In all cases, additional tool support is desirable.

The development artifacts that are used in the documentation must be mutually consistent. This can be assured by the CASE tools by means of code generation, reverse engineering, roundtrip engineering etc. as described earlier. When the development artifacts are modified in the CASE tool or in the source code, the descriptive text in the corresponding documentation may also need to be updated and vice versa. Therefore, the documentation should not be updated automatically. Instead, the user should be notified that a change has occurred. Hyperlinks between the documentation and the development artifacts can be a means for providing awareness.

2.6 Testing

Throughout an iterative project, there is a need for testing the current results. Testing can be divided into two subactivities: validation and verification. Validation confirms that the right problem is solved whereas verification confirms that the problem is solved in the right way. In domain modelling activities, user or domain expert involvement is used to assure that use domain activities will be adequately supported. In this case, the primary focus is validation. Later, in solution modelling, all artifacts of the development process have to be validated as well as verified – not only the

implementation, but also the documentation, the models and other artifacts created in the project. In an iterative approach, testing is an ongoing process. At the end of each iteration, testing assures that the iteration goal has been accomplished.

Experiences. Several different approaches were used to test the results of the pilot projects. Because traditional testing activities were not the main focus here, the software developed in the projects was not formally tested, but a black-box acceptance test was conducted in all projects. The test coverage, however, was not complete; it covered only normal scenarios. Later research in the COT project has resulted in a technical report on systematic testing approaches [Caspersen et al., 2000].

Though testing was not the main focus, testing activities was done at three levels: user level, model level and code level.

User level. In the pilot projects, the developed object-oriented code was integrated with existing, non-object-oriented legacy code. In four projects (BOPP1+2, DDPP1+2) it ran on the target platform and in the fifth (DIPP2) it ran in a simulated environment. Black-box testing was in all four projects possible by stimulating the system using the existing (or simulated) user interfaces and then monitoring the reactions of the system on the user interface and target hardware. This was used to test whether the software performed correctly in normal situations.

Model level. Rather than a direct verification of the object-oriented models, there was mostly an indirect verification: the code was verified and then afterwards reverse engineered into the object-oriented models.

In BOPP2, however, Rhapsody was used for direct verification of the object-oriented models by simulation or execution of the models. The verification capabilities of the tool consist of “real-time” animated statechart diagrams and sequence diagrams. In UML, a class may have statechart attached, which is used to describe the behaviour of instances of the class. In order to use the simulation capabilities, method calls must be added to the statecharts, and the bodies of these methods must be implemented.

When running in debugging mode, statecharts are animated in separate windows, one for each class instance, and the statecharts are updated with the current states and last invoked transitions. Stepping through common scenarios to inspect the behaviour in the statecharts is a verifying as well as a modelling activity. Animation of behaviour via sequence diagrams is also a feature in Rhapsody. It is possible to place objects in the top row of a sequence diagram. When the simulation then executes, messages with origin or destination in one of the shown objects are added to the sequence diagram. These activities can thus help to better understand the problem and make further progress by refining the statecharts.

The behaviour of the animated state charts and the resulting sequence diagrams were driven by a simple graphical user interface created in Visual C++. This worked well and resulted in a quickly growing understanding of the model and the related code. It also resulted in fast turn-around cycles. The generated code uses a Rhapsody-specific framework that makes it fairly easy to adapt the program to different platforms. This allows for development and test on one platform and deployment on

another. After the verification described above, the system was ported to the target platform and it was working successfully almost instantaneously on top of the framework. Further experiences using the Rhapsody tool and especially adapting it to a specific target platform is described in another COT technical report [Andersen, 2000a].

Code level. The traditional method of testing and verifying from a code perspective, i.e. debugging, tracing, and single stepping, was performed by means of the debugging environments used by the different companies. Prior to using Rhapsody, a tool that generated sequence diagrams based on program executions instead of models was developed and used in some of the pilot projects [Andersen, 2000b]. As described earlier we used informal drawings, UML class diagrams and UML sequence diagrams in the problem domain and early solution domain modelling activities. Sequence diagrams were at this time our main tool for illustrating behaviour, but there was no easy way to verify that the original sequence diagrams drawn on whiteboards corresponded to the behaviour of the implemented system (and vice-versa). This "missing link" led to the development of a tool that could profile running Java programs, and from this was capable of displaying messages between objects in the programs (Figure 5). A later extension made it possible to record scenarios from running C++ programs, which could then be shown subsequently.

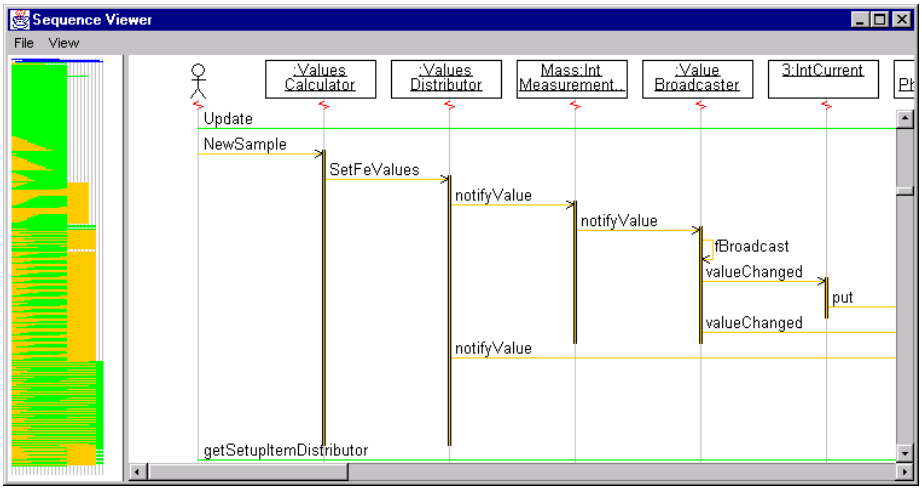


Figure 5. A tool for generating sequence diagrams from program executions

After development, the tool was successfully used in the last three pilot projects to document the behaviour of the system. This documentation was used to make a manual verification of the system. When testing a reactive system the tool worked well, whereas it was not useful for continuos systems in which periodical activation resulted in a very large number of messages. This made it clear that further work was needed to provide overview, that is to find ways of grouping objects, avoiding certain messages in the diagrams and other elimination of unimportant information at a given abstraction level. The left pane in the screen dump in Figure 5 illustrates one solution

to get an overview of a large number of messages. We believe that animated sequence diagrams are very helpful in testing and debugging. Future work will try to create more effective sequence diagramming tools for this purpose both as stand-alone tools and as integrated parts of debuggers.

Discussion. If statechart diagrams are used to model the intra-object behaviour in object-oriented modelling they can play a large role in the verification activities, e.g., to find test cases for unit testing. Preferably, a CASE tool should be used, but also paper diagrams containing statecharts can be manually debugged through different scenarios, in which transitions are marked when they occur until all transitions have been covered.

State machines play a central role in OO development of systems with concurrent processes controlled by external events. To support a smooth transition between statechart diagrams and programs, a proposal for language support for state machines within an object-oriented language has been developed. The proposal has been implemented in an experimental version of the BETA language [Madsen, 1999]

Sequence diagrams can be used as documentation and verification of correct inter-object behaviour. In iterative development, a tool that is capable of comparing sequence diagrams, e.g., a previous sequence diagram with newly recorded diagram, is useful. This enables automatic verification against the originally modeled sequence diagrams. Finally, reverse engineering of the verified code into object-oriented models can be used as a tool to ensure verified models as well (see "Solution Domain Modelling").

3. Concluding Discussion

An activity is a purposeful human endeavour: it has a goal, a context, and a set of constraints that guide its execution. A concern, on the other hand, is abstract. It is something we care about during system development, not what we do. Realising that activities are what have to be supported is consequential: no tool can support all activities equally well. Consequently, different, heterogeneous tools are needed in system development projects. We have emphasised the ways in which separate, currently available tools can be used in a concerted fashion. In general, however, there is a need to focus on the integration aspects of system development tools. Tools should be able to exchange data, be used overlappingly, and complementary all in a concerted fashion in order to support and influence concrete system development activities.

It is thus the actual activities performed that are important when choosing tools to support development. The focus on concrete activities is also reflected in our approach to tool evaluation: we propose using a set of potential tools in a (pilot) project in order to evaluate the effectiveness of the tools in a specific context.

We have reported a number of experiences with tool support for six central iterative system development activities:

- *Problem domain modelling*. Whiteboards are currently the best tools for collaborative modelling with formal and informal elements. To support the

iterations on the problem domain model, a digital camera in combination with transparencies and posters is currently the most effective solution. The Knight tool, that integrates a whiteboard with a CASE tool, however, seems to be a promising alternative.

- *Solution domain modelling.* During solution domain modelling, the focus shifts constantly between model and code. Therefore it is important to keep the model and code consistent. The optimal CASE tool should do this, but most tools have annoying limitations. The focus on code approach with periodic reverse engineering of models was, however, a successful work-around.
- *Software architecting.* UML diagrams were used to design and capture the architecture. Tools for restructuring are needed.
- *Coding.* It is well-known that a configuration management tool is important. In iterative development it becomes crucial. In addition, awareness of changes should be further supported.
- *Documentation.* It should be easy to capture documentation and development artefacts. Preferably all, but at least entity documentation, should be automatically generated from models and code.
- *Testing.* Automatic generation of sequence diagrams from program executions is very valuable both for validation against the initially modelled sequence diagrams and for verification and debugging purposes. Verification of the model through animation is a powerful technique.

A concrete CASE tool may be rejected because of some detailed or minor technicalities even though the tool might be able to support more important activities. In other words: the requirements to CASE tools should be prioritised according to the activities that are most important in the concrete development process. One crucial requirement might be support for round-trip engineering by CASE tools if they are to support an iterative, incremental object-oriented development process. Our main lesson is that tools cannot support all activities equally well and in any realistic development project, one has to work with a plethora of tools that work less than optimally together.

4. Acknowledgements

The work described in this paper was carried out in the Centre for Object Technology that has been partially funded by the Danish National Centre for IT Research [CIT, 2000].

5. References

[Allen, 1997] Allen, R.J. *A Formal Approach to Software Architecture.* Ph.D. Thesis. CMU-CS-97-144, Computer Science Department, School of Computer Science, Carnegie Mellon University.

[Andersen, 2000a] Andersen, C.J. Instantiating the Rhapsody OXF Framework for the Bang & Olufsen Apos+ Operation System. COT Report COT/2-28.

[Andersen, 2000b] Andersen, C.J. *UML Sequence Diagram Generator*. COT Report COT/2-27.

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

[Beck, 1999] Beck, K. Extreme Programming Explained. Embrace Change. Addison-Wesley, 1999.

[Bjerring et al., 2000] Bjerring, C., Hansen, F., Hansen, F.O., Kammeyer, O., Madsen, O.L., Sandvad, E., Skov, S.H., Østerbye, K. *Documentation of OO Systems and Frameworks.* COT Report COT/2-42.

[Booch et al., 1999] Booch, G., Jacobson, I., Rumbaugh, J. *The Unified Modeling Language User Guide.* Addison-Wesley.

[Caspersen et al., 2000] *Test of Object Oriented Systems*. COT Report COT/2-43.

[Caspersen, 1999] Caspersen, M.E., A C++ Framework for Active Objects in Embedded Real-Time Systems bridging the gap between modeling and implementation. In *Proceedings of TOOLS-Pacific'99 (TOOLS 32)*, Melbourne, Australia, November.

[Christensen et al., 1998] Christensen, M., Crabtree, A., Damm, C.H., Hansen, K.M., Madsen, O.L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., Thomsen, M.: The M.A.D. Experience: Multiperspective Application Development in evolutionary prototyping. In *Proceedings of ECOOP'98*, Bruxelles, Belgium, July 1998, Springer-Verlag, LNCS series, volume 1445.

[Christensen et al., 1999] Christensen, M., Damm, C.H., Hansen, K.M., Sandvad, E., Thomsen, M.: Design and Evolution of Software Architecture in Practice. In *Proceedings of TOOLS-Pacific'99* (TOOLS 32), Melbourne, Australia, November.

[Christensen, 1999] Christensen, H.C. The Ragnarok Architectural Configuration Management Model. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*.

[Damm et al., 2000a] Damm, C.H., Hansen, K.M., Thomsen, M., Tool Support for Object-Oriented Cooperative Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of CHI'2000*.

[Damm et al., 2000b] Damm, C.H., Hansen, K.M., Thomsen, M., Tyrsted, M. Creative Object-Oriented Modelling: Support for Creativity, Flexibility, and Collaboration in CASE Tools. In *Proceedings of ECOOP'2000.*

[Damm et al., 2000c] Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Tool Integration: Experiences and Issues in Using XMI and Component Technology. In *Proceedings of TOOLS Europe'2000*.

[Foote & Yoder, 1997] Foote, B., Yoder, J.W. Big Ball of Mud. In *Proceedings of the Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97)*. Monticello, Illinois, September. Technical Report #WUCS-97-34 (PLoP '97/EuroPLoP '97), September, Department of Computer Science, Washington University.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns. Elements of Reusable Object/Oriented Software*, Addison-Wesley.

[Grønbæk & Trigg, 1999] Grønbæk, K., Trigg, R.H. *From Web to Workplace*, MIT Press.

[Grønbæk et al., 1997] Grønbæk, K., Kyng, M., Mogensen, P. Toward a Cooperative Experimental System Development Approach. In Kyng, M., Mathiassen, L. (Eds.) *Computers and Design in Context*. The MIT Press.

[Hansen et al., 1999] Hansen, K.M., Yndigegn, C., Grønbæk, K. Dynamic Use of Digital Library Material – Supporting Users with Typed Links in Open Hypermedia. In Abiteboul, S., Vercoustre A.-M. (Eds.) *Research and Advanced Technology for Digtial Libraries*. Proceedings of the Third European Conference, ECDL'99, Paris, France. LNCS, vol. 1696, Springer-Verlag, pp. 254-273.

[ISO, 1995] *Information Technology for the Evaluation and Selection of CASE Tools.* ISO/IEC 14102.

[Jarzabek & Huang, 1998] Jarzabek, S., and Huang, R. The Case for User-Centered CASE Tools. *Communications of the ACM*, 41(8).

[Jepsen et al, 1999] Jepsen, H.P., Hansen F.O.: Designing Event-Controlled Continuous Processing Systems. *In Proceedings of the Embedded System Conference Europe*, November 16-18, Maastricht.

[Kruchten, 1995] Kruchten, P. The "4+1" View Model of Software Architecture. *IEEE Software*, Vol. 12, No. 6, November.

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

[Madsen, 1999] Madsen, O.L, Towards Integration of State Machines and Object-Oriented Languages. In *Proceedings of TOOLS EUROPE'99, Technology of Object-Oriented Languages and Systems 29th International Conference* and Exhibition Palais des Congres, Nancy, France, June 7-10.

[Minör, 1994] Minör, S. Editing. In Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B. (Eds.) *Object-Oriented Environments. The Mjølner Approach.* Prentice Hall.

[Roberts et al., 1999] Roberts, D., Brant, J., Johnson, R. A Refactoring Tool for Smalltalk, *Journal of Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, pp. 253-263.

[Stapleton, 1995] Stapleton, J. The Dynamic Systems Development Method Manual v2.0. The DSDM Consortium, 1995, `http://www.dsdm.org`

5.1 Online References

[CIT, 2000] `http://www.cit.dk`

[COT, 1999] `http://www.cit.dk/COT`

[Dragon, 2000] `http://www.cit.dk/COT/case5-eng.html`

[Embedded, 1999] `http://www.cit.dk/COT/case2-eng.html`

[UML, 2000] `http://www.omg.org` and `http://www.rational.com`

5.2 Tools and Technologies

[ActiveX] `http://www.microsoft.com/com/tech/ActiveX.asp`

[BumbleBee] `http://www.bbeesoft.com`

[ClearCase] `http://www.rational.com/products/clearcase/`

[CVS] `http://www.cyclic.com/cvs`

[Doc++] `http://www.rsdi.com/doc++/`

[GDPro] `http://www.gdpro.com`

[George] `http://www.k2.co.uk`

[JavaDoc] `http://java.sun.com/products/jdk/javadoc/`

[Mimio] `http://www.virtual-ink.com`

[Mjølner] `http://www.mjolner.com`

[QualiWare] `http://www.qualiware.dk`

[Rational Rose] `http://www.rational.com/product/rose`

[Rhapsody] `http://www.ilogix.com`

[SMART Board] `http://www.smarttech.com`

[SoDA] `http://www.rational.com/products/soda`

[Visio] `http://www.visio.com`

[WithClass]: `http://www.microgold.com`

To appear in Proceedings of ITiCSE'2002

Tool Support for Collaborative Teaching and Learning Object-Oriented Modelling

Klaus Marius Hansen
Department of Computer Science
University of Aarhus
Aabogade 34, 8200 Aarhus N, Denmark
marius@daimi.au.dk

Anne Vinter Ratzer

Ideogramic ApS
Mejlgade 45, 8000 Aarhus C, Denmark
avratzer@ideogramic.com

ABSTRACT
Modelling is central to doing and learning object-oriented development. We present a new tool, Ideogramic UML, for gesture-based collaborative modelling in the Unified Modelling Language (UML) which can be used to collaboratively teach and learn modelling. Furthermore, we discuss how we have effectively used Ideogramic UML to teach object-oriented modelling and the UML to groups of students using the UML for project assignments.

INTRODUCTION
Simplistically, object-oriented software development is a mapping of real-world phenomena and concepts of a problem domain to objects, classes, and their relationships in a solution domain [12]. The process of doing so is called *modelling*. Even though the reality of object-oriented development is much more complex, modelling enjoys a central position in object-orientation and is thus also important from a teaching perspective [12][13].

The Unified Modeling Language (UML; [15]) is the predominant industry standard for modelling and it is thus relevant to introduce it to students when teaching object-oriented software development. The UML has, however, a number of characteristics that inhibit learning:

- *Extent.* The UML has nine diagram types, more than 100 distinct meta-classes, and even more relationships between these. Many of these are necessary to know in order to use the UML fully.
- *Complexity.* The many elements of the UML have complex semantics and there is no concept of levels of use. In this way, the UML has a steep learning curve and is primarily suitable for modelling experts.
- *Usability.* Even though the UML is large and complex, it has no support for many commonly occurring modelling practices [8]. The UML provides one formal view of what a model is and this view cannot be changed to accommodate a modelling situation.

In an experiment to find ways of dealing with these characteristics, we used a new tool, *Ideogramic UML,* to introduce the UML to a group of second year computer science undergraduates taking a course in object-oriented programming. Ideogramic UML is a gesture-based tool for collaborative modelling which is particularly effective on large-scale input and output devices such as electronic whiteboards.

The rest of this paper reports from the observations from this experiment and discusses Ideogramic UML as a vehicle for teaching and learning object-oriented modelling.

BACKGROUND
Object-oriented techniques have been criticized as being too complex for teaching in certain situations [2][16]. [16] states that a notation such as the UML is excellent for experts, but bad for novices, since it contains a large number of new concepts, and novices need to learn both a new way of thinking, and a "highly non-intuitive graphic notation". Several possibilities exist for avoiding the complexity such as *hand waving about difficult details*, *providing a framework to cover difficult details*, or *avoid difficult details* [11]. With the “OVAL” notation, [16] provides an example of the second possibility in the context of graphical modelling notations.

In some cases, such as ours, object-orientation is introduced through object-oriented programming so that the basic concepts and principles of object-orientation are learned *before* the graphical notations [13]. This makes it possible to introduce the UML directly in connection to a known object-oriented language by relating concepts of the language to concepts of the UML. As we investigate in this paper, a tool such as Ideogramic UML that does not in itself have a steep learning curve can further help to introduce students effectively to the UML.

Teaching situations provide a rich setting in which collaboration and communication co-exists in a number of forms. Different environments have tried to support these activities through "electronic classrooms" [1][2]. The "DEBBIE" groupware system [2], allows students to make

personal annotations on a graphics tablet to an instructor's notes written on an electronic whiteboard. Our goal with using Ideogramic UML in a learning environment complements the goals of such systems: we envision the tool as a part of an *electronic learning environment* in which the tool can be used in many, different situations. Particularly, Ideogramic UML may be an effective way to engage students in *active learning* [14].

IDEOGRAMIC UML: TOOL SUPPORT FOR COLLABORATIVE LEARNING

We have based *Ideogramic UML* on a series of user studies and subsequent evaluations of UML modelling situations. The tool is designed and implemented to support direct and fluid interaction, cooperative work, and an integration of formal and informal model elements [6][7].

User Interface

Ideogramic UML runs on a variety of input devices ranging from graphics tablets over ordinary desktop PCs to electronic whiteboards. The electronic whiteboard version provides a natural support for collaborative work (Figure 1)

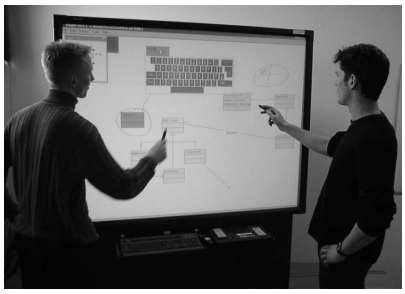


Figure 1. Collaborative modelling in Ideogramic UML

The user interface (Figure 2) is simple: it is a plain white surface, much like a traditional whiteboard, on which users draw gestures using dry pens.

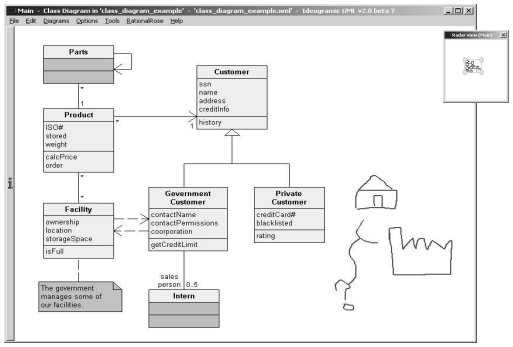


Figure 2. The Ideogramic UML user interface

The use of traditional interface elements such as dialog boxes, buttons, and toolbars is kept to a minimum. When the user draws a gesture, Ideogramic UML recognizes the gesture on-the-fly and translates it into formal UML objects. Figure 3 shows how a box gesture (left) drawn in Ideogramic UML is transformed into a UML class (right).

Figure 3. Incremental recognition and transformation of gestures

The gestures largely resemble simple drawings on a whiteboard so that gestures are easier to learn and use. Apart from learnability, gestures allow input directly on the workspace, are fast to draw, and have a low cognitive overhead.

Context-sensitive, hierarchical pie menus (Figure 4; [9]) can be opened by pressing and holding the pen. They provide a ready-at-hand way of invoking less frequently used commands.

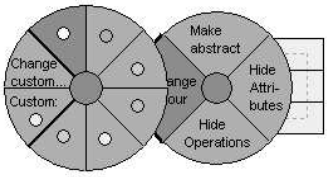


Figure 4. Context-sensitive, hierarchical pie menus

Moreover, drawing a short gesture in the direction of a command on a pie menu invokes that command [10].

Scenarios of Use

Ideogramic UML can be used in several ways for education. In this paper, the following four scenarios are of particular interest:

1. *Lecture use.* A lecturer can explain UML and object-oriented modelling techniques using Ideogramic UML in the classroom. As on an ordinary whiteboard there is a close connection between the lecturer's actions and the results of these actions.
2. *Peer instruction use.* Teaching assistants and students may use Ideogramic UML to present and develop solutions interactively. Teaching assistants may, e.g., present and review concepts from lectures and students may present and develop solutions to exercises.
3. *Peer group use.* Students may use Ideogramic UML when working in project groups, solving an exercise collaboratively. The use of Ideogramic UML can complement the use of ordinary whiteboards and blackboards and provide a more direct integration of modelling practices and computerized models.
4. *Individual use.* Ideogramic UML can be used on an ordinary PC as a modelling tool combined with all

the scenarios above. Models may be transferred and used in the scenarios above.

In all scenarios the strength of Ideogramic UML is its directness of use: the ability to express UML models in a familiar and powerful interactive way. The next section explores these aspects in the context of concrete use in learning situations akin to scenario 2. and 3.

STUDIES OF USE

Experiment background

The context of our usability studies was a project assignment in a course on object-oriented programming for second-year computer science students at the University of Aarhus. During this four-week project, students in groups of two or three created an object-oriented application for archiving, browsing and replaying music files and CDs. Part of the assignment was to construct object-oriented models of the problem domain.

All participants in our study had attended a one-hour lecture in connection with the course, in which we presented a subset of the UML notation, and showed a short demonstration video of Ideogramic UML. This lecture covered the basic UML concepts needed for the project assignment requirements.

The table below shows how often the participants had used the UML and other techniques relevant to the usability test:

	Programming	OO	UML	Gestures
Never	0%	0%	58%	42%
A few times	25%	17%	33%	42%
Some	33%	58%	8%	N/A
Frequent	42%	25%	0%	17%

Table 1. Background information for test subjects

The most commonly tried type of gesture input device was a PDA such as the Palm™ handheld computer.

Execution

The study started with a summary of the lecture given in the course. We then had a brief trial round, where the participants became familiar with the more uncommon interaction techniques: gestures and pie menus.

The groups then used Ideogramic UML to construct and discuss the models they needed for the project assignment. We did not specify any explicit tasks, but left it to the students to decide what to model and how. In a normal study group meeting, the groups would solve the assignment by constructing and discussing models around a whiteboard in groups, followed by implementation of the suggestions either in groups or individually. The setup of our usability study thus came close to a typical situation at the beginning of an assignment.

During the study, two experimenters were present, one responsible for the execution of the study and observation, and one responsible for providing help to the participants. All experimenters took notes, and we videotaped one of the groups. In our observations, we focused primarily on aspects such as breakdowns and focus shifts [4], both in use of the tool and in use of the UML.

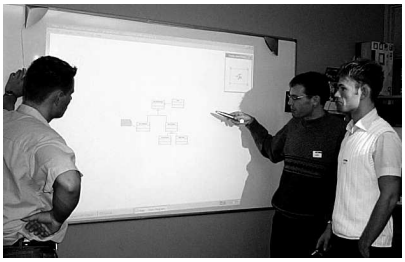


Figure 5. Student use of Ideogramic UML

Help

During the test, the participants had access to two types of help.

- *Passive.* Two print-outs showing the gestures available in the tool, and the basic elements of the UML syntax. The help menu in Ideogramic UML was also available.
- *Active.* The active experimenter provided *learning scaffolding* [5][17] in the form of tool-specific help and modelling-specific help. The first category, e.g., included help for drawing a gesture or using the hardware. The second category, e.g., included help when they specifically asked for help on the UML, or when they used UML syntax inconsistently with their voiced intentions.

To get an impression of how novice users would react to the tool if they used it with no help, e.g. after downloading an evaluation copy from our website, we gave very limited tool-specific help to one of the groups, only explaining to them how to use the pens on the whiteboard. They received the same amount of UML-specific help as the other groups.

The participants used the tool for approximately two hours. We then did an open-ended interview focusing on their qualitative impression of the tool.

The next section summarizes observations from the experiment and comments from the interviews.

Results

Based in observations and analysis of our use studies, we will now discuss how Ideogramic UML addresses some of the problems with teaching and learning object-oriented modelling using the UML.

Extent

Using a gesture-based tool for teaching UML helps the teacher limit the amount of functionality presented to novice users. Through an instructor or a manual, the users can learn a small set of gestures necessary for their tasks and become

familiar with these before moving on. Unlike a tool based on toolbars and pull-down menus, the users do not constantly have to face and ignore other, less relevant options for interaction. To ensure that the novice users can remember the gestures for their subset of commands, the gestures should resemble as closely as possible the objects they create. This is generally the case in Ideogramic UML, as shown for the class-gesture in Figure 3.

In the user study, we observed that the participants had no problems remembering the most basic gestures, and when one member of a group had problems with a gesture, the others could in most cases remind him how it should be drawn. In the interviews, the groups supported this observation, saying that they found the tool easy to use and easy to get started on.

Complexity

Ideogramic UML reduces some of the complexity of UML, as described above, by presenting fewer possibilities to novice users at a given time. The context-sensitive pie menus, showing the commands possible for a specific UML object, helps novice users understand the structure of UML. The same is true for UML specific feedback such as automatic layout of object hierarchies, which shows the user how UML objects are related and linked to each other.

Some tools similar to Ideogramic UML have additional software engineering capabilities such as the ability to keep models and code synchronized. This can help users transfer their models to code, but several of our test users commented on this, saying that they preferred Ideogramic UML because it did not have this functionality. The reason was that they felt safer using a tool that did not change their code automatically, and feeling safer, they also felt more encouraged to explore different variations of models.

Usability

The freehand mode supports drawings that are not in the UML syntax, but that in some way supplements the UML model. In the usability study we saw examples such as drawings of the user interface, sketches of a runtime object structure, and small code drafts.

This is a benefit for both expert and novice users. All users can at any point switch to freehand mode if the UML syntax is too restricted for a specific purpose. Novice users can switch to freehand mode if they are unsure of the exact UML syntax, and discuss their model just as they would on a standard white board. All groups in our study said that they found the freehand mode very important for UML modelling.

Another way Ideogramic UML increases usability of UML modelling, especially for novice users, is by using being less restrictive than the formal UML specification. Ideogramic UML allows various types of incomplete objects such as classes without names, elements with the same name, and disconnected associations. This enables learning by allowing users to start modelling while they still have a very basic understanding of UML syntax.

The figure below shows the model made by one of the groups in our study. The group created it in roughly 1 ½ hour, having no prior experience with UML modelling, and only a very basic introduction to the Ideogramic UML tool. During this time, the group sketched and discussed aspects of the user interface, examples of runtime object models, as well as several suggestions for code architecture. (Figure 5)

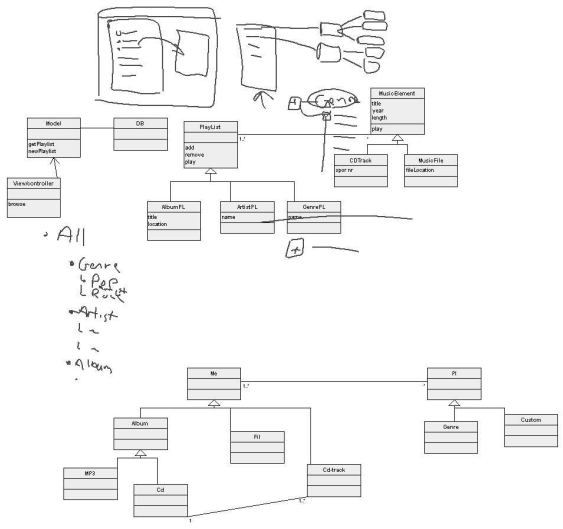


Figure 6. Model made by students in Ideogramic UML

SUMMARY AND FURTHER WORK

As the discussions above show, we have successfully applied Ideogramic UML to teaching and learning UML and object-oriented modelling

- based on an object-oriented language
- through collaborative peer work
- through vertical and sequential active scaffolding

Ideogramic UML helps in this through a direct and fluid interaction that supports initial use and a gradual revelation of more advanced UML use through gestures. The user interface supports a collaborative style of use effective for teaching and learning.

Some of the major future directions for Ideogramic UML that our studies pointed to with respect to teaching and learning are

- *Support for understanding object-oriented design processes.* Active, adaptive templates of, e.g., design patterns will provide a first means of doing this.
- *Support for iterative learning of the UML.* The initial means for this will be various form of context-sensitive help on UML design preferably coupled to the support suggested above.

Obviously, further studies of the practical, situated use of Ideogramic UML is also on the top of the agenda. Hopefully, a new flexible working environment for the students at our department will provide means for students to flexibly use Ideogramic UML personally or in group settings as discussed above. These further studies should also include the study of use of Ideogramic UML for lecture use and for personal use with a particular focus on learning.

Finally, an evaluation version of Ideogramic UML can be downloaded from `http://www.ideogramic.com` where a video demonstrating it in action is also available.

ACKNOWLEDGEMENTS

The work reported in this paper has been funded by the Centre for Object Technology that has been partially funded by the Danish National Centre for IT Research. We also thank the students participating in the usability study.

REFERENCES

1. Abowd, G.D., Atkeson, C., Feinstein, A., Hmelo, C., Kooper, R., Long, S., Sawhney, N., and Tani, M. (1996). Teaching and Learning as Multimedia Authoring: The Classroom 2000 Project. In the *Proceedings of the ACM Multimedia'96 Conference*.
2. Astrachan, O. (2001). OO Overkill: When Simple is Better Than Not. In *Proceedings of the SIGCSE Symposium 2001.*
3. Berque, D., Johnson, D.K., Jovanovic, L. (2001) Teaching Theory of Computation Using Pen-Based Computers and an Electronic Whiteboard. In *Proceedings of ITiCSE'2001*.
4. Bødker, S. (1996). Applying activity theory to video analysis: How to make sense of video data in HCI. In Nardi (ed): *Context and Consciousness*, MIT press, Cambridge 1996.
5. Chalk, P. (2001). Scaffolding Learning in Virtual Environments. In *Proceedings of ITiCSE'2001.*
6. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction* (CHI'2000).
7. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In *Proceedings of ECOOP'2000.*
8. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). On the UML's Support for Modelling Practice. In *Proceedings of UML'2000.*
9. Hopkins, I., Callahan, J., and Weiser, M. (1988). *Pies: Implementation, Evaluation and Application of Circular Menus.* University of Maryland Computer Science Department Technical Report.
10. Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus.* Unpublished Ph.D. Thesis, University of Toronto.
11. Lewis, J. (2000) Myths about object-orientation and its pedagogy. In *Proceedings of the SIGCSE Symposium 2000.*
12. Madsen, O.L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley.
13. Madsen, O.L., Røn, H., Thorup, K.K., and Torgersen, M. (1998). A Conceptual Approach to Teaching Object-Oriented Programming to C Programmers. In *Proceedings of the Educators' Symposium at OOPSLA'98.*
14. McConnell, J.J. (1996). Active learning and its use in Computer Science. In *Proceedings of ITiCSE'1996.*
15. Object Management Group (2000). *OMG Unified Modeling Language Specification. Version 1.3.* Document formal/00-03-01.
16. Raner, M. (2001). Teaching Object-Orientation with the Object Visualization and Annotation Language (OVAL). In *Proceedings of ITiCSE'2001.*
17. Wood, D, Bruner, J. S. and Ross, G. (1976). The Role of Tutoring in Problem Solving. In *Journal of Psychology and Psychiatry,* Vol. 17.