

Linköping Studies in Science and Technology

Dissertation No. 459

Development Environments for Complex Product Models

Olof Johansson



Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden

Linköping 1996

Inside cover page/back page for doublesided printing

DEVELOPMENT ENVIRONMENTS FOR COMPLEX PRODUCT MODELS

by

Olof Johansson

Engineering Databases and Systems Laboratory
Department of Computer and Information Science
Linköping University

Copyright (c) 1996 Olof Johansson, All rights reserved.

ISBN 91-7871-855-4
ISSN 0245-7524

Originally printed in Sweden by triva-tryck ab, Linköping 1996
Electronic version with minor errata corrections, Linköping 2001

DEVELOPMENT ENVIRONMENTS FOR COMPLEX PRODUCT MODELS

by Olof Johansson

Abstract

The complexity in developing high-tech industrial artifacts such as power plants, aircrafts etc. is huge. Typically for these advanced products is that they are hybrids of various technologies and contain several types of engineering models that are related in a complex fashion. For power plant design, there are functional models, mechanical models, electrical models etc. To efficiently meet new demands on environment friendly technology, models of product life cycles and environmental calculations must be brought into the product design stage. The complexity and evolution of software systems for such advanced product models will require new approaches to software engineering and maintenance.

This thesis provides an object-oriented architectural framework, based on a firm theoretical core on which efficient software development environments for complex product modeling systems can be built.

The main feature of the theory presented in the thesis, is that the software engineering models of the engineering application domain (e.g. power plant design) are separated from software implementation technology, and that source code for the basic functionality for object management and user interaction with the objects in the product modeling system is generated automatically from the software engineering models.

This software engineering technique has been successfully used for developing a product modeling system for turbine- and power plant system design at ABB STAL, using state of the art database technology.

When software products of the next generation of engineering database and user interface technology are made commercially available, a product modeling system developed according to the theory presented in the thesis can be re-implemented within a small fraction of the effort invested in developing the first system.

The product modeling system was put into production in 1993. It is now regularly used by about 50 engineers. More than 80 steam and gas turbine plants and several PFBC power plants have been designed using the system.

DEVELOPMENT ENVIRONMENTS FOR COMPLEX PRODUCT MODELS

by Olof Johansson

| | | |
|----------|------------------------------------|----------|
| 1 | Introduction to the Thesis | 1 |
| 1.1 | Product modeling systems | 1 |
| 1.2 | Contributions | 3 |
| 1.3 | Thesis overview and readers' guide | 4 |
| 1.4 | Research method | 5 |
| 1.5 | Motivation for the work | 6 |
| | Acknowledgements | 6 |

| | | |
|---------------|---|-----------|
| Part I | Properties of Knowledge-Based Systems for Engineering | 7 |
| 2 | Introduction to Part I | 9 |
| 2.1 | Knowledge in a knowledge-based system | 9 |
| 2.2 | Challenges | 13 |
| 2.3 | Motivation for the work | 15 |
| 3 | Components of Expertise | 17 |
| 3.1 | Deep versus surface knowledge | 18 |
| 3.2 | Conceptual and pragmatic aspects of expertise | 18 |
| 3.3 | Domain models and deep knowledge | 19 |
| 3.4 | Generic tasks | 20 |
| 3.5 | Problem solving methods | 21 |
| 3.6 | Summary and challenges | 22 |
| 4 | Knowledge-based Systems for Design | 23 |
| 4.1 | Concepts for describing the design process | 24 |
| 4.2 | The components of a design instantiation | 25 |
| 4.3 | Design plans versus iteration | 26 |
| 4.4 | Observations and conclusions | 26 |
| 4.5 | Some challenges for knowledge-based design systems | 28 |
| 5 | The KBS Development Process | 29 |
| 5.1 | A model of the KBS development process | 29 |
| 5.2 | Important qualities for reducing KBS development time | 35 |
| 6 | Observations from the Development of a Knowledge-based Intelligent Front End | 37 |
| 6.1 | The task of an intelligent front end for a CAEsystem | 37 |
| 6.2 | KBS features for building and maintaining an IFE | 38 |
| 6.3 | An IFE for damage tolerance design on aircraft structures | 39 |
| 6.4 | Method | 39 |
| 6.5 | Discussion | 41 |

| | | |
|----------------|--|-----------|
| Part II | Experience from Development Environments for Product Modeling Systems | 45 |
| 7 | Introduction to Part II | 47 |
| 7.1 | Overview | 47 |
| 7.2 | Readers guide | 47 |
| 8 | ProCAD - A Product Modeling System for Power Plant System Design | 49 |
| 8.1 | Introduction | 49 |
| 8.2 | The ProCAD system architecture | 50 |
| 8.3 | Outline of the ProCAD domain model | 51 |
| 8.4 | The plant browser application | 53 |
| 8.5 | The P&ID application | 55 |
| 8.6 | 4GL applications | 58 |
| 8.7 | Measurements for product models | 59 |
| 8.8 | Summary | 59 |
| 9 | Using a Meta Database to Implement Product Modeling Systems | 61 |
| 9.1 | Introduction | 61 |
| 9.2 | Problems while developing complex product modeling systems | 63 |
| 9.3 | A generalized software architecture for PMS | 65 |
| 9.4 | Architecture of a PMS development system | 69 |
| 9.5 | An object-oriented domain model example | 72 |
| 9.6 | Source code generation example | 74 |
| 9.7 | The meta-database domain model | 76 |
| 9.8 | Experience from the ProCAD development | 80 |
| 9.9 | Conclusions | 81 |
| 10 | Source Code Generation from Domain Models | 83 |
| 10.1 | Domain model for a mini meta-database | 83 |
| 10.2 | Declarative 1-step source code generation | 84 |
| 10.3 | Declarative 2-step source code generation | 86 |
| 10.4 | Measurements from the development environment | 88 |
| 10.5 | Summary and conclusions | 91 |

| | | |
|-----------------|--|------------|
| Part III | Theoretical Framework for the Information System Platform | 93 |
| 11 | Introduction to Part III | 95 |
| 12 | Concepts and Notation from Infological Theory | 97 |
| | 12.1 A quantifiable infological framework | 97 |
| | 12.2 Data and information | 98 |
| | 12.3 Infological model | 99 |
| | 12.4 Infological object system concepts | 100 |
| | 12.5 Information entities | 102 |
| | 12.6 Transmission oriented information theory | 105 |
| 13 | A Design Space Spanning Benchmark Domain Model | 107 |
| | 13.1 Object model diagram of the benchmark domain model | 107 |
| | 13.2 Overview of the benchmark domain model | 109 |
| 14 | Primitives for Domain Models | 113 |
| | 14.1 Class | 114 |
| | 14.2 Attributes | 117 |
| | 14.3 Relationships | 119 |
| | 14.4 Relationship operations | 121 |
| | 14.5 1-N & M-N Relationship operations | 124 |
| | 14.6 Higher degree relationships | 126 |
| | 14.7 Cardinality calculation examples | 127 |
| 15 | An Information-oriented Task Description Language | 129 |
| | 15.1 Some hypothetical measures on KBS user interfaces | 129 |
| | 15.2 Object model of the task description concepts | 135 |
| | 15.3 UserRole | 136 |
| | 15.4 Task | 136 |
| | 15.5 UserTransaction | 136 |
| | 15.6 InfoSet | 137 |
| | 15.7 Elementary constellation type role, ECTRole | 137 |
| | 15.8 Attribute e-constellation type role AECTRole | 138 |
| | 15.9 Relationship e-constellation type role RECTRole | 139 |
| | 15.10 Summary | 139 |
| 16 | The Benchmark Task Descriptions | 141 |
| | 16.1 ExecutiveDirector tasks | 141 |
| | 16.2 ChiefDesigner tasks | 141 |
| | 16.3 Project Manager tasks | 142 |
| | 16.4 Designer tasks | 142 |
| | 16.5 Specifying a functionality feature checklist | 144 |

| | | |
|----------------|--|------------|
| Part IV | Theoretical Framework for Object-Oriented User Interface Configurations | 145 |
| 17 | Introduction to Part IV | 147 |
| 18 | The User Interface Software Architecture | 149 |
| | 18.1 Introduction | 149 |
| | 18.2 Components of the UISA | 151 |
| | 18.3 2D user interface support | 156 |
| 19 | The User Interface Managers | 159 |
| | 19.1 The display manager | 159 |
| | 19.2 The selection manager | 160 |
| | 19.3 The clipboard manager | 161 |
| | 19.4 The transaction manager | 161 |
| 20 | Discussion | 163 |
| | 20.1 Prediction properties of the domain modeling language | 163 |
| | 20.2 Other object-oriented domain modeling languages | 164 |
| | 20.3 Other work on user interfaces | 165 |
| | 20.4 Meta-CASE-tools | 167 |
| | 20.5 Future work | 168 |
| 21 | Conclusions | 169 |
| | References | 171 |

| | |
|---|------------|
| Appendixes | 181 |
| Appendix A The Benchmark Domain Model | 182 |
| Appendix B Benchmark Application Task Descriptions | 190 |
| Appendix C User Interface Object Characteristics | 191 |
| C.1 Information recorded in a log record | 191 |
| C.2 Logging begin and end of user transactions | 193 |
| C.3 Window types | 194 |
| C.4 Object editor types | 198 |
| C.5 Attribute editor | 201 |
| C.6 Relationship1Editor | 203 |
| C.7 RelationshipNEditor | 207 |
| C.8 Relationship and object link editors | 209 |
| C.9 Views | 213 |
| Appendix D Information Processing Model of a User | 215 |
| Appendix E Model of Human Mental Object Models | 218 |
| E.1 Chunks | 218 |
| E.2 Learning, or storing chunks in long term memory | 219 |
| E.3 Recalling of chunks | 219 |
| E.4 The Soft Semantic Network Hypothesis | 220 |
| E.5 Connection between KB, UI and SSN | 220 |
| E.6 Mapping the SSN to the KB | 221 |
| E.7 Reservation against the concept ofSSN | 221 |
| E.8 Conclusions | 222 |
| Appendix F User Interaction Example | 223 |
| Appendix G Sample OO-model of a Car Braking System | 224 |
| G.1 A structural view (71 e-messages) | 224 |
| G.2 A classical knowledge-base browser view | 225 |
| G.3 Topological index views | 226 |
| G.4 A part-of view (30 e-messages) | 227 |
| G.5 A simple table view (30 e-messages) | 227 |
| G.6 A PERT-diagram view. | 227 |
| G.7 An elementary constellation view (71 e-messages) | 228 |
| G.8 Conclusion | 230 |
| Index | 231 |
| Index of Abbreviations | 235 |
| List of Tables | 237 |
| List of Figures | 238 |
| Pages for Notes and Comments | 240 |

Corrected errata, comments and minor improvements from the original printed version

- p 72 Section 9.5 An object-oriented domain model example
This section has been adapted to describe an efficient subset of the industri standard UML notation for class diagrams¹. One handy user defined compartment named <<superclass>> is added between the <<name>> and <<attribute>> compartments. Instead of taking up a whole text line with <<superclass>> that is the UML-practice for labeling a customized compartment, a small inheritance symbol "⬆", is preceding the contained superclass name(s) to save space. This added compartment significantly eases the layout and readability of the UML class diagrams.
- p 82 FIGURE 21. An objectmodel diagram for the meta-database.
Changed to UML class diagram notation.
- p 84 FIGURE 22. A subset of the EER-model for the meta-database.
Changed to UML class diagram notation.
- p 106 (EQ 10) a mistakenly included term of +1 has been removed.
- p 108 FIGURE 31. The benchmark domain model.
Changed to UML class diagram notation.
- p 135 FIGURE 39. Object model of the task description language.
This model has now several improved successors, but still illustrates the main ideas. The diagram notation is from [Sundgren 73], where 1-N relationships are shown with a fork entering the N-side class.
- p 152 FIGURE 42. Domain model of user interface state.
Changed to UML class diagram notation.
- p 183 FIGURE 44. The benchmark domain model.
Changed to UML class diagram notation.
- p 216 (EQ 43) a mistakenly included term of +1 has been removed.

1. OMG Unified Modeling Language Specification version 1.4, September 2001, Section 3.19 Class Diagram, <http://www.omg.org/cgi-bin/doc?formal/01-09-67>

1 Introduction to the Thesis

The complexity in developing high-tech industrial artifacts such as power plants or aircrafts is huge. Typical for these advanced products is that they are hybrids of various technologies and contain several types of engineering models that are related in a complex fashion. For power plant design there are functional models, mechanical models, electrical models etc.

This thesis provides an object-oriented architectural framework, based on a firm theoretical core on which efficient software development environments for complex product modeling systems can be built.

The main feature of the theory presented in the thesis is that the software engineering models of the engineering application domain (e.g. power plant design) are separated from software implementation technology, and that source code for the basic functionality for object management and user interaction with the objects in the product modeling system is generated automatically from the software engineering models.

This software engineering technique has been successfully used for developing a product modeling system for turbine- and power plant system design, using state of the art database technology. See chapter 8.

When products of the next generation of engineering database and user interface technology are made commercially available, a product modeling system developed according to the theory presented in the thesis can be re-implemented with only a small fraction of the effort invested in developing the first system.

1.1 Product modeling systems

A product modeling system (PMS) is a computer-integrated development environment for a specific class of advanced products. A PMS consists of a product model database which is interfaced with CAD¹-applications that support graphical designs of engineering models, graphical user interfaces for browsing and modification of the object structures in the product model, and CAE²-applications that make engineering calculations on the models.

Figure 1 shows the approach taken to manage the software engineering of

1. Computer Aided Design
2. Computer Aided Engineering

product modeling systems. The idea is to maintain a high-level PMS design specification in the form of an object-oriented CASE³ model in a meta-database.

The OOCASE model is developed in cooperation with product-, CAD-, and CAE-application experts.

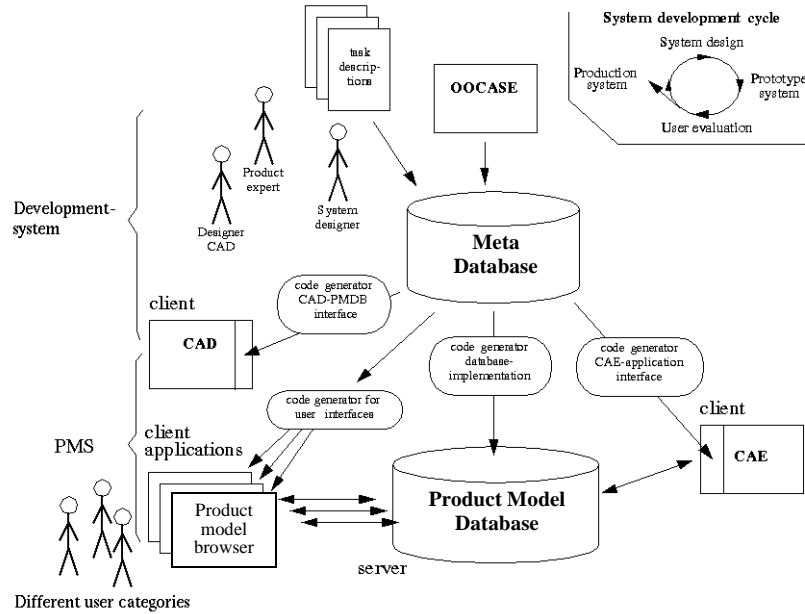


FIGURE 1. Software development approach for product modeling systems.

Most of the source code for the PMS implementation is generated automatically, using SQL-based source code generators. Our development platform generates database schemas with stored procedures and triggers that provide a high level interface for application program interaction with product models. It also generates browser applications for form-based interaction with product model data, and interface modules in the native application development language of a CAD-system. Through these, a CAD application developer has access to the product models in the database on an abstraction level that is natural for an engineer.

By automatic generation of most of the surrounding OOCASE-model dependent software, changes in the model can quickly be implemented in a prototype system and evaluated by experts and end users.

1.2 Contributions

The main contribution of the thesis is the object-oriented architectural framework. This integrates various types of software, software development methods and different professional roles into a coherent whole, which enables efficient development and long term maintenance of complex product models. The framework addresses both development of single instances of product models and the evolution of their representations, due to changing business and technical requirements.

On the way to this contribution a few others have been achieved:

1) The integration of state of the art software engineering technology with a theory that enables explicit numerical quantification of information content in large and complex product models. This theory integration enables product models from different engineering domains to be compared by the numerical information quantity they contain. Information quantity measurements can be used for prediction and planning. Product models that are constructed with a formal language defined with the concepts presented in Part III of the thesis can be compared and related to the resources needed for their engineering. The theoretical core for the information systems design is taken from [Sundgren 73]. His framework has been extended with a multiple inheritance class hierarchy, distinct modeling of part-of and refers-to relationships, and support for datatype independent attribute declarations. The object-oriented CASE-models have a meta-model according to our meta-database (Figure 21 on page 82). This meta-model has simple mappings to EER⁴-models, an infological model (OPR⁵-approach) [Sundgren 73][Sundgren 89], the entity-relationship model [Chen 76], an object-oriented analysis models [Coad&Yourdon 90], OMT⁶ [Rumbaugh 91] and EXPRESS [EXPRESS 88].

2) The meta-database contains an information-oriented language for task description that covers responsibility areas for different types of information in a product model for different user roles. The formalisation of both the concepts for object modeling, and the description of user tasks enables development of user interface compilers for advanced graphical- and form-based browsers which can be used directly in a PMS production environment. The information-oriented task description language (section 15.2 on page 135), is mainly based on experience gained when using the ideas presented in [Johansson 91] for developing the product modeling system for steam turbine and power plant system design [Johansson 94]. Some additional motivation for introducing the concept of roles for elementary constellation types was gained from [Reenskaug 96].

3) The formalisation of the user interface design into a user interface software architecture (UISA). This architecture uncouples the

4. Extended Entity Relationship

5. Object Property Relationship

6. Object Modeling Technique

representation of product models in a database from their presentation- and interaction mechanisms in the user interface. The UISA is applicable for traditional form-based user interfaces and advanced graphical user interfaces. It provides a user interaction model that visualizes the underlying theory, and enables automatic implementation of functionality for object-based selection-, clipboard- and transaction management when several windows displaying the same information are modified in parallel.

4) The support within the architecture for fast incremental prototype development of large product modeling systems. This is enabled by the clean mapping between theoretical information-oriented concepts in the software engineering models, concepts in the database and concepts in the user interface. The clean mapping simplifies the implementation of SQL-based source code generators. The framework also defines how user interaction with generated user interfaces can be logged automatically, so that statistics from prototype usage in the log can be presented and related to the formal description of the domain model in the meta-database. This kind of statistics can be valuable decision support when deciding how to improve a prototype implementation after an evaluation period.

1.3 Thesis overview and readers' guide

The contributions of this thesis focus on development of complex object-oriented information systems and begin in Part II.

Part I, 'Properties of Knowledge-Based Systems for Engineering' describes concepts from knowledge-based systems development, and **provides a vision** from the early 90's that motivated the development of the theories presented in Parts III and IV. The purpose of Part I is to give an idea of what future knowledge engineering environments for product model based development could be like. The material is taken from [Johansson 91]. **Chapters 2-4 and 6 and are NOT needed to understand the theory presented in the rest of the thesis.** The Chapter 5, "The KBS Development Process" describes the difference between mental models, conceptual models, meta-models, domain models and database implementation models.

Part II, 'Experience from Development Environments for Product Modeling Systems' describes ProCAD, a PMS for turbine and power plant system design and its development platform. These systems were developed during a five-year period in two cooperation projects with ABB STAL. The projects provided a test of the development principles initially described in [Johansson 91] and a refinement of tools and techniques to practical solutions for developing PMS for complex products. Chapter 9 describes the meta-database design of the development platform, and chapter 10 the SQL-based source code generation. **Chapter 9 and especially Section 9.7 "The meta-database domain model" are important for understanding Part III.**

Knowledge gained from the work described in Parts I and II has been generalized and is documented in Parts III and IV.

Part III, 'Theoretical Framework for the Information System Platform' introduces the concepts from infological theory that enable exact numerical quantification of information content in product models. It introduces a benchmark domain model that is used for explanation in the thesis, and can be used by other researchers for comparing performance of different user interface implementations. The final chapters describe the information oriented task description language, and provide examples of task descriptions from the benchmark domain model.

Part IV, 'Theoretical Framework for Object-Oriented User Interface Configurations' introduces the user interface software architecture, which has been successfully used for generation of domain model-specific user interfaces. Together with Appendix C, "User Interface Object Characteristics", the part provides a common framework that enables performance comparison of different user interface implementations.

1.4 Research method

This research was conducted according to the following method:

Part I) Study of the research area and identification of the research topic.

Part II) Practice with development of the system under study. Development of a practical solution to the research topic, and measurements of its performance.

Parts III, IV) Formulation of a theory that enables a more precise comparison of the systems under study.

The topic under study is efficient development environments for complex product models. In the future, development environments and product modeling systems of a similar performance as the one presented here should probably be developed in-house within commercial companies.

There is a need for further standardization efforts, which must be guided by advanced prototype implementations to become competitive.

I believe that tools based on the theoretical framework presented here and future derived and improved standards thereof can leverage public research to new levels where new relevant research questions can be found that are too general to be efficiently studied in closed industrial research laboratories.

1.5 Motivation for the work

My intension with this work is that it should be a significant contribution to the development of environment-friendly technology for complex products.

The complexity that engineers must handle when they in addition to developing a high quality product, also have to take the life cycle of the product with regards to environmental influences into consideration is huge. New types of knowledge must be incorporated into the design processes and new types of software must be developed to support environmental calculations.

Complexity is a major enemy against the development of long term successful software. Advanced software needs to be maintained - work which requires advanced software engineering skills, which is a scarce resource that many times is used to its limits. I hope that these new software development techniques will improve the working situation for engineers in software industry.

Acknowledgements

This research has been funded by the Swedish National Board for Technical and Industrial Development and ABB STAL AB.

The work was conducted during three years of graduate studies at the Department of Computer and Information Science at Linköping University, and five years of work as project leader for two consecutive research projects in cooperation with ABB STAL AB.

During this time I have met and worked with many people who I appreciate and would like to acknowledge. Instead of writing a list here, I hope that we in one way or another can cooperate in the future and share added value from this work.

Part I Properties of
 Knowledge-Based
 Systems for
 Engineering

2 Introduction to Part I

This thesis has been written with a special goal in mind, namely bridging the gap between model-based KBS⁷-technology and its utilization in large-scale industrial applications.

The difference between research models and useful industrial models is size. It is reasonable for a researcher or knowledge engineer to hand-craft a model that contains a few hundred components directly into a textual frame-based language. It is, however, impossible to keep track of the complexity of a realistic industrial model containing thousands of components at this level of abstraction. The user interface software architecture (UISA) described in the thesis is meant to be a fundamental framework for implementing knowledge engineering tools for managing such complexity. It combines existing knowledge from research disciplines such as model-based knowledge engineering, object-oriented programming, user interfaces and cognitive psychology⁸. This novel combination is believed to be a powerful vehicle for approaching a larger goal: efficient utilization and management of large scale model-based knowledge-bases in industrial applications.

The user interface issues dealt with in this thesis are related to many other research challenges in the area of knowledge-based systems. The following two sections will give a brief introduction to some of the fundamentals questions and important research challenges to the field.

2.1 Knowledge in a knowledge-based system

A definition of the goal of knowledge-based systems presented in 1990 was:

Knowledge, to be useful, must be available to those who need it. The goal of knowledge-based systems research is to discover how to deliver, in a timely and accurate way, the knowledge helpful for a particular task - that is, how to make wide varieties of expertise available to decision makers when and where they need it.[Buchanan et al. 90] page 395.

Knowledge in a knowledge-base can be seen as a stored potential for generating rational actions that will lead to the completion of a particular

7. Knowledge Based System

8. Appendix D page 215, and Appendix E page 218.

task.

The thesis presents an information system software architecture built on a thorough theoretical basis, that enables providing fast interactive access to large amounts of knowledge for a group of proficient users, namely knowledge engineers and domain experts who provide the KBS with its knowledge. Fast access is provided through a special user interface design that requires understanding some application-specific graphical syntax and semantics which cannot be expected from an “ordinary” user without special training. The description of the software architecture for implementing this kind of “fast knowledge access interface” is given in chapter 18.

Before we continue by describing related topics and aspects of KBS-research to put this research in context, it might be valuable to examine one of its most important fundamentals, namely the definition of the term knowledge.

What do we mean by “knowledge”? Everyone has some understanding of the word and uses it, but few can give it an appropriate definition. During the 70’s and early 80’s there was a significant debate amongst artificial intelligence (AI) researchers about the benefits and value of different kinds of knowledge representations.

Allen Newell argued that progress could be speeded up by a better understanding of knowledge and the nature of knowledge. In “The Knowledge Level” [Newell 82] he described knowledge in functional terms; as something that lies above the symbolic representations it can be given in a computer. A hierarchy of computer system levels was defined, where the highest is the “knowledge level”, and the one immediately beneath it is the “symbolic level”. The idea was formulated in;

***The Knowledge Level Hypothesis.** There exists a distinct computer systems level, lying immediately above the symbol level, which is characterized by knowledge as the medium and the principle of rationality as the law of behavior.[Newell 82] page 99*

Knowledge exists at the knowledge level, but can only be represented at the symbolic level⁹. His description was formulated much in terms of a rational agent. If the agent has a goal, it tries to reach it by using its knowledge for selecting actions that lead to progress towards the goal. The following quote gives an idea of the nature of Newell’s conception of knowledge;

9. There are other ways of representing knowledge than symbols. Connectionist approaches and conceptual spaces.

Knowledge is intimately linked with rationality. Systems of which rationality can be posited can be said to have knowledge. It is unclear in what sense other systems can be said to have knowledge.

Knowledge is a competence-like notion, being a potential for generating action.

Knowledge serves as a specification of what a symbol structure should be able to do.[Newell 82] page 100

The coupling between knowledge and rationality are especially articulated in Newell's definition of "the principle of rationality", and "knowledge".

Principle of rationality. *If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.*

Knowledge. *Whatever can be ascribed to an agent, such that its behavior can be computed according to the principle of rationality.[Newell 82] page 102,105*

From an industrial designer's point of view the two definitions could be interpreted in the following way. The goal is to fulfill the requirements specification. Knowledge or "know-how" must be represented in a way that allows rational actions to be selected and performed, until a design is created which satisfies the requirements.

The definitions leave the knowledge representation questions open. Knowledge can be encoded in any form that proves itself to be valuable in practice. This was also one of Newell's messages

... the solution lies in more practice and more attention to what emerges there as pragmatically successful.[Newell 82] page 94

Several knowledge representation techniques have proven themselves to be successful for industrial applications. Probably the most well-known and successful applications are the X-family of systems, developed by Digital Equipment, for solving computer configuration tasks amongst others. In 1989 they were used to configure some 100,000 DEC systems[Harmon 89]. The basic knowledge representation used in the systems are rules written in OPS5 [Forgy 81]. Problems of managing complexity arise when the size of the knowledge-base grows. XCON, the largest system in the X-family contains more than 10 000 rules. Structure must be imposed above the rule-level, or rather the rule-base should be generated from "better" knowledge structures. "Better" means that the source knowledge structures should be easier to acquire, explain and maintain. The origin and some ideas behind the organization of these structures are described in more detail in chapter

3.

The idea of “components of expertise” has been pursued by Luc Steels [Steels 90]. His initial ideas were presented in 1984 under the title “Second Generation Expert Systems” [Steele 84]. The “Second Generation” meant that it was a new approach to overcome many of the limitations with the first generation rule-based systems, particularly those limitations dealing with knowledge acquisition.

Experts do not think in terms of rules, but rather in terms of domain concepts. Expertise often lies in knowing *how* to think about a problem, i.e. in what terms, and at what level of abstraction. Once this is formulated explicitly in a conceptual model¹⁰ of the domain, the expert has a language in which the large bulk of detailed domain knowledge can be expressed. More about this will be discussed in chapter 3 and chapter 5.

Before describing some major challenges in knowledge-based research, some comments must be made on the term “expert systems” compared to “knowledge-based systems”.

The term “expert systems” is commonly used to describe these (knowledge-based) systems for three reasons. Very often their performance goal in a target task is the level of competence exhibited by human experts, their knowledge is elicited from experts, and their reasoning methods are based on problem-solving techniques and strategies used by experts.

Because these three characteristics are only suggestive, but are neither necessary nor sufficient for building intelligent systems of some utility, we prefer the term “knowledge-based systems” for applications of artificial intelligence (AI) to problem-solving and decision-making tasks. [Buchanan et al. 90] page 395

There is good reason for adopting the more general term “knowledge-based system” in the area of knowledge-based design (which is my target). Many design problems are rather routine-like¹¹. The importance of developing and applying KBS-technology to these problems is not because a high level of scarce expertise is needed to solve them, but rather because of their inherent complexity and that they are boring and error prone when conducted manually. Examples of such tasks are design rule checking, and critiquing of complex designs. KBS-technology seems promising for enabling some of these tasks to be automated at a reasonable cost, which would not be the case with traditional programming techniques.

10. More specifically what is needed is a formally specified conceptual model of the target domain. Conceptual models and formal conceptual models or meta-models are discussed in more detail in chapter 5 on page 29.

11. Brown and Chandrasekaran refer to routine design tasks as class 3 design [Brown et al. 89] pp 34.

Despite this, since the term “expert system” has been around for quite a long time and is well established and frequently used in the literature, it will also be used here as a synonym for “knowledge-based system”.

The next section mentions important research challenges in the KBS-area [Buchanan et al. 90]. These will become easier to work on if progress is made *first* on facilitating the development of user interfaces for knowledge engineering and knowledge-based systems.

2.2 Challenges

There are several technical challenges that have to be overcome before more advanced knowledge-based systems can be built than the ones in use today. Most challenges are design considerations, i.e. the basic componential framework of expert systems has to be enhanced in order to meet these challenges.

2.2.1 Use and representation of knowledge

During the 70's, production-rules were the commonly adopted mechanism for knowledge representation in expert systems. Later, frame-based and hybrid systems appeared, and filled in some of the gaps left by pure rule-based systems. To a large group, expert systems became analogous with their computational mechanisms. Knowledge engineering became a way of transforming an expert's problem solving method into the computational mechanism provided by an expert system shell. This perspective was misleading.

The challenge is to describe knowledge at the domain knowledge level. To build tools that can easily acquire those domain knowledge level descriptions in the terms and concepts used by domain experts, not in terms of rules and frames. In the area of engineering design, for instance, a domain knowledge level description could be a description how to make an extension to a partial design, how to check design constraints and how to change the design if certain constraints are violated.

The knowledge descriptions acquired at the domain knowledge level must then be transformed into an adequate executable representation at the computational level, such as rules or procedures. How to do this and how to do it efficiently are challenges in themselves.

2.2.2 Acquisition and maintenance of knowledge-bases

Knowledge acquisition is a famous bottle-neck in the building of knowledge-based systems. It is a major effort to enter enough knowledge into a system so that it can deliver an acceptable performance. And once the knowledge is captured, the effort to maintain it remains. To mention an example, Digital Equipment Co. needs 40 knowledge engineers for

developing and maintaining the X-systems. Although there is some variation, about 40-50% of the rules are either created or re-written every year. For all systems, there are about 20,000 rules and a database is used that contains descriptions of about 30,000 parts [Harmon 89].

2.2.3 Explanation

The contents of a knowledge-base are, as mentioned, highly dynamic. Changes and updates naturally provide a continuous stream of opportunities for introducing errors. Users of a knowledge-based system must be able to check the generated solutions, and get justifications and explanations why certain choices were made. Otherwise they will not have confidence in the system. The first generation systems' *how* and *why* explanations, which are based on a trace of the rule execution, are not enough. This is particularly the case for second generation systems, where rules might have been generated automatically and much of the contents in their premises might consist of heuristics to optimize the reasoning process. A strong connection with the descriptions used at the domain knowledge level for acquiring the knowledge seems to be necessary in order to provide satisfactory explanations.

2.2.4 Flexibility

The KBS must be extensible without a substantial amount of reprogramming being required. Adding new knowledge or making modifications should not have unexpected "side effects". Reasoning about problems at the border of the system's competence should be possible, with a "graceful degradation".

2.2.5 Partnership Role

In the future, more engineering work should be conducted in cooperation with computers. With better user interfaces and a better adaptation of the internal organization in the computer to the actual problem solving situations for the users i.e. their domain knowledge level, significant enhancements in productivity can be expected.

2.2.6 Summary of general challenges

To conclude, we need ways of representing knowledge at the domain knowledge level that are closer to the terms and concepts that experts think in. We need tools for managing large knowledge-bases. A knowledge-based system must be able to explain and justify its solutions in a way that is easy for the user to comprehend.

2.3 Motivation for the work

Useful industrial KBS-applications have the common feature that they become large and complex. Development and maintenance become difficult tasks. It should be possible to make this type of work more interesting, creative and efficient if user interfaces are adapted to what is known about human cognitive capabilities.

The thesis points to theory in the literature that should be taken into consideration when implementing user interfaces for knowledge-based systems. A user interface software architecture intended to meet the knowledge engineers' needs for developing model-based KBSs in the future is given. Some differences between our approach and others' are described in section 20.3 on page 165.

A knowledge-base should reflect the knowledge of a human expert. It has been shown that a KBS-prototype can be a useful tool for the expert in refining his knowledge and gaining new insights about it during the knowledge acquisition process [Eriksson 89][Eriksson 91][Sandahl 87][Shaw et al. 90][Steels 90]. A prerequisite for this "active expert approach" is that the system has a suitable user interface that can be directly operated by the expert. This presupposes that the interface operates on the domain knowledge level, i.e. the objects that are manipulated through the interface correspond directly to objects in the KBS's target domain.

Another prerequisite for successful knowledge acquisition for a model-based KBS, is that to some extent a formalized conceptual model¹² of the domain can be found and explicitly expressed in the user interface representation. During the knowledge acquisition process, many KBS-prototypes may have to be built to support entering knowledge and to give the expert an understanding of how the system is going to work. During the prototyping stage, major modifications might be made to the meta-model of the domain, which in turn has major impacts on the user interface design. If customized graphical user interfaces could be built and modified easily, it would be of great value for the knowledge acquisition process. The proposed user interface software architecture (UISA) will significantly reduce the implementation time for this type of user interface compared to the current state of the art. The way to achieve this is using a language with full object-orientation [Wegner 87]¹³, a clean separation of functionality between different object classes and a large structured class-library of

12. More details about the formalized conceptual model (or meta-model as it is referred to in this thesis) are given in section 5.1.4 on page 33. A meta-model has much in common with entity-relationship models [Chen 76] but has many additional features that are needed for specifying valid structures in knowledge-bases.

13. A fully *object-oriented* language has objects that are instances of classes embedded in an inheritance hierarchy. Smalltalk and Simula are fully object-oriented languages. Other types of languages are *object-based* languages such as ADA or *class-based* (without inheritance) such as CLU.

easily configurable user interface object classes.

In order to develop a UISA for future model-based KBSs, we must know what these will look like. As will be described in chapter 3, a general framework for knowledge components is maturing. In the future, many of these components will be implemented, carefully studied and well understood. Once a large enough number of such components are available, there will be a need for libraries of “knowledge components”, that make different knowledge entities easily available as building blocks for the implementation of new systems.

The complexity-level¹⁴ of both the components and the applications that are assembled is expected to be very high. In order to manage the complexity, the expert and the knowledge engineer will need better tools for visualization and manipulation of the contents of the KBS. Since the external shape of knowledge in different application domains shows large variations, the development of improved¹⁵ knowledge components will continue to be a hand-crafted task requiring a high degree of skill and a good understanding of the workings of different reusable building blocks. The development of such skills will benefit greatly from a higher communication bandwidth and incrementality in the human-computer interface.

The proposed UISA is a frame-work for the design and implementation of efficient browsers and manipulation tools for object graphs. Such interactive tools allow knowledge structures to be directly manipulated, without the need for transformations into some kind of textual language or knowledge representation. As mentioned, this is especially valuable in the early knowledge acquisition phases, where concepts and appropriate names for them are not yet clearly defined. In a graph there is no need to define appropriate names, since different concepts can be accessed and referred to by pointing. The graphs are used as a way of modelling the contents of experts’ mental models of the target domain for the developed knowledge-based problem solver. Tools for efficient interaction with such conceptual graphs may have the same impact on productivity for knowledge engineering as hierarchical text editors have had for document development.

14. Complexity-level in terms of the number of interacting parts, and overall size of the system.

15. Improved in terms of their generality, easiness to learn and use, and performance in the final delivery system.

3 Components of Expertise

This chapter will present an overall picture of the different components in what was called second generation knowledge-based systems in the early 90s. This is intended to highlight the importance of organized knowledge structures, and justify the effort of building efficient tools for their interactive manipulation. In order to be brief, the description has to be somewhat condensed. Most of the ideas come from [Steels 90][Steels 89a] and [Brown et al. 89].

Steels, as one of the proponents of a “paradigm shift”, away from the simple rule-based model of expert systems, launched the term “Second Generation Expert Systems” [Steele 84].

It was based on a criticism that first-generation systems like MYCIN..., only contained surface knowledge and therefore were brittle, had weak explanation facilities, no clear boundaries for knowledge acquisition, etc. Its purpose was to overcome many of the limitations in the first generation of rule-based systems. Second generation expert systems were supposed to have both a representation of the deep knowledge and of the surface knowledge. They could do two kinds of reasoning. Surface reasoning was used to solve cases quickly and efficiently. The deep reasoning was available there as a backup when the surface knowledge failed. [Steels 89a]

Second generation systems have a representational component which is a model of the artifact of interest, and a problem-solving component which solves a problem¹⁶. Typical target problems could be diagnosing a malfunctioning device given a set of symptoms, or to generate the design of a device from a given functional specification.

The representational and problem-solving components contain subcomponents that are separable and generally useful for many purposes. Subcomponents of the problem-solving component for diagnosing malfunctions in a car could be used for diagnosing malfunctions in an aircraft. Many subcomponents of the representational model of a subsystem in a computer could be used for both diagnosing faults and for configuring similar subsystems.

16. There is also a learning component, whose function is to compile experience from consultations of the deep model into fast surface knowledge. The learning mechanism, although important and interesting in itself, will not be treated further here.

3.1 Deep versus surface knowledge

Before describing the representational and problem solving components in more detail, some comment must be made on the division of knowledge into deep and surface knowledge.

Deep knowledge makes explicit the models of the domain, and the inference calculus that operates over these models. A typical example of a domain model for diagnosis is a causal model, linking properties of components through cause-effect relations. An inference calculus operating over this model could take the form of a set of axioms that prescribe valid inferences over the causal network.

The term surface knowledge usually stands for knowledge that is immediately applicable in a problem-solving situation, i.e. recognizing the situation will immediately lead to selection (and execution) of an appropriate action. In the problem-solving situation, there are two tasks that have to be conducted simultaneously. Firstly, to contribute to the solution of the problem, and secondly, controlling the path of the problem-solving process. Surface knowledge usually contains a mixture of both these ingredients. An entity of surface knowledge (e.g. a rule) is therefore often not self-contained and explainable in isolation. It consists of selected portions of deep knowledge and heuristics for guiding the problem-solving process.¹⁷

3.2 Conceptual and pragmatic aspects of expertise

Expertise is the ability to solve certain problems in a specific domain. When analyzing expertise at the knowledge-level, a distinction can be made between its conceptual and pragmatic aspects. The conceptual aspects concern the terms in which to think about a problem. The pragmatic aspects focus on how to deal with limitations that humans and computers have during the problem solving process. Examples of such limitations are:

Limitations in time: During design the number of alternatives for a design-choice may be large. It takes too much time to check all of them. Ways of pruning the search-space must be considered.

Limitations in space and access-times: Storage space is always limited. Efficient ways of representing data for solving certain sub-problems are needed.

Limitations in requirement formulations: It is not efficient to specify

17. Note that the distinction between deep and surface knowledge has nothing to do with the formalism that is chosen to implement it. The distinction is made at the knowledge level, and is not a computational one. Deep knowledge can be implemented by rules and surface knowledge by using frames. Brown's language, DSPL, is a good example of the latter [Brown et al. 89].

everything. Assumptions must be made about the unstated.

Limitations in evaluation possibilities: The resources for evaluating a design are limited. Experiments and simulations must be carefully prepared to reveal valuable information compared to costs.

Limitations in theory formation: Models must be derived inductively via real world interaction, or via communication with other humans. This often makes the models limited in their accuracy and scope of prediction.

The conceptual aspects are mainly captured in the domain models and the pragmatic aspects in different problem solving methods.

3.3 Domain models and deep knowledge

A domain model resembles some aspect of a domain. A static domain model is NOT deep knowledge in itself. It first becomes deep knowledge when it is accompanied with an inference calculus¹⁸, that can generate answers to queries to the model.

Typical domain models for diagnosis and to some extent design are [Steels 90]:

- * A structural model describing part-whole relationships between components and subsystems.
- * An interconnection model showing how different components are connected to each other.
- * A causal model representing the cause-effect relationship between properties of components.
- * A geometrical model representing the spatial relations between components.
- * A functional or behavioural model representing how the function of the whole follows from the function of the parts.
- * A fault model, representing for each function, possible faults and components that may be responsible for the fault.
- * An associational model relating observed properties with states of the system.

Domain models are particularly suitable for knowledge acquisition. In the area of design, engineers use drawings, function diagrams, and so forth for

18. An inference calculus can be implemented in rules or using object-oriented methods. Rules are usually inefficient, but require much less programming knowledge on the part of their implementor. Methods are preferred when the speed is of concern, since they allow explicit control of the computation processes. Usually the method-type implementation is a recursive straightforward procedure.

developing designs. The inference calculus that engineers use in their thinking processes is intimately connected to this type of representation media. The inference calculus for the drawing of an electronic circuit design (which is an interconnection model) will have rules and methods for deriving the components that are connected to one particular component, the fan-out needs for a certain circuit, etc.

In order to develop tools for interacting with and manipulating the deep knowledge, we need explicitly formulated conceptual models of the domain, represented by for instance object-oriented domain models (Section 9.5 on page 72 shows an example).

Using an editor for domain models and a set of instantiated model cases, it should be possible to elicit the inference calculus of the deep knowledge. Given an appropriate language integrated into the domain model editor, the expert should be able to enter parts of the inference calculus himself, without support from a knowledge engineer.

Much of this deep knowledge is probably usable in several different domains. Providing extendable components inside a “knowledge engineer’s toolbox” will certainly be beneficial for knowledge engineering productivity.

3.4 Generic tasks

A generic task is characterized by information about the following [Bylander 87]:

1. The type of problem (the type of input and output). What is the function of the generic task? What is the generic task good for?
2. The representation of knowledge. How should knowledge be organized and structured to accomplish the function of the generic task? In particular, what are the type of generic concepts that are involved in the generic task? What concepts are the input and output about? How is knowledge organized in terms of concepts?
3. The inference strategy (process, problem solving, control regime). What inference strategy can be applied to the generic task? How does the inference strategy operate on concepts.

Examples of generic tasks are interpretation, diagnosis, construction (including design and planning), monitoring and instruction. A generic task can be implemented with different problem solving methods, using different domain models. Generic tasks can serve as an indexing mechanism for compatible methods and models. Once a generic task is identified in the target problem, its corresponding collection of problem solving methods and domain-model components can be consulted, and a configuration selected to implement a solution.

The idea of generic tasks was put into focus by Chandrasekaran [Chandrasekaran 86]. Later, different authors gave different interpretations of the term “generic task”, but one main feature seems to be that properties of generic tasks should be transferable across application domains. Examples of generic tasks are diagnosis and classification.

3.5 Problem solving methods

A problem solving method is a knowledge level characterization of how a problem may be solved. Much of the work on problem solving methods originates from a series of knowledge acquisition tools developed by McDermott and his collaborators [Marcus 88a][McDermott 88].

Typical examples of problem-solving methods are cover-and-differentiate for diagnosis, and propose-and-revise for construction. Cover-and-differentiate first proposes a set of candidate explanations that cover most symptoms, and then tries to find the most likely ones by gathering facts on features that differentiate the candidate explanations.

Propose-and-revise first proposes a partial solution and then revises it by resolving violated constraints. There are three roles that knowledge can play in this problem solving method [Marcus et al. 89]:

- 1) PROPOSE-A-DESIGN-EXTENSION
- 2) IDENTIFY-A-CONSTRAINT on a part of the design.
- 3) PROPOSE-A-FIX for a constraint violation.

Knowledge roles of these types establish a framework for the knowledge acquisition process. The knowledge-engineer knows what to look for and how to transfer different knowledge entities into a working system. By restricting the context for knowledge-acquisition to a particular problem solving method that is well understood, the work of the knowledge engineer can be automated. This has been successfully done in several knowledge acquisition tools. Their job is to query the expert for knowledge-entities, check them for consistency and ask questions such that enough knowledge is acquired for generating a working system.

MOLE is an example of a knowledge-acquisition-tool for the cover-and-differentiate problem solving method [Eshelman 88], and SALT is a well-known example of a knowledge acquisition tool for the propose-and-revise method [Marcus et al. 89].

To give an idea of the sophistication of such a knowledge acquisition tool we can mention that SALT has been used for developing a propose-and-revise problem solver that designs elevators [Marcus et al. 88]. The generated knowledge-base for the performance system contains about 3000 OPS5 rules, of which about 70% are domain-specific and generated from the acquired knowledge, and the rest are used for controlling the problem-solving process.

3.6 Summary and challenges

As described above, the contour of a general framework for organizing and reusing different components of knowledge is emerging. One important question and challenge is: how are we going about to make this growing body of knowledge efficiently available to knowledge engineers in industry? To do this, we clearly need to organize and store knowledge, code, documentation etc. in some kind of engineering databases [Johansson 89]. The size of these databases will become very large, so the ways of accessing information must be flexible.

4 Knowledge-based Systems for Design

Knowledge-based systems for design are of interest since a high potential benefit from better user interfaces for knowledge engineering can be expected. There are many existing expert systems for design¹⁹. Although developing new KBSs for design is much of a hand-crafted task starting at the expert system shell level or even lower, there are some knowledge-acquisition tools such as SALT and SIZZLE that support knowledge-acquisition when a certain kind of problem solving method is applicable [Marcus 88a]. Ideally, what one would like to have is a knowledge engineers tool- and component library that contains useful well-working modules that could be assembled into a system. The tools and modules could be indexed according to their corresponding generic tasks. The next organization level in the library would be a set of problem-solving methods that are applicable for performing each generic task. Each method should have its accompanying knowledge-acquisition-tools and knowledge representation components. Which problem-solving method is chosen will depend on what knowledge is available for acquisition for the current application.

An average design application would probably need to combine problem-solving methods for several different generic tasks. Therefore a standardization of the tool and component interfaces will be needed to enable proper integration. One important research topic is to find out what these interfaces should look like.

Existing applications that are built from components in the library should be accessible for the knowledge engineer to study and copy code from. They will be a shareable and an efficient source of knowledge engineer know-how.

Since each such application will be large, the size of the toolbox, including many existing applications, will be considerable. To be able to find something in information quantities of these sizes, the contents must be well-organized into access structures that are easy for the knowledge engineer to remember and navigate in. The access time is the time it takes from the moment the knowledge engineer knows that he needs some information, knowledge or blocks of implementation code, until he has it²⁰

19. Papers surveying relevant topics and suggesting general frameworks for knowledge-based design are referred to in section 4.4

on the screen.

The accessibility, i.e. the ease by which a suitable entity of information can be found, depends on a combination of the access time and the remembering and searching effort required for finding the desired content in the knowledge-base. If the knowledge engineer is highly skilled, accessibility and access time will be fundamental constraints that limit his/her performance. If the supporting system is highly incremental, the knowledge engineer can develop his thoughts in cooperation with the system, where the system is responsible for remembering the details and checking for consistency. The user interface is fundamental for providing these properties.

This chapter will give an idea of what kind of components for knowledge-based design that could be part of the suggested knowledge engineer's toolbox.

The intention is to show that there is a rich variety of mutually non-replaceable components that have to be present in such a toolbox. Extensive support in the user interface is needed for providing easy access to available components. Efforts to develop libraries of knowledge components will not become rewarding until tools for building flexible interactive user interfaces that enable development, assembly and manipulation of such components are available.

4.1 Concepts for describing the design process

The notion of design-space is a useful metaphor for describing design in general terms without having to refer to an explicit example. It will be used here to introduce some terminology for later discussion.

Many design-problems can be seen as the process of searching for a good point in a space of possible designs. The axes of the space represent a scale of different possible design-choices, and may be both continuous or discrete. The design-domain will define which axes there will be, and which areas in the space contain legal designs.

Input to the design-problem-solver will be a set of requirements that will determine certain areas inside the legal design-areas as acceptable solutions. The output is a single point or a set of points inside the acceptable areas that can be ordered according to some cost- or desirability-function.

In the space, the hyper-volumes of acceptable solutions are shaped by different design-constraints. The design-process will proceed as a sequence of design-choices that will restrict the space to be searched by one dimension for each choice. A design-choice may restrict the space-to-be-searched in such a way that no part of it will cover any acceptable area.

20. Or handles to it that can be manipulated or used as a symbolic reference.

This is detected by constraint-checking.

The dimensionality of the design-space is large for any non-trivial design-task. Each design-choice and constraint-check is associated with a computational cost, and for non-trivial cases it is not realistic to search the whole space for acceptable solutions. Instead the search must be guided by knowledge.

An entity of guiding knowledge can be seen as a demon guarding an area that covers a set of dimensions in the space. If the design-state enters that area, the demon says how and where to proceed²¹. If there is no knowledge, (i.e. no demon) then the area must be searched, and its different points be evaluated.

At this point we can identify three types of knowledge that have different roles²². The first type applies when the design-state covers acceptable solution areas. It generates the next design-extension, i.e. makes a design-choice. The second type checks constraints, and reports if a constraint is violated. The third type applies when the design-state is completely unacceptable and can be referred to as a fix. An entity of fix-knowledge guards a particular illegal area for the design-state and suggests what design parameters to change to make it move to legal and acceptable areas again.

The three types of knowledge are given different names by different researchers. Here we have adopted the terminology of [Marcus et al. 88] and call them 1) design-extension 2) constraint-checker and 3) fix.

Each of these knowledge-roles have common behavioural properties that could be extracted and implemented in an object-oriented class-hierarchy framework and supplied as building blocks in the knowledge engineer's toolbox.

4.2 The components of a design instantiation

The output of a KBS for design is a design instantiation that consists of a complete description of components and their relations which implement the required functionality. The “deepness” in the description depends on the level of functionality in the primitive (i.e. terminal) components. When having access to a domain-specific component library, there will always be “build or buy” decisions to make. These will be based on evaluations of different alternatives, or on previous experience (i.e. surface knowledge). To enable search and evaluation, the components that may be building-blocks in a design instantiation must provide functions that can answer questions such as: What is the weight? Price? Center of mass? This kind of

21. This is similar to the usual rule-based black-board architecture, where the design-state is on the black-board, the dimensions represent the variables in the LHS of the rules, and the area is determined by the expression using the variables.

22. The same knowledge roles were mentioned in section 3.5 on page 21

inference calculus must be provided in the form of functions and their interfaces should be “standardized” for a certain domain of applications. This will allow libraries of deep knowledge components to work together in easily assembled configurations that were never thought of by their initial component implementors. Many rather trivial but necessary interface design decisions will have to be made in order to allow “software-IC”-libraries²³ of these kind to be built. Once a working standard is defined, the work of developing components can be distributed to groups of domain specialists that do not necessarily have any contact with each other. No such standards exist yet, but many schemes of common practice already exist. It is research a task to expose such schemes and find out which ones are candidates for standardization and justify why.

4.3 Design plans versus iteration

Depending on the nature of the design space, different problem solving methods are preferable. In a wide design space, most solutions are legal. The problem lies in an efficient division of the space and filling out the details in a rational way. Surface knowledge in the form of design plans are well-suited for this job. In a sparse design space, design choices may depend heavily on each other, and many conflicting constraints may interact. If no well-functioning standard cases are available, an iterative problem solving method like the propose-and-revise method may be more applicable. The “toolbox components” for this method have already been mentioned.

Brown has developed a framework for plan-based design [Brown et al. 89]. The components of a plan consist of nine different component-types or “agents” that each have common behavioural functionality. These could be implemented as reusable components in a knowledge engineer’s toolbox. The work of the knowledge engineer, or domain expert, would be to assemble and specify the details of instances of the appropriate planning components.

4.4 Observations and conclusions

There are numerous other techniques of design problem solving that could be supported with tools in a knowledge engineer’s toolbox. Detailed descriptions of this subject can be found in for instance [Chandrasekaran

23. The term “software-IC” draws a parallel with the explosive development of electronics after proper interface standards were defined and accepted by industry. Note that also for hardware integrated circuits, the contents of the standards were somewhat less exciting details from a functionality point of view. They specified chip-package pin-outs, voltages on chip power supplies, fan-outs etc. In the same way standards for software-IC’s will be “low-level details” such as naming conventions for messages, message protocols, standardized abstract datatypes etc. A discussion on software-IC’s can be found in [Ledbetter&Cox 85]

90][Mostow 85][Mittal&Araya 86][Tong 87]. It is clear that design is very varied in nature, and that there is no single problem solving-method that can cover more than a small domain of design problems. Therefore, to be efficiently equipped for the development of design problem solvers, the knowledge engineer must have access to a large library of problem-solving methods and their knowledge building blocks.

Since the details of the components' internal workings might be rather complex, it should be possible to just copy or inherit the relevant parts, without having to scrutinize much of their implementation code. Examples of existing applications must be available to allow the knowledge engineer to inspect and recall how to use the different knowledge components. A problem-solving method or set of deep knowledge components may require some adaptation to suit a new domain. In order to facilitate interaction with new and adapted knowledge components, tools for implementing user interface components for such types of interaction must also be provided.

As has been indicated, there are a large number of different types of knowledge entities that have to cooperate in a knowledge-based system for design. There is no particular ordering among them, and therefore no natural way to order them in a textual programming language. In the same way, as it is tremendously inefficient to work with a large object-oriented system in Smalltalk without a browser, it will be impossible to develop a large knowledge-base for design without good browsing tools. Smalltalk has a well-defined domain model²⁴ that is revealed to the user through the browsers. It is an important research question to discover meta-domain models and browsers for knowledge-based design-systems.

This discussion leaves us with the conclusion that efficient support for building customized user interfaces for any type of knowledge components would be of great importance for making progress in research on knowledge-based systems for design.

24. The Smalltalk domain model consists of class-categories, classes, message-protocols, messages, class-variables, stack-frames etc. Access to instances of each of these entity-types is well provided for by the browsers, inspectors etc. [Goldberg&Robson 83].

4.5 Some challenges for knowledge-based design systems

Here, a list of research challenges in the area of knowledge-based design is given. It establishes the main topic of this thesis in relation to other subjects.

- * Finding useful structures for deep knowledge components. This would include identifying generic components and finding designs for method protocols that implement their inference calculus.
- * Finding useful structures and efficient implementations of surface design knowledge. These could, for instance, be based on design-plans as suggested by Brown [Brown et al. 89].
- * Development of class hierarchies of knowledge components for different domains.
- * Developing a taxonomy of problem solving methods for design.
- * Learning in design. A design problem solver can increase its efficiency by learning (i.e. generating surface knowledge) after successfully finding a solution in the design space after search [Horner&Brown 90].
- * Methodology and tools for visualization of the problem solving process for debugging and explanation purposes.
- * **Finding efficient methods and tools for generating user interfaces that allow incremental interaction with large design knowledge-bases.**

5 The KBS Development Process

Facilitating large-scale knowledge acquisition is one of the fundamental tasks for the future knowledge engineer. The job will be to provide the domain expert with a language and tools so that he/she is able to enter the large bulk of detailed knowledge entities into the knowledge-base by himself/herself. This chapter provides a map of the KBS development process, including acquisition of an understanding of the concepts and abstractions in the target domain, such that special “customized” knowledge acquisition (and maintenance) tools can be developed. It also describes the properties of some important supporting tools for the knowledge engineer which speed up this process. The UISA described in Part IV is an important component for building such supporting tools.

5.1 A model of the KBS development process

Figure 2 on page 31 depicts some of the important relations between processes and intermediate forms of knowledge during the idealized development of a knowledge-based system. This model is our conception of how the KBS development process should be divided to benefit from knowledge about:

- * Models of human cognitive processes and knowledge organization (e.g. [Card et al. 83], [Lindsay&Norman 77],[Minsky 75],[Schank 82]).
- * Techniques and models for knowledge acquisition (e.g. [Marcus 88a],[Shaw et al. 90])
- * Theory and techniques for object-oriented modelling of limited subsystems in the real world [Sundgren 73][Sundgren 92] [Coad&Yourdon 90][Rumbaugh 91].
- * Current understanding of implementation techniques for object-oriented data-bases (e.g.[Cattell et al. 96][Johansson 89][Zdonik 90]).

The whole model is drawn as if it were a uni-directional development process. In reality it is heavily iterative. The arrows depict a partial ordering of what must be available before the next process can be started to generate or modify its successor(s).

One major benefit of this development model is that it enables the expert and the knowledge engineer to separate their different domains of expertise. The domain expert is relieved from technical KBS implementation details and the knowledge engineer will not have to understand all the details of the target domain. It is important that the expert and the knowledge engineer are able to iterate towards a well-defined and acceptable meta-model of the target domain.

Having an explicit meta-model, no matter if it is acceptable or not, the knowledge engineer can provide the target domain expert with interactive tools for developing and experimenting with prototype models of the domain. An explicitly formulated meta-model will be a significant aid for the specification and understanding of both the target domain and how the KBS should operate.

The KBS development process depicted is introduced from top to bottom as follows:

The development of expertise starts with prospective experts interacting with some artificial²⁵ system in the real world, the real world system. The interaction enables development of skills in how to handle such systems in their particular target domain. The skills are stored in the form of mental models in the minds of the experts and may in some cases hold decades of experience in a particular domain.

During the early knowledge acquisition process, parts of the mental models are formulated in some representation that can be communicated to others. These formulations are called conceptual models. The conceptual models are then analyzed and a model of the conceptual models is developed that allows a more precise reformulation of the contents of the conceptual models. This “high-level” model is called a meta-model. A synonymous term is domain model. It can be seen as a language for describing instances of models of the target domain. The meta-model is later used as an input specification for the design and implementation of knowledge acquisition tools and data acquisition tools. Many of the concepts specified in the meta-model will have a direct one-to-one correspondence with definitions in the database implementation model or database schema. The knowledge acquisition tools will be used by the expert for interactive modelling and entering of knowledge. Output from the tools will be knowledge entities such as rules and partial plans that represent expert behavior in the domain.

25. In the scope of this thesis, we are mainly interested in KBS for artificial systems developed by engineers. In these domains, concepts can be cleanly defined and adequate formal object-oriented models developed. In natural target domains such as medicine, concepts are often hard to describe in terms of formal models and thus less applicable for the type of development process and tools described here.

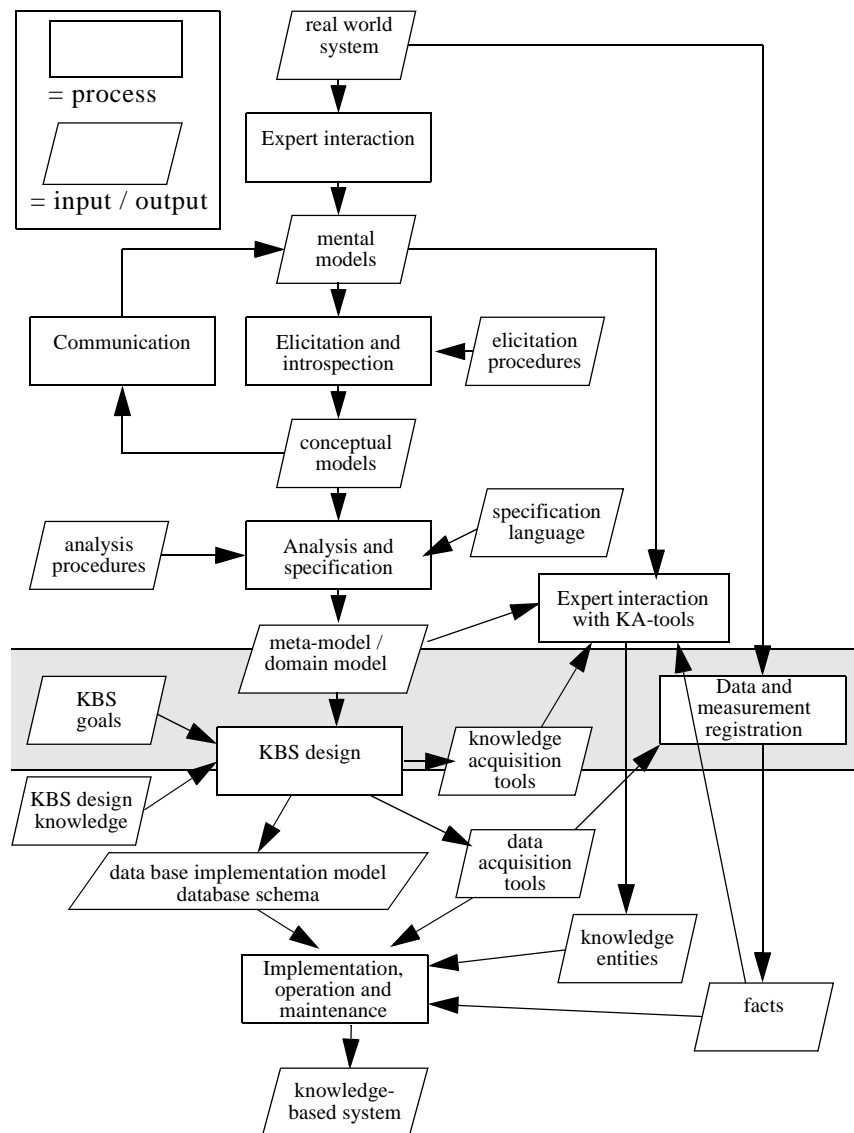


FIGURE 2. The process of developing a knowledge-based system. Arrows depict inputs needed and outputs produced or modified by the different processes. The whole process is highly incremental. In the shaded area, both the domain expert and the knowledge engineer must share a detailed understanding.

Data may often be available in various electronic forms (e.g. CAD-drawings or measurements stored in data-bases) and therefore transformable into facts that can be accessed by the KBS. Facts may of

course also have to be entered manually. Facts describe properties of different objects in the target domain and how particular instances of domain objects relate to each other.

Output from the KBS design process is, in addition to the acquisition tools, a database implementation model or database schema of the final knowledge-based system. This model is in its essence a transformation of the meta-model with several computational implementation decisions specified. The database implementation model depends on which software environment is selected for the final KBS, and how needed functionality can be mapped onto provided facilities.

The process of implementation, operation and maintenance will be continuous for the rest of the life of the knowledge-based system.

The whole process described above is as mentioned heavily iterative. Knowledge and data acquisition tools may have to be modified or re-implemented every time changes are made to the meta-model. Interactive graphical editors for object-oriented models require much effort to implement. The UISA described in Part IV is a framework for facilitating this iterative implementation work.

In the following subsections, some of the above introduced terms are described in more detail.

5.1.1 Real systems in the target domain.

The term real system will be used to denote the system in the target domain that is the focal point for the problem solver in the knowledge-based system which automates parts of the expert's skills. No intrinsic models exist in the real system itself, but rather an ordered or repetitive sequence of activities or events. In the case of R1, an ancestor of the XCON expert systems for configuring Digital Equipments Corporation's VAX-11/780 computer systems[McDermott 82], the real systems are computer installations. The target domain is how these types of computers work and how to configure them into proper installations. In the case of diagnosis, the real system may be a particular computer configuration installed at a certain site and the target domain is how to diagnose this particular type of computer installation.

5.1.2 Mental model

When the expert interacts with real systems, he will acquire mental models from them. These are fragmentary, partial models generated to account for the expert's purposeful interaction with the real system. Mental models represent unexpressed, uncommunicated knowledge in the expert's mind. Mental models are not constrained by any external medium for representation. They can contain memories of everything that the expert can observe, perform and reason about in the real system. Parts of the

mental models can be reformulated and represented symbolically. Such symbolic representations become conceptual models.

5.1.3 Conceptual model

A conceptual model of the real system is a description generated from the mental models of the real system. A conceptual model finds representation in the communication primitives of language and behavior. The mental models affect the form and representation of the conceptual model which, in turn, is constrained by the language and behavioural representational system²⁶. The conceptual models are available for communication to others, since others can acquire mental representations for the primitives of language and behavior that are used in the conceptual models. Since the mental model is unavailable to an observer, the conceptual model is taken to represent the model used to address the task demands of the real system. Conceptual models are used for communicating between humans. They are not formal and not necessarily consistent, and usually require further refinement.

5.1.4 Meta-model / Domain model

Once a conceptual model is made explicit, a formal representation can be used to describe its properties. The formal description of the conceptual model becomes a model of the conceptual model²⁷. It should contain the necessary primitives²⁸ to recreate the operation of the mental model. The model of the conceptual model is also called meta-model since it describes the properties of models that can be expressed using the primitives defined in the meta-model. The meta-model will contain both conceptual and heuristic parts. The conceptual parts can be expressed in some data-model formalism, such as extended entity relationship diagrams or some graphical representation of the domain modeling primitives in our meta-database (Figure 21 on page 82). The heuristics can be expressed in some textual notation (e.g. rules or procedures/methods). When a meta-model has been specified, it is possible to implement a customized knowledge acquisition tool that supports collection and management of the larger knowledge quantities needed for proper problem solving in the real system.

5.1.5 Prototype knowledge-bases

During elicitation procedures, prototype knowledge-bases play an

26. A behavioural description can be communicated by performing a simulation of how a particular computer is configured. The types of actions performed during this simulation are the communication primitives of the behavioural representation system and thus components of a conceptual model.

27. The term "Model of conceptual model", is from [Shaw et al. 90].

28. Some primitives can be implemented as reusable deep knowledge components (recall section 3.3 on page 19).

important role as an expression medium, memory²⁹ and communication channel for conceptual models.

Sometimes prototype knowledge-bases can remain paper-based, but as soon as they become large and complex it is beneficial to store and manipulate them in a computer.

It must be possible to display prototype knowledge-bases in a format that allows them to be a communication medium between the knowledge engineer and the expert, allowing validation and confirmation to take place. Prototype knowledge-bases also work as intellectual tools for the knowledge engineer. They both help him to remember details, and provide focal points for further elicitation. Interactive computer support for developing prototype knowledge-bases is very important. It helps the knowledge engineer to find errors and misconceptions in the conceptual models, and meta-models. Such arise when the models do not correspond to the real system. There may also be errors in the computer-based implementation of the models, which may be internally inconsistent. It is very hard to discover errors and misconceptions without being able to work out sample structures in detail that can be inspected, computed and analyzed. Interactivity and incrementality are very important in speeding up the concept formation process with the help of prototype knowledge-bases.

A knowledge-based system represents a model, that was constructed through the elicitation, analysis, design and implementation process, and is based on the assumptions underlying the representations in each modeling phase. During the development of a knowledge-base it is desirable to maintain a flexible development environment where the knowledge representations can easily be manipulated and updated, but where there is not necessarily any time criticality. High execution speed can be traded with reduced flexibility. It may be worthwhile to generate and distribute a final delivery system from the knowledge-base in the development environment.

5.1.6 Delivery system

This is a final working version of the expert system used by end users, or incorporated into an embedded system. The delivery system usually has to be fast and perform well with restricted resources in terms of processing power, storage space, user-interaction equipment etc. Sometimes the delivery system is called the performance system or target system (not to be confused with target domain and real system). The hardware needed for its

29. Most KBS-development projects last over long periods of time. It is hard for the participants to remember the details of how models were formulated without some kind of documentation. A computer-based prototype knowledge-base with proper browsing tools can function as a common shared memory for knowledge-engineers and experts which provides its own continuously updated documentation.

operation is called the delivery system platform, and the software needed is called the execution environment.

5.2 Important qualities for reducing KBS development time

In the future, the knowledge engineer will have libraries of components of the type described in chapter 2 for developing prototype knowledge-bases. Components, knowledge-bases and test cases from other projects can be made available as initial building blocks and sources of knowledge engineering know-how. This will produce far more efficient communication channels between knowledge engineers than the written word, although a pre-requisite is that the contents and workings of existing systems are easily accessible through browsers and different types of visualization tools. KBS development time can be reduced by supporting reuse of previous know-how, but support must also be given for the knowledge modelling of new domains.

In summary, the following features should be provided by a KBS development environment for reducing the time needed for KBS development:

- 1) High visibility, i.e. a knowledge acquisition tool should provide support for interactive inspection and navigation through models of knowledge.
- 2) High incrementality, i.e. the knowledge and data acquisition tools should enable flexible interactive change and manipulation of models of knowledge and facts.
- 3) Support for incremental development of meta-models. The meta-models should be used as specifications for the design and (automated) implementation of knowledge and data acquisition tools that provide high visibility and incrementality.

6 Observations from the Development of a Knowledge-based Intelligent Front End

In the previous chapters a framework was described for knowledge engineering in general. This chapter reports observations from practical knowledge engineering work done in KEE [Intellicorp 87].

The work on an intelligent front end for structural design was performed in a collaboration project with the Department of Mechanical Engineering [Orsborn 91][Orsborn 93]. It covers areas that require competence from both disciplines.

The chapter will very briefly describe the problem studied, and summarizes the observations that are relevant for the topic of this thesis.

6.1 The task of an intelligent front end for a CAE³⁰-system

Mechanical engineering is one of many disciplines that make considerable use of software systems for assisting in engineering tasks. Such computer aided engineering systems often become very complex. Efforts to make them easier to use seem to have a high yield potential in the form of increased engineering productivity and improved quality in product development.

An intelligent front end (IFE) is software that relieves the engineer from some of the complexity of the underlying CAE-system [Bundy 84]. It allows engineers to define inputs to CAE-programs in domain-specific terms, without having to know much about computer-related details such as input file formats, program commands, naming conventions of objects stored in databases etc. The output of the CAE-system is transformed back into domain-specific representations that are easy for the engineer to interpret.

There are many subtasks for an intelligent front end that can be characterized as routine design³¹. The input files are assembled with the engineers high-level problem description as a kind of requirement

30. Computer Aided Engineering

31. Characteristics of a routine design task are mentioned on page 12.

specification. Knowledge is needed for selecting relevant CAE-analysis programs and routines, for supplying programs with additional data and analysis models relevant for the current problem, and for pure transformation of parts of the information supplied by the engineer into formats readable by programs.

Generated output may need additional post-processing that also depends on how the problem was formulated. The selection and execution of relevant post-processors can also be handled by an IFE.

6.2 KBS features for building and maintaining an IFE

The environmental conditions for an IFE are more demanding than for traditional software products. It must be maintainable by the company or organization that uses it. New modified versions of CAE-software components continue to appear and new in-house programs are developed to meet new requirements. The IFE is the glue between them and must continuously be updated with knowledge of changes in the environment. It must provide the facilities of the CAE-system to the engineers, without encumbering them with otherwise necessary routine work.

Knowledge-based systems are suitable for the IFE-task because of the solution transparency and the explanation possibilities. Any erroneous behavior must be easy to detect and correct immediately. In general, errors generated when using a CAE-system with support from an IFE are much less frequent than without it.

To build an IFE application for a given CAE-system, a suitable meta-model for its engineering domain must be found. The meta-model serves as specification of the knowledge structures needed for keeping track of all details and relations. Besides the target engineering domain, models of the CAE computing domain (i.e. a model of the execution environment for the IFE) must also be present to keep track of files, programs, substructures in files etc. The number of details to be specified becomes large, even if the domain is simple and the target CAE-system just consists of a few programs. To cope with the complexity it is very important that the IFE keeps track of all the relations itself. It must have a good user interface that can visualize all aspects to assist the engineer responsible for system maintenance to maintain a good understanding of its workings. Changes to the IFE will not be made very frequently, so good support is needed for recalling its workings. The IFE should be its own documentation. Paper documentation concerning changeable aspects would introduce an considerable update bureaucracy.

To summarize, visibility and incrementality have a similar importance for IFE applications as they have for general development environments for knowledge-based systems.

6.3 An IFE for damage tolerance design on aircraft structures

In the collaborative project, the target CAE-system for studies is a set of programs developed and used by the SAAB Aircraft Division³². The domain is damage tolerance design, including crack growth calculations for predicting the life-time of aircraft structures. An IFE-prototype has been developed that supports the generation of input-files to a program³³ that does the actual crack growth calculations. With the help of an informative graphical user interface, the engineer can, for instance, enter the information needed for specifying one crack growth calculation. This includes, among other things, specifying a crack geometry (See Figure 3 on page 43). The IFE checks the input for completeness and permissibility, and then generates an input-file consisting of command-sequences, parameters, etc. to the analysis program.

Orsborn from the Mechanical Engineering Department has developed the domain model of the target domain, while I have concentrated mainly on user interface and general domain modelling questions. At one development stage, the prototype contained more than 900 objects of which 350 were rules. During a consultation a case-model is built, usually consisting of about 70 instances of concepts. The implementation has been performed in KEE on an Apollo DN3000 workstation.

6.4 Method

The work on the IFE problem began with studies of similar work (e.g. [Bundy 84]), the domain [Fredriksson 86], manuals for both the CAE software [SAAB 90][Ansell 88] and the KEE-system [Intellicorp 87]. Following this a collection of requirements for an IFE was formulated. Iterative development of prototype knowledge-bases was very helpful for both acquiring a better understanding of the problem, and discovering and documenting the concepts and relations of the domain.

The knowledge-base browsers worked as tools for developing ideas. Many insights were gained by the ability to study the graphical structure of the knowledge-base as it emerged. Although after some time when the conceptual model began to stabilize, it was discovered that support for flexible visualization and manipulation of models (e.g. instantiations of our meta-model) was lacking. The user interface tools we found in KEE only supported visualization and interaction with static concept structures. And these interfaces had to be written by hand. Recall here that there is a difference between the user interface for the end-user and for the KBS developers. The support provided for building the end user interface was in

32. Acknowledgements to SAAB for support with domain expertise, software and documentation.

33. The program is called CAFE for Computer Aided Fracture Engineering [SAAB 90].

some respect adequate, but the interfaces for the knowledge engineering work we were doing could have been much better.

Our problem solver generated case models (e.g. a test case model for a particular crack growth calculation) dynamically. These needed to be inspected for debugging and gaining better insights in what further relations to model. The KEE-browsers were slow and did not display the interesting domain relations and attributes of the case models in an efficient way for inspection and manipulation. If such support had been available, our work on finding an adequate meta-model of the domain would have progressed much faster.

The lack of adequate visualization tools was early recognized as a problem of general importance. Therefore it seemed to be a good idea to work on finding a satisfactory solution. The resulting user interface software architecture (UISA) was first described in [Johansson 91], and an improved version is given in section 18.2 on page 151. A prototype was developed for verifying and tuning the UISA. It was built in a commercial object-oriented Smalltalk-like programming environment, Actor, that runs on a PC under Microsoft Windows³⁴[Duff 86][Whitewater 90]. The PC prototype was developed far enough to verify that there were no logical bugs in the UISA-design, that it could be made fast enough, and that a selection mechanism and other supporting functions could easily be incorporated. The UISA-part of that prototype contained 18 classes of reusable user-interface components, and the test-application about 30. The system-supplied classes used for abstract datatypes, window management etc. are not included in these figures. Later, parts of the software architecture in the PC-prototype were re-implemented in KEE to support the IFE-prototype. Unfortunately this implementation was too slow to provide any practical support for incremental work with dynamically generated instance configurations of the meta-model, but it works well for demonstrating the IFE's end user interface for the domain engineer. Figure 3 on page 43 shows a user interface object for editing the attributes of an instance of the class `CRACK_GROWTH_GEOMETRY_MODEL` that is modelled within the knowledge-base. The functionality of this user interface object has been implemented with the UISA. The diagram supports the domain engineer in recalling the names and the purpose of the different attributes of the geometrical model. The fields below the diagram make the values of the attributes directly accessible to and editable by the domain engineer. The smaller figure above shows a brief sequence of the command input file for the fracture analysis program. Sub-sequences such as the one given are generated automatically by using information stored or computed in knowledge-base objects.

34. This was summer 1990. At this time, the state of software development environments for window systems under SUN-UNIX was rather confused. There was actually only one development environment that fulfilled the requirements of incrementality and stability for the development work. But the windowing software of this Smalltalk-80 based system was also in danger of becoming obsolete in the next version so it was ruled out.

6.5 Discussion

The development towards an acceptable domain language in the form of a meta-model is an iterative and time-consuming process when it is hard to gain access to a domain expert who has thought in detail of the target domain at that level of abstraction before. One knows what instances of target domain models should be able to describe, but with help of which concepts i.e. how should the domain be structured into objects, properties of objects and object relations? Which inferences should be made over the object structures i.e. what is the deep knowledge? Experiments with formulations of concept structures in prototype knowledge-bases have to be made in order to gather information and form concentrated concepts to reason about.

For our domain many meta-models seemed to be adequate. Transformation of one meta-model to another seemed to be possible, where the transformation implied moving attributes and relations between certain object classes. The author's advice is that when one discovers that such transformations seem possible, it is better to stick with one choice of the affected part of the transformable meta-model and proceed with modelling the other parts. Otherwise the danger is to rebuild and change the prototype knowledge-base more because of what is currently pre-occupying the knowledge engineer than of what later will prove to be practically useful. When the knowledge-base is large (i.e. the meta-model contains more than fifty classes), it may not be obvious whether one is putting effort into rebuilding the meta-model of the prototype knowledge-base into different almost equally valid transformations, or if one is actually proceeding with the modelling process. Progress is recognized when conceptualizations are found that significantly simplify the overall structure of the meta-model without losing any functional properties. In many cases, the meta-model can be greatly simplified by finding a "correct" set of "orthogonal" concept formulations. Some transformations have to be experimented with in order to reveal such simplifications. However, for many parts of our meta-model there was no such adequate simple and clean structure because the target domain actually **is** complex. Understanding which meta-model is the better has to come from experiments, i.e. by testing how well each meta-model applies for modelling a set of test cases. The problem is that no such test case can be prototyped without help of an at least to some extent completed meta-model. Therefore, hesitation in designing parts of the meta-model may delay the modelling process more than a temporarily satisfying design decision. Inconsistencies and weaknesses in such temporarily designed parts of the meta-model are discovered and can be redesigned when the test cases are built. Trying to figure out a good meta-model without help from experiments that focus attention on real problems is difficult. This is particularly the case when the domain is complex, making it almost impossible to pay attention to all the interacting parts at one time.

Despite these arguments in favour of completing a "prototype" meta-model

in order to detect flaws in it, if the development environment gives insufficient support for experimenting with test-case instantiations of different meta-models, one test example may cost weeks of tedious implementation time. Therefore it is natural to hesitate and try to solve the problem on the meta-model level (with perhaps insufficient understanding of the real influencing factors) in order to “get it right” from the beginning and save implementation effort.

Creativity and productivity seemed to be highly correlated to the speed of the knowledge-base browsers. This was best demonstrated by a certain user interface functionality that took less than two days to develop in the Actor programming environment. To re-implement it in KEE on the system configuration we had took more than one week. Each part of the UI-functionality was easy and straightforward to implement. The problem was in establishing correct symbolic references between the different interacting parts of code. Without high-speed browsers, the re-implementation was frustrating work.

Although rather slow, KEE’s graphical knowledge-base browsers were invaluable for recalling work with long interruptions (months) in-between. In most cases it took much less than an hour to recall the working context and be productive again.

The possibility to refer to, and operate on, objects by pointing is invaluable. We have frequently been working with object names such as `GENERIC_CRACK_GROWTH_GEOMETRY_MODEL_MODELLING.169`. It is questionable whether abbreviations can be remembered while performing creative incremental work on many parts of a knowledge-base that contains almost a thousand different objects.

To conclude the discussion we can say that enhanced possibilities in accessing, inspecting and operating on instance configurations (i.e. test cases) of different meta-models would have a strong positive influence on the productivity of knowledge engineers and domain experts. Support for building interaction tools of this kind should be provided by the KBS development environment since they require considerable skill and effort to implement.

Part I of the thesis ‘Properties of Knowledge-Based Systems for Engineering’, has described the vision of future knowledge bases we had in 1991. It guided and motivated the development of the systems described in Part II and the theory presented in Parts III and IV.

Subsequence of the input file to the CAE-program:

```
GEOMETRY
GEOMETRY = 3DIM_330(A,1.27,1.27,100,5.0,10.0,0,Y)
EXIT
```

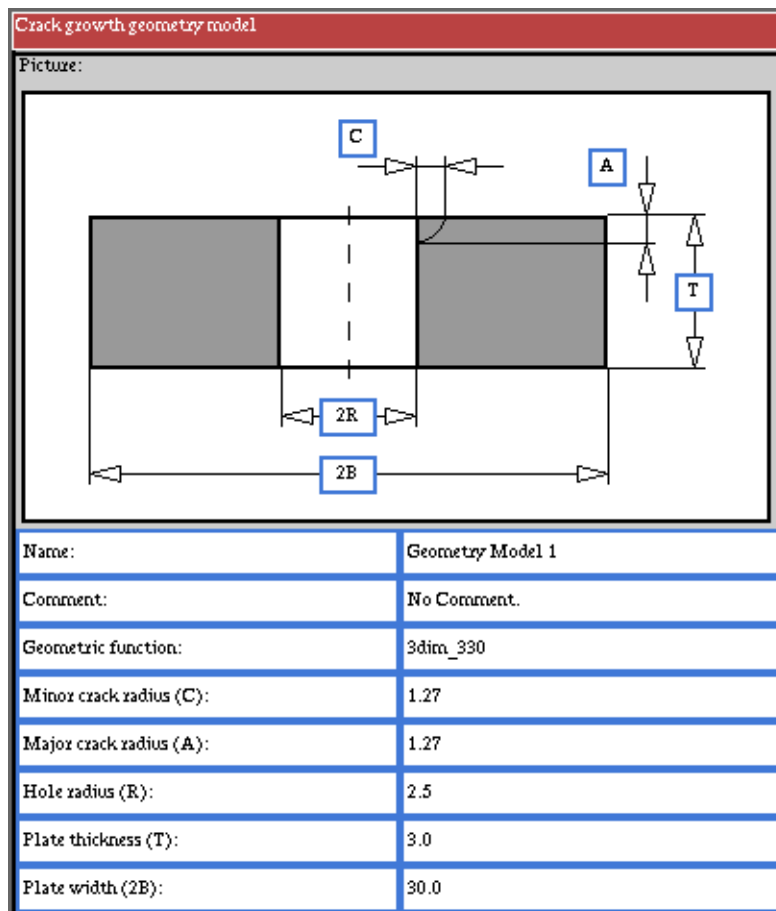


FIGURE 3. A user-interface view of a knowledge-base instance of the class `CRACK_GROWTH_GEOMETRY_MODEL`. The view contains only one object editor, that in turn holds eight fields which allow editing of the attributes of the knowledge-base instance. The KB-instance is used to hold facts about a crack growth geometry model. The smaller picture shows a command sequence that is generated from the intelligent front end (See 6.3 on page 39) as input for a CAE-program that performs crack growth calculations.

Part II Experience from Development Environments for Product Modeling Systems

7 Introduction to Part II

7.1 Overview

This part describes practical experience from developing a product modeling system using the iterative KBS development process described in chapter 5.

Chapter 8 describes ProCAD, a product modeling system for industrial turbine and power plant system design. ProCAD provides an example of what a product modeling system is, and why it poses certain requirements on the development environment and its underlying software components.

Chapter 9 describes the software architecture for the major components in a PMS, and the meta-database based development platform that was used for ProCAD. This chapter is a revised version of a paper that was originally presented at the ADB94 conference [Johansson 94].

Chapter 10 describes the source code generation technique introduced in chapter 9 in more detail. This technique significantly speeds up the implementation and maintenance of the product modeling system. For the system developed at ABB, the generated source code was about 10 times the size of the source code generators.

7.2 Readers guide

Chapter 8 can be read to gain some understanding of the engineering domain of industrial turbine system design, and the complexity of the functionality that this type of product modeling system must provide.

Chapter 9 is essential for the understanding of the meta-data-based software engineering approach and the utilisation of the theory introduced in Part III and Part IV.

Chapter 10 is not necessary for understanding the theory presented in Parts III and IV, but contributes with an engineering approach to source code generation that has proven to be of significant value in practice.

8 ProCAD - A Product Modeling System for Power Plant System Design

8.1 Introduction

ProCAD is a product modeling system for power plant system design. It is the result of a joint research and development effort of the Department of Computer and Information Science at Linköping University and the power plant engineering and manufacturing company ABB STAL.

ProCAD is used for the system design of steam turbine, gas turbine, combined cycle, and pressurised fluid bed coal combustion (PFBC) power plants. A combined cycle power plant uses both steam and gas turbines to get a better exchange in the transformation of the energy produced from the burning of fuel into electricity. The coal combustion technique in a PFBC power plant, in combination with advanced smoke cleaning, provides an environment-friendly alternative to traditional coal power plants.



FIGURE 4. Photo of a steam turbine power plant from ABB STAL.

The product model of a steam- or gas turbine plant contains descriptions of many hundreds of functional components: turbines, electrical generators, pipes, pumps, valves, instruments, heat exchangers etc. The power plant system designer can interact with the detailed information in the product model database directly from the drawing environment in the CAD-system.

The development effort started in 1991 with a pre-study of the information structures, processes and dataflows involved in turbine plant production. The borders of functionality for power plant system design were roughly determined. The product modeling system was to provide an application for drawing process and instrumentation diagrams of the power plant systems, enable entering of data for components on the diagrams and generate various lists and reports from the product model database. A series of prototypes, based on an object-oriented domain model, were developed and which demonstrated the concept.

The prototypes were tested with the design of real plants quite early. This led to requirements on an efficient CAD-drawing environment. In the fall of 1992 cooperation with S&S Systemutveckling was started. They had long experience of developing AutoCAD applications for different domains. By basing the ProCAD prototypes on their commercial drawing environment for 2D-diagrams, a professional environment was achieved.

In 1993 the system was put into production. At that time, the domain model for the product model database had been revised and implemented in eight successive prototypes.

In 1994 cooperation with the compiler development company Softlab was started. They augmented the development environment with source code generators for deep copying functions for product models in the database.

Since then, two revisions of the domain model and the product model database has been made on the production version of ProCAD. The revisions included conversions of the developed production power plant product models to the new database schema.

ProCAD is now regularly used by about 50 engineers at ABB. More than 80 steam- and gas- turbine power plants of the size of 10-100 MW and several PFBC power plants have been designed using the system.

8.2 The ProCAD system architecture

Figure 5 on page 51 shows the system architecture of ProCAD. The product model database manages objects and the integrity of the engineering structures in the product models for the plants. The implementation has been made on a relational database management system (DBMS) from Sybase. Through the browser-, CAD- and 4GL³⁵- client applications, the engineers can inspect the product models, and create, update, and copy various complex product structures.

35. Fourth Generation Programming Language

The plant browser application allows a project manager to navigate and manipulate the major structures of the plant models. He may, for instance, build the structure of a new plant by creating the objects that constitute its major “part-of” structure. Some of the objects represent different systems in the plant, such as turbine system, lubrication system etc. Within the plant browser the project manager can assign user access rights to the substructures of, for instance, a turbine system. In this way he can delegate the task and authority to develop the details of systems to different system engineers on his staff.

A turbine systems engineer can use the AutoCAD application to draw diagrams which specify how different functional components in a turbine system are connected. These Process and Instrumentation Diagrams (P&ID) show how the steam is processed through different turbine components, and where instruments are placed that measure pressure, temperature etc. There is an interactive interface between the P&ID CAD-application and the product model database. The systems engineer can, for instance, select an instrument on the P&ID drawing and bring up a form that provides read and update facilities of the data for this instrument object in the product model database.

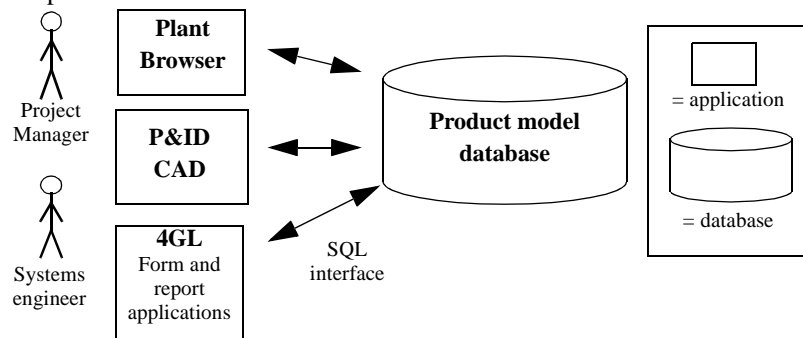


FIGURE 5. Architecture of the ProCAD product modeling system for power plant system design.

Client applications for data entry and report generation are developed using modern 4GL-tools with an SQL-interface. Currently, Microsoft Access is used for user-friendly data display and generation of reports.

8.3 Outline of the ProCAD domain model

The domain model is a precise design specification of the objects and structures in the product model database. Figure 6 shows an overview of the major objects and structures in the ProCAD domain model. The boxes represent distinct classes of objects, and the lines represent relationships, with intervals of *min..max* at the ends telling the minimum and maximum number of object instances of the class that may participate in a relationship. A star “*” means infinity. A black diamond at for instance the

Plant class means that a plant is composed of blocks. The overview only shows 10 classes and 10 relationships. The actual domain model contains about 50 classes and relationships, and more than 400 attributes.

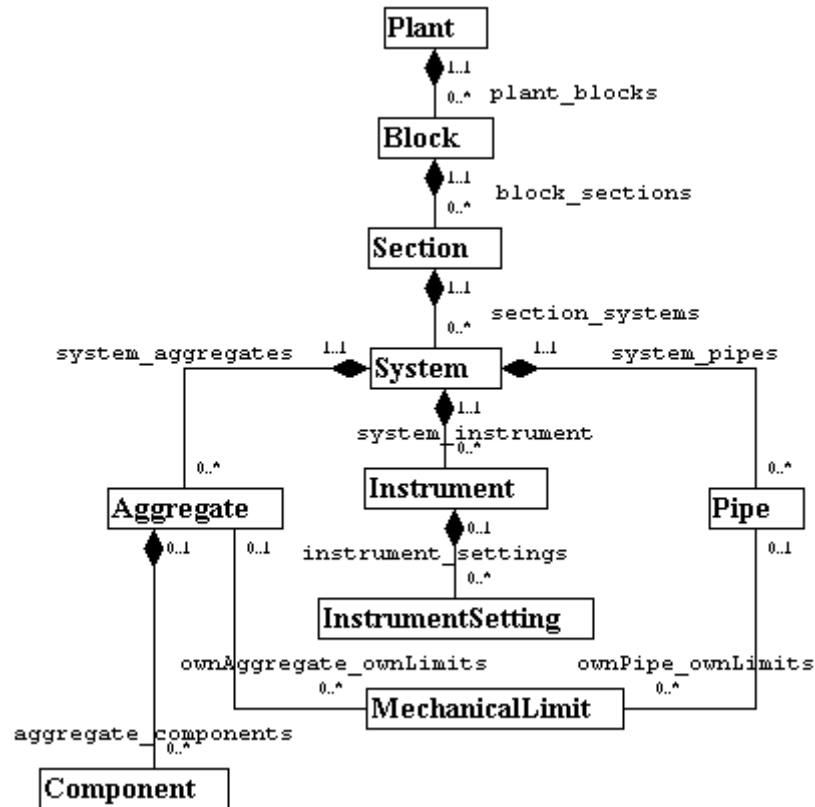


FIGURE 6. Outline of the domain model for ProCAD.

8.3.1 Plants, Blocks, Sections and Systems

The plant objects contain data about a plant. There are, for instance, attributes like catchword, customer_name and remark. A plant may consist of several blocks. A block is typically an individual building containing a complete operational plant. The customer may need a multiple of the power of an existing plant design, and it is sometimes advantageous to build several copies of existing proven designs instead of developing a new one.

A section can be a major functional unit within the same power plant, for instance a boiler or a steam turbine. It may also be a functional category of equipment such as internal- and external electrical systems or systems for cooling. Each section is divided into several systems. For a steam turbine there is a turbine system, a lubrication system, a generator system etc.

8.3.2 Aggregates, Components and Pipes

An aggregate is a functional unit within the system. It may be a heat exchanger, a pump, a valve, a tank, etc. Aggregate objects within the product model database have attributes for different designations, description, size, weight, location, various references to detailed drawings etc. Aggregates may be decomposed into several components of different types.

A Pipe object contains attributes for different designations, location, dimensions, medium, test_group, drawings etc.

8.3.3 Mechanical Limits

A mechanical limit documents the interface between an aggregate and a pipe. The object contains attributes for location, connection standard, references to connection type drawings etc.

A special type of mechanical limit is a delivery limit. Such an object documents an interface between systems developed by different system engineers or suppliers. Values for medium, heat, pressure, flow, dimensions and location are specified for the delivery limit, and serve as a contract on what the different system engineers or construction partners have agreed upon.

8.3.4 Instruments and Instrument Settings

Each system has a set of instruments that supply the control system with information. They measure, for instance, pressure, temperature or vibrations. An instrument may have several instrument settings. A setting documents a threshold when a certain control signal should be activated.

8.4 The plant browser application

The plant browser shown in Figure 7 allows a project manager or systems designer to browse the hierarchy from Plant down to System. With the correct privileges the hierarchy can also be edited. The first listbox shows the plants in the product model database. By selecting one, its corresponding blocks are listed in the block listbox. When the user selects a block or section, its corresponding sections or systems are displayed in the listbox to the right.

A product model database can contain hundreds of plants and thousands of systems. After the correct plant is selected, it only takes three clicks to gain access to the detailed data of any of its systems on a form.

The plant browser also provides functionality for deep copying an entire plant or a selected set of systems, including registered CAD-drawings that contain references to objects in the copied product model.

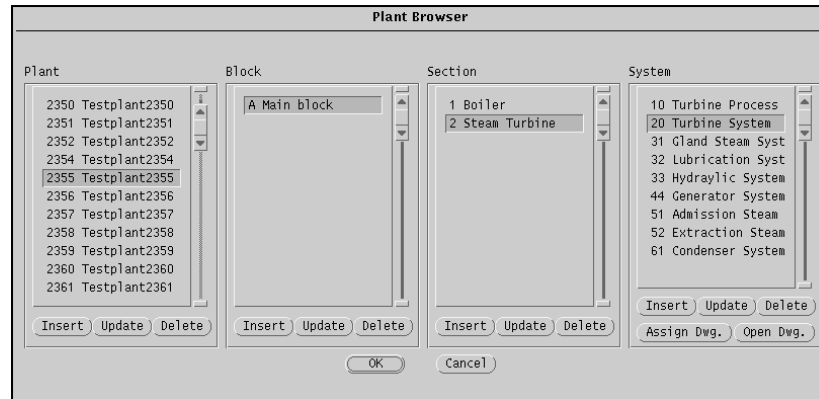


FIGURE 7. The plant browser.

8.4.1 Project-specific attribute assignment

Various object types in the product model, such as aggregate, pipe, instrument etc. each have their own predefined attributes from the domain model. A new power plant project may require additional project specific attributes and default values for its different object types. A certain country may, for instance, require a special certification on the pipes according to its national standard. References to this standard must be included in the product model so that they can be extracted on reports and documentation. These kind of project-specific fields can be configured individually for each plant by its technical project manager without support from any computer specialist. The configuration functionality is available in a form that appears when a plant is double-clicked in the plant browser.

8.4.2 Assignment of update access rights

A gas- or steam turbine power plant development project may take from 10 months to more than a year from when an order is placed by the customer until the plant is put into production on site. The same PMS database may be used for the development of twenty orders in parallel. For each project, different engineers are assigned to develop different substructures in the product model of the plant. To support quality assurance and borders of responsibility, access privileges to the substructures must be configured individually by the technical project manager for the engineering of the plant.

A project manager can configure which database users can update a system in a certain plant without support from a database administrator. This is done on a form accessible from the plant browser.

8.5 The P&ID application

The process and instrumentation diagram CAD-application is built on AutoCAD. Figure 8 shows how the application looks on the screen while editing a turbine system.

The application window is divided into four areas: the pulldown menu with “File Edit” etc.; the drawing area, with an additional menu to the right; and the command line at the bottom where commands and LISP code can be entered directly.

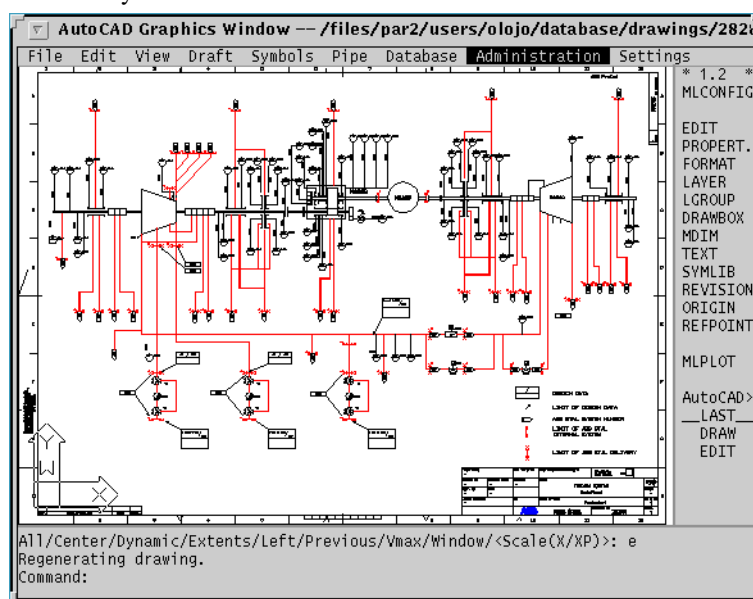


FIGURE 8. P&ID CAD-application editing a turbine system drawing.

The menus serve as front ends to the textual command language and lisp programming language. The menu choices call P&ID-application specific macro functions. All normal tasks of the turbine system engineers are supported by such macros.

Figure 9 shows a zoomed-in region of a P&ID. It contains a heat exchanger, two valves, two instruments indicating the position of the switch valve and two pipes with flags. By selecting a menu-command and pointing to any of these components, the engineer can bring up a form from which data in its corresponding object in the product model database can be edited. Figure 10 shows the form for the valve “LAB30AA002”. There are more than 30 fields on this form. Some of them, such as designations and description are entered early in the design phase. Others may not be applicable for the aggregate in question and are never specified. The fields and listboxes on forms will change as new versions of the domain model are implemented. The drawing environment itself is insensitive to most

such changes.

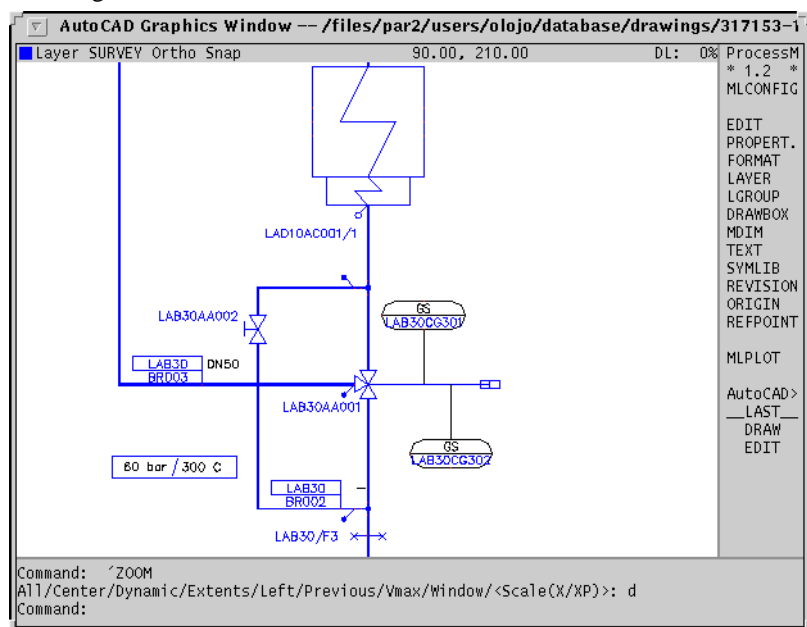


FIGURE 9. A zoom-in of a P&ID drawing.

The form is titled 'Aggregate' and contains various fields for editing data. The fields are organized into sections: 'highid', 'lowid', 'added', 'modified', 'status', 'release', 'abb_designation', 'customer_designation', 'kks_designation', 'description', 'aggregate_type', 'remark', 'abb_documentation', 'documentation_reference', 'main_group', 'central_controlled', 'control_code', 'control_location', 'electrically_connected', 'function_lettercode', 'function_lettercode_custom', 'function_lettercode_kks', 'noise_level_max', 'registration_of_duty_time', 'abb_article_no', 'technical_specification', 'type_drawing', 'object_group', 'location', 'dimension_drawing', 'height_mm', 'length_mm', 'width_mm', 'weight_kg', 'supplier_documentation', 'customer_plate_text', 'ID-plate_size', 'ID-plate_text', and 'Project specific fields'. The 'Project specific fields' section includes a table with 'Label' and 'Value' columns.

FIGURE 10. A form for editing data for an aggregate in the product model database.

8.5.1 Drawing environment functionality

The drawing environment has a rich set of functions for editing the graphics of a P&ID. It also provides a library of drawing frames for different purposes. The drawing frames have a predefined set of layers. Layers that are visible on the drawing behave like a stack of transparencies.

Different customers require different designations on the different component types, depending on what is specified in national standards. Besides the customer designation system, ABB maintains two internal designation standards. One that is there for historical reasons and was used by the previous STAL-Laval company, and KKS (Kraftwerks Kennzeichen System) which is a European standard for designation of power plants.

By keeping the designation systems on separate layers in the drawings, the same drawing can be plotted for different documentation purposes, showing only relevant information. The layer management functionality also enables the same drawing to be displayed with hidden details. This is useful for the sales-, tender and documentation process.

To be able to provide all combinations of information presentation, a P&ID drawing has about 60 layers.

8.5.2 Symbol library

From the “Symbols”-pulldown menu, the designer has access to more than 300 symbols. After selecting the main category from the pulldown menu, the appropriate symbol is chosen from a dialog box as shown in Figure 11. The symbol can be selected either by the textual name in the listbox to the left, or by its picture. Some additional symbol documentation can also be accessed directly from the dialog box.

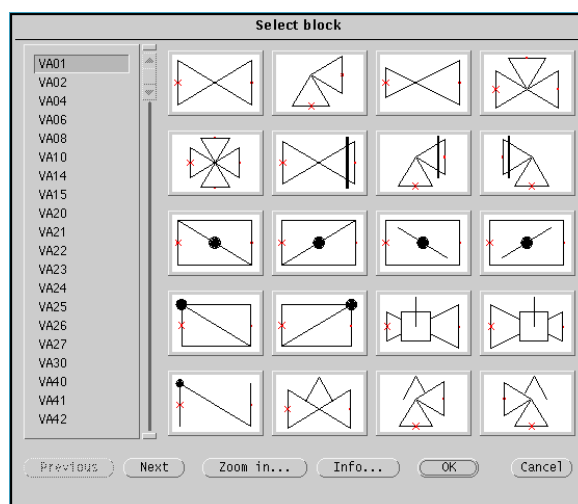


FIGURE 11. Menu providing 20 of the more than 300 symbols in ProCAD.

8.5.3 Database functions

The P&ID application provides a set of functions for managing the product model database (PMDB). Besides logging in to the correct database from the CAD-system, it provides functionality for listing and editing various categories of product model objects, and assigns and updates connections between symbols on the CAD drawing and objects in the PMDB.

Since the CAD-drawing and the product model database can be updated independently, the consistency between the two must be maintained. The menu command macros aim to support this, but using low-level CAD-commands from the command line it is easy to add or erase a symbol without performing the corresponding operation on an object in the PMDB.

To make sure that the drawing and the product model database are consistent, checking functions are available. If a symbol has no corresponding database object, the checking function marks this symbol on the drawing with a special report symbol that is placed on a message layer (see Figure 12). The checking function allows the designer to directly identify which ones of maybe one hundred symbols on the same drawing are not connected to an object in the product model database.

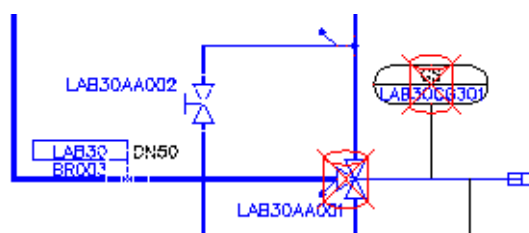


FIGURE 12. Graphical database reports directly on the drawing. The symbols marked with a crossed over database have no corresponding object in the product model database.

8.6 4GL applications

Several 4GL applications have been developed in Microsoft Access. One report generation application provides more than 50 different reports that are generated from more than 40 specialized tables or views in the product model database.

The reports are typically very information-dense, and prepared for various engineering, manufacturing, assembly or maintenance tasks.

The report application has some support forms for browsing the product model database, and select the appropriate data and report formats for display or printout.

There is also an application for form-based entering of data directly into the product model database.

8.7 Measurements for product models

More than 80 plants have been designed using ProCAD. The figures sketched here are presented to give an idea of the size and complexity of the P&ID drawings and plant product models. The figures are derived from one production database that contained many small gas turbine plants.

A typical plant has one block containing three sections. The average section consists of 4 systems, but this varies. About a third of all sections are implemented by another subcontractor and only entered in the PMDB for documentation purposes. Otherwise, a section with more than 10 systems is not unusual. Systems are very different, but some kind of average system contains about 20 aggregates and 15 instruments. Half of the aggregates have one to three components, and about half of the instruments have one to three settings, but most of them only one.

The average number of pipe objects in a system is 12, but this is also very varied. Systems with more than 30 documented pipes are not unusual, and about two thirds of all systems have no documented pipes at all.

MechanicalLimits document an interface to other systems and to external subcontractors. About half of the systems contain documented mechanical limits, and these have on the average seven mechanical limits. About three percent of the aggregates are connected to a mechanical limit, and about five percent of the pipes.

In summary, a typical plant product model contains one block, four sections, a little more than ten systems, some hundreds of aggregates and instruments, and about fifty documented pipes.

8.8 Summary

ProCAD is a PMS for turbine and power plant system design that has been developed in a cooperation project between the Department of Computer and Information Science at Linköping University and ABB STAL. The work started in 1991. Since the system was taken into production in 1993, more than 80 plants have been designed.

The PMS consists of three client applications connected to a server of the product model database. The plant browser applications allows a project manager to deep copy entire plants or selected sets of systems, to specify project specific attributes for a particular plant and assign update access rights to individual system designers. The P&ID application provides advanced drawing functionality, a symbol library with more than 300 symbols and database functionality for checking the consistency between CAD-drawings and the product model database. The 4GL-report application provides more than 50 different reports.

Plant system product models for industry turbines vary in size. In the sample database described in section 8.7 a typical model contains about 1500 objects.

9 Using a Meta-Database to Implement Product Modeling Systems

9.1 Introduction

A product modeling system (PMS) is an object-based computer integrated development environment for a specific class of advanced products. PMSs have a strong potential for increasing engineering productivity and ease the management of complex high-tech products.

There are however three major obstacles that severely prevent the development of long-term successful PMSs.

- I The complexity of the product itself.
- II The extensive amount of software engineering skills needed for design and implementation of a PMS.
- III Computer technology is under continuous development and a PMS implementation will become obsolete within a few years.

To overcome these, we have taken an approach which is depicted in Figure 13. The key idea is to separate the knowledge of product experts and software engineering experts with a clean and small interface. In our case, this is achieved by an object-oriented domain model (OODM) which can be represented graphically by object model diagrams.

The core of a product-specific object-oriented CASE model is the OODM. The OO CASE model serves as a PMS object system design specification which is expressed in a graphically representable formal language. Its semantic expressiveness is similar to the data definition constructs of the product data modeling language EXPRESS [EXPRESS88][STEP92a]³⁶, but provides more flexible facilities for maintenance, documentation and source code generation.

36. EXPRESS has an object-based flavour and is a fundamental part of the STEP standard ISO 10303, "Product Data Representation and Exchange" which has its roots in the early 1980's. The purpose of our object-oriented domain models is primarily to accelerate the development of practically useful PMS-prototypes.

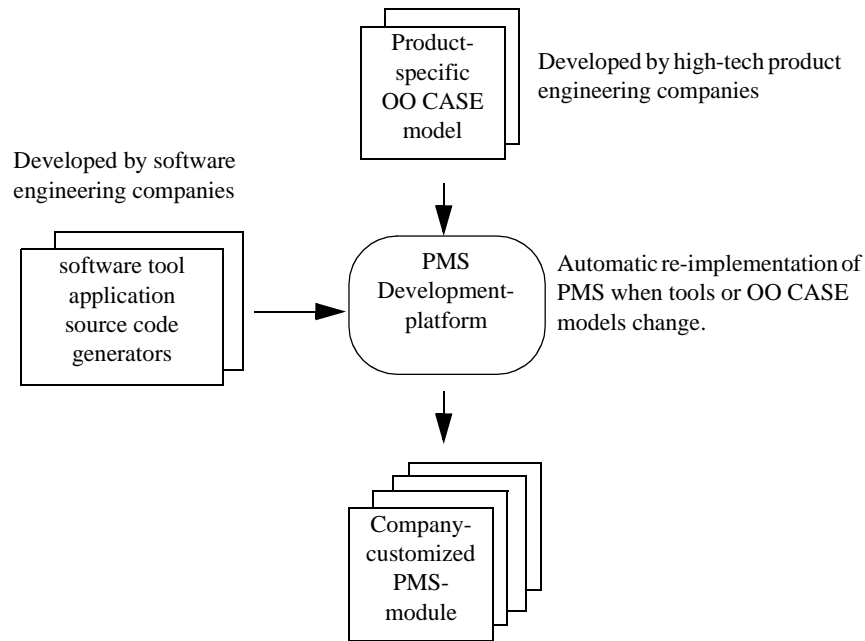


FIGURE 13. Approach to overcome the three obstacles to PMS development.

Our object-oriented domain models and object model diagrams have similarities with Chen's ER-diagrams [Chen76], but have their formal ground in the thorough theoretical work on infological models performed by Sundgren [Sundgren73] [Sundgren89] [Sundgren92]. We use a compact graphical notation for our object model diagrams for describing classes, relationships and attributes, see Figure 19 on page 73. Object model diagrams and reports generated from an object-oriented CASE tool can be understood by both product experts and software engineering experts and serves as communication medium between the two engineering disciplines.

Software engineering companies who develop tools such as databases, CAD-systems and user interface toolkits can develop knowledge in how to transform an OO CASE model into a working implementation for their own specific software tools.

Once the transformations have been formalized in a programming language, software engineering companies can package and sell their software engineering knowledge in the form of application code generators.

If the domain model of the PMS development platform in Figure 13 can be standardized, it would enable many software tool companies to implement applications code generators that would be applicable to many engineering companies' PMSs, and thus open up a competitive and developing market.

This chapter first describes the above-mentioned obstacles to PMS-development and then presents the architecture of the major software components in a product modeling system in more detail.

The meta-database based architecture of our PMS-development platform is described, and exemplified by an implementation of a small object-oriented domain model.

Section 9.7 describes the domain model of the meta-database in more detail. This section should be read carefully, because the theory presented in Part III assumes this as pre-knowledge.

The chapter concludes with some experience from how we used the development platform during the development of ProCAD, and some conclusions.

9.2 Problems while developing complex product modeling systems

It is no easy task to develop a PMS which can be maintained as a versatile tool for a high-tech product engineering company. The products contain thousands of details, and engineers from many disciplines must cooperate using different engineering models that have complex relationships.

Each product must be customized according to customer requirements and the special conditions at the location of operation. Once delivered, the product must be operated and maintained during a lifetime of several decades.

The following three sections give some additional details about why it is hard to develop a long-term successful PMS for this kind of product.

9.2.1 The complexity of the product itself.

In addition to the different interrelated engineering models there are the aspects of scale and development time. A typical steam turbine plant, for instance, contains thousands of articles which are assembled and interconnected in a complex fashion. A project can take years to complete and may involve a hundred people who create and use different sorts of product-related information.

Thus, the PMS must provide security functionality for authorization of access privileges for different user categories. It must also allow privilege assignments to individual users who are to work on a subset of several parallel product models that belong to concurrent development projects. For example, a turbine systems engineer must be able to manipulate the turbine systems for the three turbine plants he has been assigned to, but not the other fifteen that are being developed in parallel and stored in the same database.

ISO 9000³⁷ puts demand on a secure system for quality control. Hence, the PMS must provide configuration and version management including locking of inspected and approved submodels to secure certified subdesigns from unauthorized changes.

9.2.2 The design and implementation of a PMS requires extensive software engineering skills.

The software engineering process of a PMS includes designing an adequate OODM for the product. This is a heavy, iterative process. Several engineering disciplines must work together and no single person has a complete understanding of all aspects of the product.

Without adequate CASE-tools the requirements- and design specifications become large and difficult to manage and understand. This is a threat, since the design documentation has to be iterated efficiently with knowledgeable product experts who are a scarce resource and do not have much time. If the project loses their interest and high-quality influence, it will severely affect the acceptance of the resulting PMS.

A PMS may need to integrate CAD, CAE, CAM³⁸ and economic information systems. This requires software engineering skills in CAD, databases, user interfaces, different programming languages and so on. Software engineers with this comprehensive amount of software engineering knowledge are rarely available in enough quantity at the engineering company itself. Using software consultants without knowledge transfer to the engineering company can make it dependent and vulnerable.

9.2.3 The fast development of computer technology.

This is a problem which all developers of large-scale software systems fight with today. New advances in software and hardware technology will in a few years render a developed system obsolete.

It is hard for an engineering company to decide which CASE-tools, database systems or user interface development toolkits to purchase. An investment is expensive, and often ties the company to a certain set of software products for a long time. The software maintenance task quickly becomes a major problem. Levering a PMS to a new generation of computer technology may become an extremely expensive project, especially if its design is stored in old-fashioned “closed” CASE-tools.

37. ISO 9000 is a standard for quality assurance (QA). A product development company can be certified according to ISO 9000. To get the certification, the company has to provide proof that they manufacture their products according to procedures which guarantee an adequate QA. Part of the proof is given in the form of documentation of the QA-procedures, and these have to fulfill the requirements stated by the standard.

38. Computer Aided Manufacturing

9.3 A generalized software architecture for PMS

Figure 14 shows a generalized architecture of a product modeling system for use in a concurrent engineering environment. The architecture described assumes complex products with long development times and the need for locking of checked and certified subdesigns for quality assurance and prevention of accidental changes due to human mistakes.

The product model database manages object-oriented engineering structures which have a large and complex database schema. Different CAx³⁹-applications need to manipulate and interact with these structures on an efficient abstraction level. Engineers must also be able to browse the product model database interactively, and create, update, and copy various complex product structures.

A production version of a PMS needs support for project management. This can be provided by a project browser application. It should enable a project manager to choose different projects, navigate and manipulate the major structures of their product models and assign *user access rights* to individual substructures so that the task and authority to develop the details can be delegated to different engineers.

To avoid accidental unauthorized changes to data by user-developed 4GL applications, the access control system on the object and substructure granularity must be implemented within the product model database. How this is done is explained in section 9.3.1.

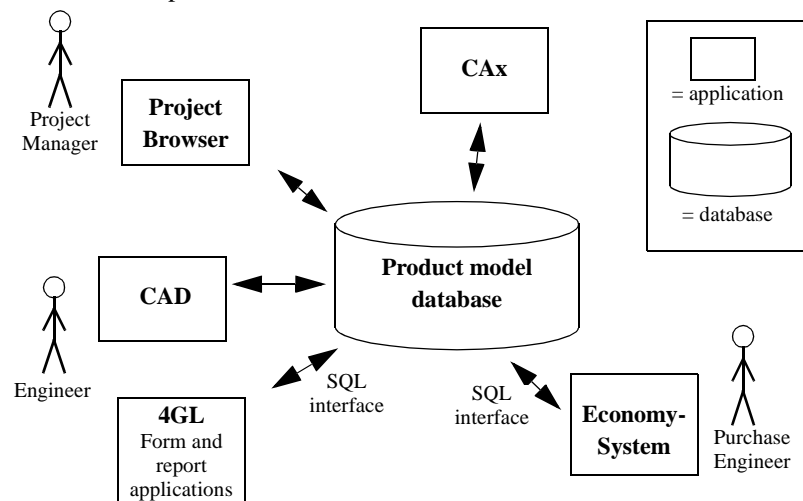


FIGURE 14. Sample product modeling system.

39. CAx is a common name for CAD, CAE and CAM which stand for Computer Aided Design, Computer Aided Engineering and Computer Aided Manufacturing.

An engineer can use a CAD application for the graphical design of different parts of the product model. Different geometrical objects in the CAD-drawings can have attributes attached that refer to objects in the product model database. Similarly unique drawing item identifiers can be stored in attributes of objects in the PMDB and thus refer to geometrical objects in a CAD-drawing.

Traditional client applications for data entry and report generation can be developed using modern 4GL-tools with an SQL-interface if the chosen DBMS for the PMDB provides it.

There are other applications that need direct access to information in the product model, for instance computer aided engineering (CAE) and manufacturing applications (CAM). This is symbolized by the right side CAx- and economy system application in Figure 14.

Integrating logistics and purchase systems with the PMS can save large amounts of boring and error-prone routine work, and provide economic decision support to the engineers when choosing between different design alternatives. Today careful economic analysis of a design is often omitted because of the cost in time and effort to gather the needed economic data.

The following sections describe the software layers in the product model database, the CAD-application and the browser applications. Structuring the software into layers enables some of them to be generated automatically from the domain model. Building the manually written applications around generated layers makes their source code easier to understand and more robust against later changes in the domain model.

9.3.1 Software layers in the product model database

Figure 15 shows the software layers in the product model database. The bottom layer implements the database manager, which provides, for instance, persistent storage, concurrency control, transaction management and basic authorization of multiple users. This may be a commercial relational database or a special PMDB client server application with multiple user access control implemented on a commercial C++ database.

The second layer provides a data definition language, a declarative query language and preferably a library of abstract data types that can be used for implementing various types of relationships and data structures in the domain model.

In the future these two layers could be replaced by an SQL3 database.

Most DBMSs do not provide functionality for access control on individual rows in tables or individual objects. This functionality is necessary if a project manager is to be able to configure access rights to different users on individual objects without support from a database administrator. The configurable access control system can be implemented using triggers in a relational DBMS, and calls to pre-condition checks in client callable

methods in a C++ database.

The domain model dependent software layer implements the basic object manipulation functionality and enforces domain model specific integrity constraints. This software layer can be generated automatically by source code generators.

The highest abstraction software layer in the PMDB is implemented manually by the product-developing company. This may include common routines used by many client applications or routines for different types of expensive or critical data processing, for instance a deep copy function or certification checking and design-state-changing functions.

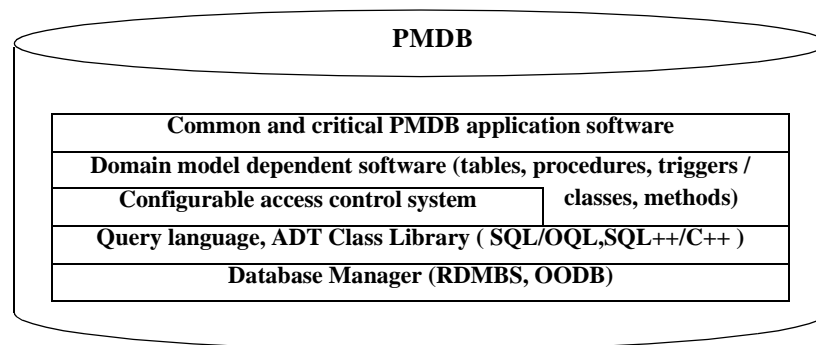


FIGURE 15. Software layers in the product model database.

9.3.2 Software layers in CAD client applications

Figure 16 shows the structure of software layers in a CAD-client. The border-surrounded software layers are implemented in the CAD-system. They provide command-, and application programming language access to the geometry database of the drawings and a user interface library. The UI-library usually supports programming and configurations of various types of screen-based menus, digitizer tablets and forms for entering data, to mention a few examples.

The database interface (DBI) layer is purchased from the DBMS vendor of the PMDB. It provides a library of DBMS functions that can be called from a 3GL programming language (C, C++, Fortran).

The CAD-DBI interface implementation depends on both which CAD-system is used and the DBI of the selected DBMS. Its function is to convert product model data received from the DBI into data structures that can be manipulated from the application programming language of the CAD-system. Similarly data created in the CAD-system must be converted to formats that can be transmitted back to the PMDB.

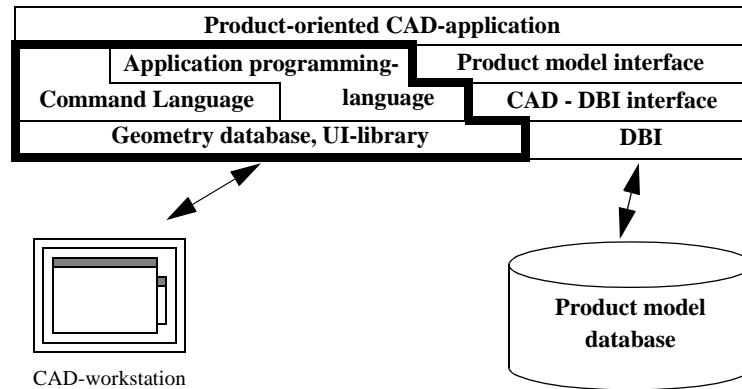


FIGURE 16. Software layers in the CAD-client.

The product model interface layer provides a domain model-specific library of functions or methods for transparent manipulation of objects and structures in the PMDB. It may also implement a library of functions for form based editing of loaded product model data using the user interface functionality in the UI-library.

The purpose of the layer is to give a domain model-specific naming abstraction above the product model data, which makes manually written CAD-applications more compact and easy to maintain.

The software in the product-oriented CAD-application layer is company or application domain specific and implements the drawing functionality for the particular application, including libraries of symbols and/or mechanical components.

PMDB-related modules in this layer can, for instance, check the CAD-drawing for consistency with the product model data. Other functions may cache geometric design information taken from the CAD-drawings into objects in the product model database. From there it can be used by external CAE-applications or by programs that are executed directly in the PMDB server.

Various processing results that are stored in the PMDB can be read back into the CAD-system and visualised directly on the geometries, using symbolic markers, texts or geometrical objects.

9.3.3 Software layers in browser applications

The layering structure of a browser application is similar to the one for a CAD-application. It has a DBI layer and a GUI⁴⁰-DBI interface. The graphical user interface library layer provides the primitives from which

40. Graphical User Interface.

high-level user interface components such as windows, object editors, attribute editors, and various relationship editors can be built (see chapter 18).

The layer providing product model-specific UI-components contains default layout and configuration declarations for the user interface objects that will represent the information for different types of objects in the product model. The declarations include datatypes, widget coordinates, and small action procedures that are executed when the user presses buttons, double clicks items in listboxes and so on.

The product model application layer must implement login facilities and user interface access to some “root”-object from which browsing can start.

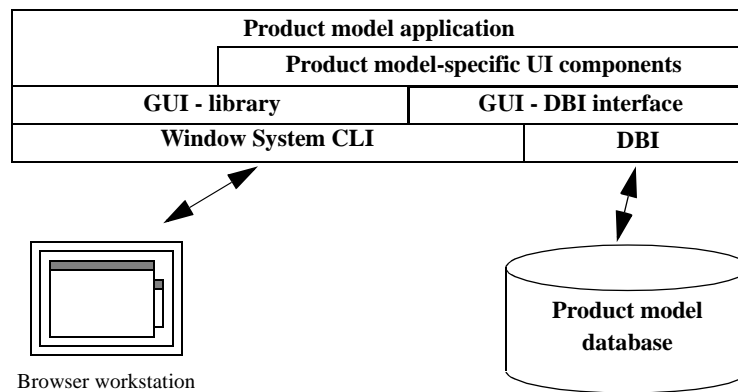


FIGURE 17. Software layers in a browser application.

If the requirements are not a single application program provided as one executable file, browser applications can, of course, also be implemented using 4GL tools or internet browsers where the product model-specific UI-components are, for instance, generated HTML-scripts.

9.4 Architecture of a PMS development system

Figure 18 on page 70 depicts the architecture of the PMS development system we have used for developing ProCAD. It can be divided into three parts. In the upper part there is an OOCASE tool for interactive graphical design of object-oriented domain models. In the centre we have the meta-database which is implemented on a relational database. The meta-database can be read by source code generators written as SQL-scripts, which generate the layers of the database- and application source code that are specified by the domain model. The bottom part is the generated PMS, which can be recognized from Figure 14 on page 65.

The OOCase-program we use has its roots in an experimental system for testing the validity of a novel user interface software architecture,

developed as a thesis project [Johansson 91]. A special feature of OOCASE is that it allows several smaller object model diagrams to depict related subsets of a larger object-oriented domain model. In this way, a system designer can concentrate on a smaller diagram which only shows the subset of classes, attributes and relationships that are of interest for a certain aspect of the domain model. There is less need for the traditional “spaghetti” diagrams which “take up a wall” to get a grasp of the whole model. Printed out, our PMS domain model consists of about twenty A4-sized object model diagrams, defining different functional parts of the power plant product model.

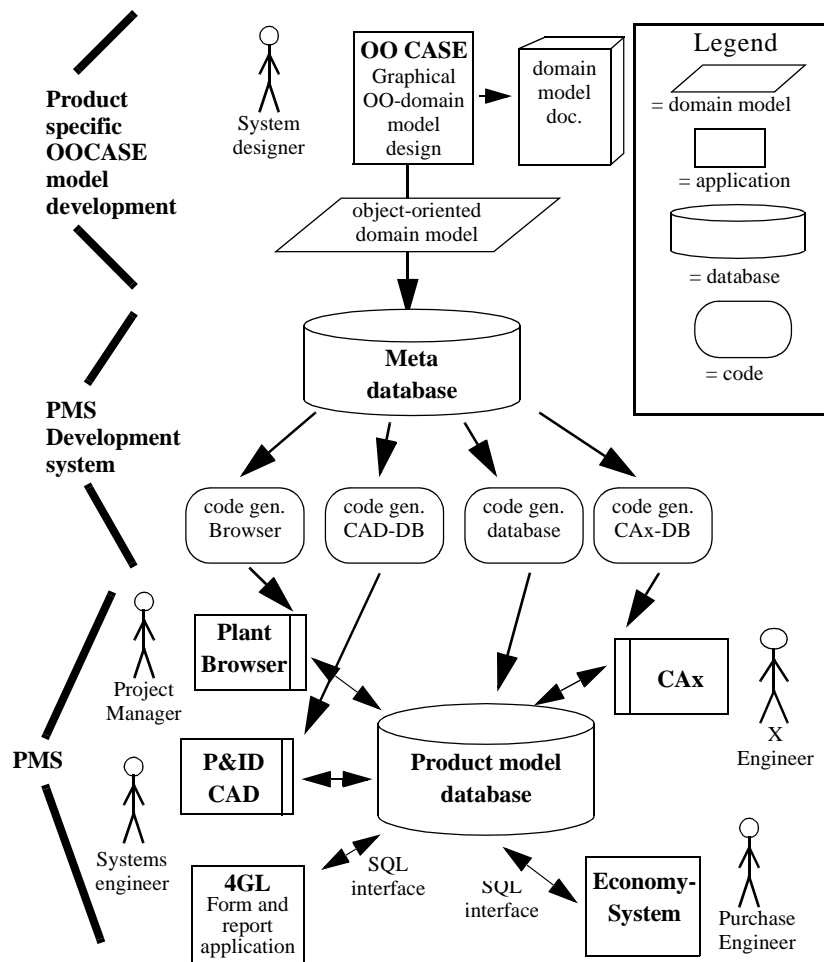


FIGURE 18. Architecture of the PMS development system.
The CAx and Economy system are probable extensions.

Domain models can be exported from OOCASE in a record format or as a batch file of SQL-insert statements. We use the relational database Sybase

for implementing the meta-database. Classes, relationships, attributes and other OO-domain modeling object types have their corresponding relational tables. The SQL-batch inserts tuples representing domain modeling objects into these tables. A system designer can interact with the meta-database directly using SQL, or indirectly via a 4GL-tool such as Microsoft Access.

Before code generation can be applied, the domain model needs to be supplemented with implementation data that is dependent on the programming language of the target tools. Standard data types, for instance, have different keywords in different target languages. The keywords are stored in the meta-database, and have to be associated with attributes in the domain model. We have a set of stored procedures that add this extra implementation information.

The code generators are SQL-scripts written in Sybase Transact-SQL [Sybase 94][Sybase 94a]. They consist of select statements that combine literal source code text strings with data from the meta-database and produce a set of string tuples that make up the generated lines of source code.

A code generation script can be seen as a source code template, which is instantiated with data from the meta-database. When generating source code for a relational PMS database implementation the templates typically define different standard types of triggers and stored procedures.

For each class in the domain model P stored procedures are generated. These are typically procedures that create, update, or delete an instance of a class in the PMS database. See Figure 20 on page 75 for an example. Stored procedures are pre-compiled within the DBMS and can be called from a CAD-system by a remote procedure call. Such communications give fast response times.

For a complex domain model with C classes, the number of generated procedures becomes $P \cdot C$. Data which is stored in one place in the meta-database is duplicated to a large number of places in the generated source code. Duplication factors of 100 are not unusual.

During the development of the domain model, an attribute of a class which is heavily inherited may be changed. If the domain model-dependent parts of the PMS implementation have been coded manually, such a minor change may need manual changes to several hundred lines of source code spread around many different source code files.

Tracking down the locations and make the changes may take a significant number of hours even for a programmer who is familiar with the code. Some places will probably be forgotten and bugs will appear later on.

Using source code generators, significant amounts of manual change and debugging work is saved. While still working with prototypes, a complete re-implementation of the domain model dependent parts of the system may

be done in a few hours including manual work.

For our power plant PMS more than 50 000 lines of source code are generated and it takes some time to recompile and reload it. For one version of the ProCAD database, it took about half an hour to generate the source code and execute the generated SQL-batches that create tables, stored procedures and triggers. Note, however, that very little effort have been spent on performance improvements of our meta-database implementation.

The main advantage of source code generation, which really speeds up the PMS development process, is that the software engineer who performs the re-implementations does not have to refresh his/her understanding of the vast amount of details in the source code.

9.5 An object-oriented domain model example

Object-oriented domain models serve as a medium for communication between product experts and software engineers. Figure 19 shows the object model diagram of a simple domain model, containing 4 classes, 3 relationships and 18 attributes. The example is used to introduce the UML class diagram notation. It also gives the reader an idea of what is stored in the meta-database (further described in Section 9.7 on page 76).

The domain model describes a fictitious PMS database for a company with several *Departments*. A *Department*-class has the two attributes *department_id* and *department_name*. Each department can own an interval between minimum zero and maximum infinity [0..*] products. Each product must be owned by exactly one [1..1] department. Thus the class *Department* has a one-to-many relationship *department_products* to the class *Product*. A black diamond on a relationship end means that objects instances at that end are composed of object instances at the other end. This is a part-or relationship, as opposed to an more light weight association.

A product is described by a hierarchical structure of articles which has its root in a main article. This is represented by giving *Product* an one-to-one relationship *product_mainArticle* to *Article*. *Article* has a one-to-many relationship *owner_owns* to itself, and thus an *Article* can be composed of a hierarchical structure of sub-articles according to this domain model.

For the management of a PMS database we need some common database attributes on each product model object. These are gathered in the class *DatabaseObject*, which is inherited by all other classes in the domain model. More about *DatabaseObject*'s attributes later.

Inheritance is represented by the symbol \triangleleft followed by the name of the inherited class. Such a superclass reference is easy to explain to non-software engineering people. Just say that $\triangleleft DatabaseObject$ represents a copy of all attributes in *DatabaseObject*. This notation makes the diagrams look clearer than drawing lines with an inheritance symbol which inexperienced people easily confuse with ordinary relationships.

Each object in the PMS-database needs a unique object identifier. Since we use a relational database for implementing PMS-databases, we have chosen 64-bit object identifiers divided into two 32-bit standard long integers in the key attributes *highId* and *lowId* of *DatabaseObject*.

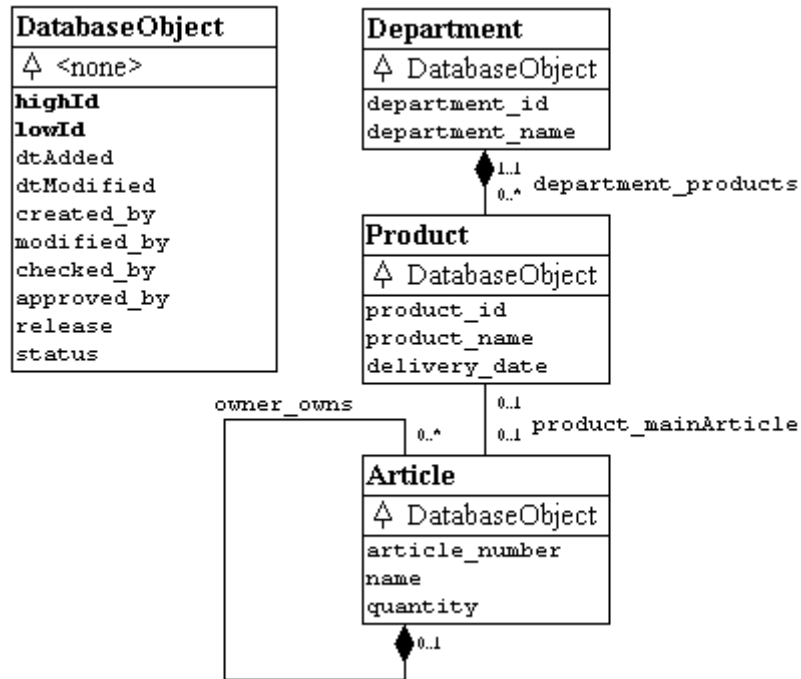


FIGURE 19. Simple PMS domain model example to describe the graphical syntax of our object model diagrams.

To ensure object identifier uniqueness amongst product models developed in parallel PMSs, each PMS-database has been given a unique *highId*. *lowIds* are generated from a counter that is ensured to always be larger than a certain value calculated from the current time. In practice this works as a reasonable insurance that the same object identifiers will not be generated again, even after an old database backup has been restored. If one object identifier is generated every second, a 32-bit counter will last for 68 years.

The attributes *dtAdded* and *dtModified* are time-stamp attributes which store the date and time when the object was first added to the database, and when it was last modified. These attributes are managed automatically by triggers in the generated PMS database implementation.

The attributes *approved_by*, *checked_by*, *created_by* and *modified_by* store a login-identifier for the user performing the corresponding task. Using triggers in a relational database, the modification of an object can be inhibited if it has already been checked or approved.

9.6 Source code generation example

Figure 20 on page 75 shows an example of an automatically generated stored procedure written in Sybase's Transact-SQL. The procedure updates a row in the table *de_department* where instances of the class *Department* are stored. "de" is a two character *prefix* which uniquely identifies the class *Department* within the domain model.

During code generation, the parameter names (@de_highid for example) and the field names of the table are fetched from the attribute objects stored in the meta-database. The same template type of update procedure can be generated for all entities in the meta-database.

If the domain model is changed, for instance an attribute is added to the class *DatabaseObject* which is inherited by *Department*, the procedure in Figure 20 has to be regenerated. Examples of other automatically generated procedures for the *Department* class are given in Table 1.

Table 1: Generated stored procedures for object handling.

| Generated procedure | Description |
|---------------------|---|
| de_insert | Creates a new Department instance. Return its new unique object identifier in the l-value parameters. |
| de_update | Updates the attribute values of the Department instance. See the example in Figure 20. |
| de_delete | Delete a Department instance whose object identifier is supplied as parameter. |
| de_select | Select all attribute values for a department object with the supplied object identifier. |

Procedures that manipulate relationships are bound to a particular class. Examples of procedures that are generated automatically for the 1-N relationship *department_products* are given in Table 1.

Table 2: Generated stored procedures for relationship handling.

| Generated procedure | Description |
|----------------------|---|
| de_relAllProducts | Select all attribute values for Product objects that belong to the department whose object identifier is supplied as parameters to the procedure. |
| de_relCreateProducts | Create a new product object and connect it to the department object whose object identifier is supplied as parameters. |
| de_relDeleteProducts | Delete a product object which belongs to a particular department object. |

The procedures are loaded into the PMS database server, and can be called from a client application, for instance when a user selects a command from a menu in the user interface of a generated browser application.

```
create proc de_update
    @de_highid obid,
    @de_lowid obid,
    @de_dtadded datetime null,
    @de_dtmodified datetime null,
    @de_approved_by char(5) null,
    @de_checked_by char(5) null,
    @de_created_by char(5) null,
    @de_modified_by char(5) null,
    @de_department_id char(8),
    @de_department_name char(20) null
as
    declare @result int
    begin
        transaction de_update
        update de_department
        set /* key de_highid is not updated */
            /* key de_lowid is not updated */
            de_dtadded = @de_dtadded,
            de_dtmodified = @de_dtmodified,
            de_approved_by = @de_approved_by,
            de_checked_by = @de_checked_by,
            de_created_by = @de_created_by,
            de_modified_by = @de_modified_by,
            de_department_id = @de_department_id,
            de_department_name=@de_department_name
        where de_highid = @de_highid
            and de_lowid = @de_lowid
        execute @result=check_error_and_rowcount1
            @@error, @@rowcount, "de_update"
        if @result=0 commit transaction de_update
        else rollback transaction de_update
        return @result
    go
```

FIGURE 20. Example of an automatically generated stored procedure⁴¹.

The technique of concatenating literal strings with meta-database data in declarative SQL-scripts seems to be generally applicable for source code generation of all types of textual programming languages. We have used it

41. In Sybase Transact-SQL, all parameters and local variables are preceded by '@'. 'obid' is a user defined type equal to a 32-bit integer. @@error and @@rowcount are Sybase system variables holding a possible error number and the number of rows that were affected in the last operation.

for generating code in SQL, C++⁴², LISP⁴³ and textual user interface definition languages.

9.7 The meta-database domain model

In the same way as a PMS-database is an engineering database described by a *PMS domain model*, the meta-database is a software engineering database, described by a meta-database domain model. As a matter of fact, the domain model for the meta-database is actually stored in the meta-database itself, and has been used for generating stored procedures and meta-database browsers.

Figure 21 on page 82 shows an object model diagram for the meta-database. It is helpful to keep a bookmark at this page for quick reference while we explain the some of the important characteristics of the different meta-database objects.

9.7.1 DBObject

All meta-database objects inherit from DBObject. It manages unique key attributes for object identifiers (*highId*, *lowId*) and time stamping attributes (*dtAdded*, *dtModified*). After the creation of a DBObject instance, the current user's login-identifier is recorded in the *createdBy* attribute. After each modification to any attribute value within a DBObject instance, the current user's login-identifier is recorded in the *modifiedBy* attribute. In a relational database implementation of the meta-database, this kind of functionality can easily be implemented with triggers.

9.7.2 Object

Object inherits DBObject. Meta-database objects which are created by the user inherit from Object. Most meta-database objects, such as Class, or Attribute need a *name* and a short descriptive textual definition. When working at an international company such as ABB, the PMS domain models have to be coordinated between different sites in different countries. Therefore it is useful to have an alternative name and an auxiliary name that can document names in other languages. In our case, we have used *name* for English names, *altName* for the Swedish names, and *auxName* for German names.

In our implementation, the English names can have about 30 characters and are used for source code generation. Sometimes, however, older target languages do not allow such long identifier names and then a *shortName* can be used instead.

42. [Lippman 92] [Stroustrup 93]

43. [Steele 84].

9.7.3 DataDictionary

There is only one instance of *DataDictionary* in the meta-database. This instance is used for holding “global” meta-database variables, and represents a “root” object in a part-of structure, from which the user can navigate down to other domain model objects in an automatically generated browser.

9.7.4 DomainModel

For each PMS-implementation, there is a corresponding domain model. Instances of *DomainModel* represent the “root”-object of such a domain model.

9.7.5 Class

Instances of *Class* hold class-related information. When generating source code for a relational DBMS, the inheritance hierarchy is flattened, so that leaf-classes have all their attributes stored in the same table. Only leaf classes are relevant for generation of tables. To distinguish them from inherited ones such as *DatabaseObject* in Figure 19, leaf classes have their *genSqlFlag* set to TRUE. The *prefix* holds a two-to-four character prefix that uniquely identifies the class within a domain model. This is useful since many named code objects in generated code are related to a specific class. Examples of named code objects are table related stored procedures, triggers, or user interface forms for editing instances of a particular class. As presented in Figure 19 on page 73, stored procedure names such as *de_insert*, *de_update* and *de_delete*, are generated for the class *Department*. If the name of a generated table for some reason cannot be the same as the English *name* for the class, the system designer can override it by entering something in *tableName*.

9.7.6 Relationship

Relationships in our domain models are binary. The connected two classes are called *class1* and *class2*. For a one-to-many relationship, *class1* is on the one-side, and *class2* on the many-side. The relationship type, e.g. 1-1, 1-N or M-N, is stored in *type*.

A relationship can be viewed from the perspective of *class1* or *class2*. The relationship is referred to from *class1* by the name specified in *name1to2*. *name2to1* is used for referencing the relationships from *class2*.

A relationship also has implementation-descriptive attributes, such as *cascadeDelete2*. This flag tell the code generators that if an instance on the *class1*-side of the relationship is deleted, then all related instances on the *class2*-side should be deleted.

9.7.7 Attribute

A class owns a set of attributes via the relationship *class_attributes*. In the meta-database it is possible to specify a *defaultValue* for an attribute and define if the value is *mandatory* i.e. NULL-values are not allowed. The *keyNumber* specifies the position of a key attribute within a composite key. Non-key attributes have the *keyNumber* set to zero.

The *type* of an attribute can hold an optional data type definition. If an attribute is connected to a Property, it will receive its data type declaration via the path `Attribute.property->Property.domain->Domain.typeDef->TypeDef.declarations(language)->Declaration.declaration`, where *language* is a parameter that selects a Declaration for the programming language.

9.7.8 AttributeGroup

When classes inherit a large number of attributes from a deep subclass hierarchy, semantically related attributes for the same class get scattered if they are ordered by *inheritance level* and *name* on forms and listings. The meta-database object AttributeGroup enables a grouping according to semantic relatedness. Each attribute group is assigned a *groupId* consisting of a four character code. When selecting data from all inherited attributes for a particular class from the meta-database using an SQL-query, the results can be ordered by *groupId* and *name*. This is very useful for providing review listings when product experts are asked to provide comments on information content for various classes in the OODM.

9.7.9 Property

While developing several larger domain models within the same product domain, one discovers that there are attributes which appear over and over again. In the domain of turbine design, the attribute *article_number* is a good example of this. In such cases, it is useful to store a common definition for that standard attribute as a Property in the meta-database. This allows several attributes to share the definition of, for instance, data type or default value. If a company implements several PMS databases from the same meta-database, they can easily combine data from the different databases, by joining over standard attributes.

When several domain models have been developed, a property library emerges. While developing new domain models, company standard attribute names and definitions can be selected from the property library and thus enforce reuse of definitions. This gives an opportunity to gain control over diverging semantics for company information resources.

9.7.10 TypeDef

All variable and attribute implementations are based on some data type. A PMS may be implemented using several different programming languages which need to share data and thus type definitions. A *TypeDef* stores a standard type definition and has a set of *Declarations* for various programming languages. A signed 32-bit integer is declared as a long in C and C++, while the name of the same data type is int in Sybase' SQL. When generating code, especially for interface libraries between CAD-systems and databases, this kind of data-type mapping information is necessary.

In our meta-database, we have a set of standard atomic data types defined on the basis of the ones available in OMG's Interface Definition Language (IDL) which is specified in [CORBA91].

9.7.11 Domain

A *Domain* defines a value domain. An example of a *Domain* is *week_number*. A *week_number* can have the *TypeDef* "Integer", and a range of values between 1 and 53, i.e. *minValue* = 1 and *maxValue* = 53. A PMS domain model may have classes for project planning. An Article class may have a DesignActivity class that contains the attributes *start_week* and an *end_week*. There may also be a ManufacturingActivity class for the article's manufacturing process, which has the same week attributes.

Now it is possible to define the two properties *start_week* and *end_week* which both belong to the *Domain* *week_number*. Through the relationship *property_attributes* the attributes of the DesignActivity and Manufacturing Activity are connected to their corresponding properties *start_week* and *end_week*.

Later during the domain model development, one may discover that the planned design period for certain large scale articles may last for more than a year. Hence the domain *week_number* also must include the number of the Year. All affected attributes in the meta-database can be traced through the relationships *domain_properties* and *property_attributes*.

By changing the definitions in the domain, the change can be automatically propagated through the properties to attributes in all domain models. Using the code generators, the PMSs can be re-implemented with the new data type.

This section gave a short description of some of the classes and relationships in the domain model of our meta-database. Many of the classes which are not described here are used for storing abstract intermediate implementation models which aid the code generation. Others are used for modeling user roles and descriptions of their tasks. These are presented in chapter 15, "An Information-oriented Task Description Language".

9.8 Experience from the ProCAD development

ProCAD is being developed iteratively using prototypes. Engineering experts in power plant design participate in the continuous development of the PMS domain model. When a new version of the model is confirmed, a prototype is implemented which takes somewhere between a day and a couple of weeks, depending on how much code that has to be written manually.

Plant designers and other users work with the new prototype, get new ideas and give their suggestions on what should be included in the next prototype version. The suggestions are compiled, remitted and ordered according to priority to become the input specification for the next version of the PMS domain model.

In one version of the turbine power plant PMS, the object-oriented domain model contained about 70 classes, 40 relationships and 380 attributes. When the model was printed on paper in the form of object model diagrams and reports, it produced about 170 A4 pages. The generated Sybase implementation of the PMS database built 50 tables which contained about 1800 fields.

The CAD-application for drawing process and instrumentation diagrams is based on a commercial product and has been extended with special drawing functionality by an AutoCAD application development company. This is a recommended way of saving researchers from spending most of their time doing painful, but necessary quality- and usability- improving implementation work. Without a commercial status of the PMS, it is hard to maintain the interest from busy product engineers.

A few of our leading product engineers develop their own report- and search form applications in Microsoft Access with some support from the computer department of ABB STAL.

The software developed at Linköping University consists of more than 60 000 lines of code, of which more than 60% is generated automatically from the meta-database. At ABB the system runs on a mixture of platforms, including IBM RS 6000, Sun Sparc, Silicon Graphics and PCs.

9.9 Conclusions

There is a large potential benefit in using object-oriented product models for engineering of high-tech mechanical artifacts. There are, however, three major obstacles that prevent development of successful product modeling systems (PMS).

- I) The design of the product models quickly becomes very complex.
- II) The implementation of a PMS requires extensive software engineering skills, and
- III) Computer technology is under continuous development and a PMS implementation will become obsolete within a few years.

Our solution to overcome these problems is to use object-oriented CASE models which enable a clean separation of the domains of product specific engineering knowledge from software engineering implementation knowledge. The models are stored in a meta-database from which source code implementations can be generated automatically.

Following this approach means that:

- I) Product-specific engineering knowledge is documented as an object-oriented CASE-model which is stored in a meta-database according to standard format. Object-oriented CASE-tools facilitate the development and maintenance of large object-oriented domain models for PMSs.
- II) Different software vendors can package their software engineering knowledge for a particular software platform into source code generators which transform standard format object-oriented CASE-models into working implementations on their specific software platforms. Typical targets for such implementations are databases, user interfaces, and interface libraries between CAD-systems and databases.
- III) New generations of computer technology can be put into production when source code generators for the new target technology have been developed. In the future we see SQL3-databases, a wide range of parametric CAD-software packages, and 3D-user interfaces where object-oriented product models can be developed in virtual 3D worlds.

The approach is being successfully used for the development of a power plant PMS. We expect this system to become easy to port to new generations of computer technology. These are exemplified by [FahlRischSköld 93].

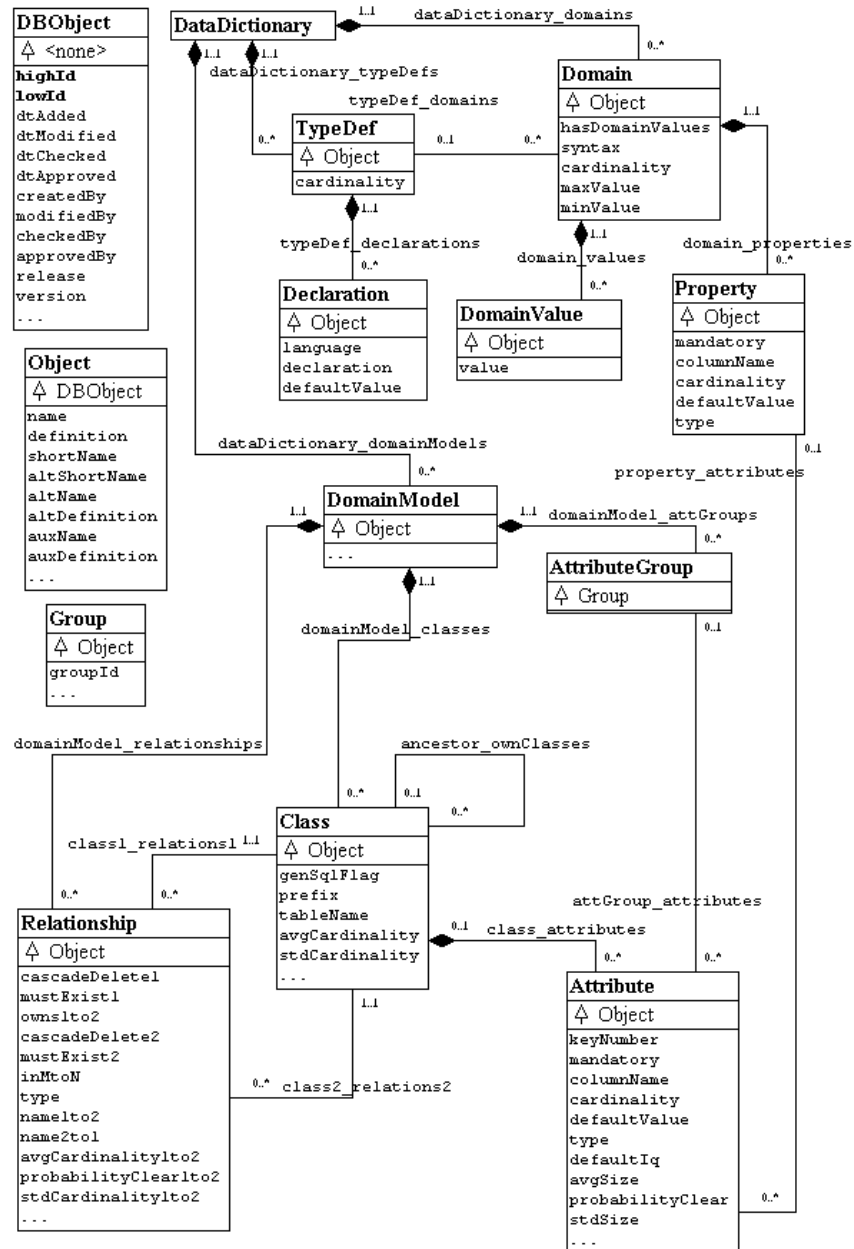


FIGURE 21. An object model diagram for the meta-database. See section 9.7 on page 76 for a text description of the different domain model primitives. Section 9.5 on page 72 explains the graphical syntax.

10 Source Code Generation from Domain Models

This chapter describes the principles behind declarative source code generation from domain models that are stored in a meta-database.

The prerequisite for using this technique is that the meta-database has a declarative query language with the expressive power of SQL. It must also provide traditional programming language constructs such as loops and if-then-else constructs.

Section 10.1 describes a small version of a meta-database, containing a minimum set of information needed in a domain model for the generation of basic object management functionality in a product model database.

Section 10.2-3 describe declarative 1- and 2-step source code generation.

Section 10.4 describes measurements on the size of the domain models, the source code generators, and the generated source code.

Finally the conclusions show that source code generation from domain models enables a significant increase in productivity for software implementation and maintenance.

10.1 Domain model for a mini meta-database

Figure 22 shows a mini version of the domain model for the meta-database described in section 9.7 "The meta-database domain model" on page 76. This subset is needed for implementing the source code generators that generate the core object management functionality of a product model database. The class *Object* is the root class. The leaf classes in the inheritance hierarchy have tables in the meta-database. These are *DomainModel*, *Class*, *Relationship* and *Attribute*. Their common attributes (highId, lowId, definition, and name) are inherited from the class *Object*.

Classes in a stored domain model have a two character unique prefix, which is used to prefix generated tables, stored procedures and triggers that are related to the table. The naming convention we have chosen for generated tables is the prefix plus an underscore followed by the name of the class in lower case. i.e. the name of the table for company in Figure 26 on page 86 is **co_company**.

We use 64-bit object identifiers as keys, both for the meta-database and the generated product model database. The identifier is divided into a 32 bit

highId which is assigned uniquely for each database, and a 32 bit lowId that is generated from a counter within the database.

A table like the one in Figure 26 on page 86 is generated for all leaf classes⁴⁴ in the domain model inheritance hierarchy.

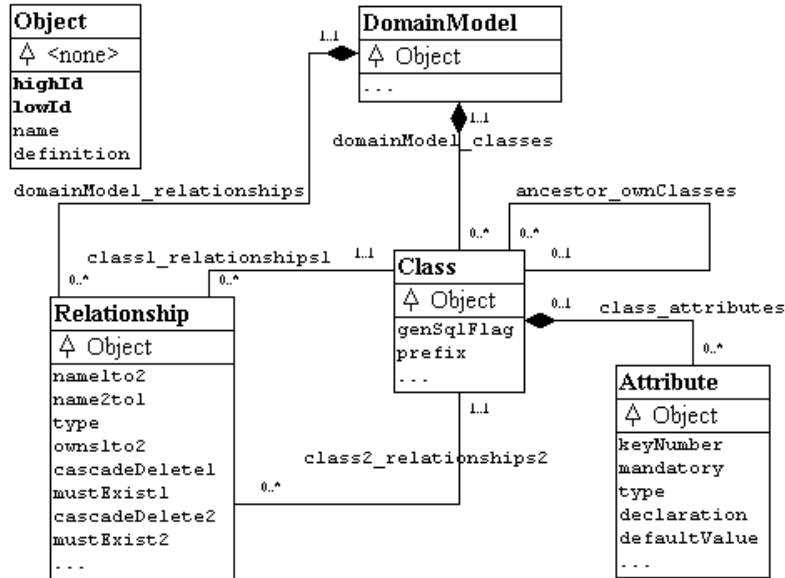


FIGURE 22. A subset of the domain model for the meta-database

The source code generator is an SQL-select statement where values in the prefix and name columns of the *Class* table in the meta-database are selected and concatenated with literal strings that contain the template source code.

10.2 Declarative 1-step source code generation

Source code generation is performed in a data-driven template-oriented fashion. Figure 23 on page 85 shows a portion of a simple source code file generated from a domain model. The example grants select permissions to all tables to the user with the authorization ID chief_designer. The ‘go’ command sends the preceding SQL statements in the input file to the server and waits until the batch is executed and the results returned, if any.

The source code in Figure 23 is generated from the source code generator in Figure 24. The source code is generated as shown in Figure 25 from the benchmark domain model (Appendix A page 182). Input to the meta-

44. Leaf classes have their genSqlFlag set to true to provide a simple selection criteria for queries (see Figure 24)

database query program (ddq for data dictionary query) is a sequence of SQL-batches like the one shown in Figure 24. Its SQL-statements select the appropriate data from the meta-database and combines them with literal source code strings. The database server compiles and executes the batch and sends the resulting columns back to the ddq-program, which writes each column on a separate line in a generated source code file.

```
grant select on ar_article to chief_designer
go
...
grant select on co_company to chief_designer
go
grant select on da_designactivity to chief_designer
go
grant select on de_department to chief_designer
go
grant select on dl_deliverable to chief_designer
go
grant select on dr_drawing to chief_designer
go
grant select on ia_includedarticle to chief_designer
go
grant select on pd_productdata to chief_designer
go
grant select on pr_product to chief_designer
go
```

FIGURE 23. Example of simple generated source code from the benchmark domain model presented in chapter 13.

```
select "grant select on "+
      c.prefix+"_"+lower(c.name)+
      " to chief_designer"
      ,"go"
from class c
where c.gensqlflag = 'T'
order by c.prefix
go
```

FIGURE 24. Source code generator for the code in Figure 23.

In the simplest case, the ddq-program takes an SQL-batch as standard input, and prints the generated source code to standard output:

```
ddq <generate_grantselect.sql >grantselect.sql
```

The simple source code generator in Figure 24 consists of 8 lines and about 150 characters. The generated source code, which is only partly presented in Figure 23 consists of 36 lines and about 850 characters, a generation factor (generated source code size / source code generator size) of 4,5 for the lines and 5,5 for the number of characters for the benchmark domain model. This generation factor is linearly proportional to the number of

classes in the domain model. Thus even if the source code will only be written once, it can be beneficial to write a source code generator if the number of classes are many.

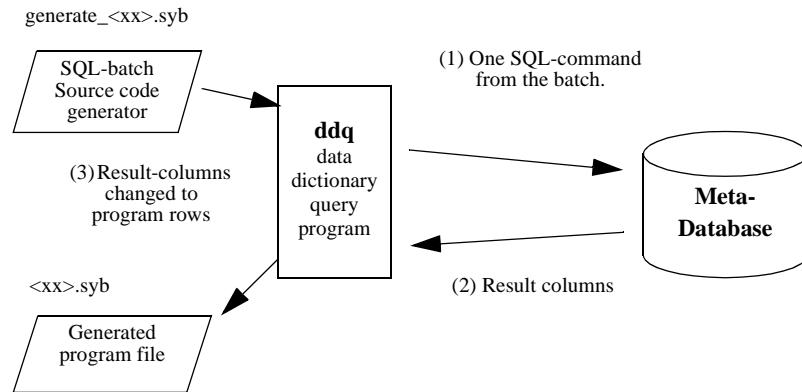


FIGURE 25. Principle of declarative 1-step source code generation.

10.3 Declarative 2-step source code generation

Figure 26 shows the source code for one of 18 generated table declarations from the benchmark domain model. The plain text is the literal source text in the source code generator, and the bold faced text is data supplied from the meta-database.

```
/* Table declaration for class Company */
create table co_company (
  created_by char(8),
  dtAdded datetime,
  dtModified datetime,
  highId int,
  lowId int,
  modified_by char(8),
  name varchar(40),
  ref_id char(8))
```

FIGURE 26. Source code generated by a 2-step source code generator.

This source code is generated in two steps.

```
ddq <generate_build_createtbl.syb >build_createtbl.syb
ddq <build_createtbl.syb
```

The first step generates an SQL-code generator which in turn generates the final source code files. Figure 27 shows some more details of the 2-step source code generation procedure.

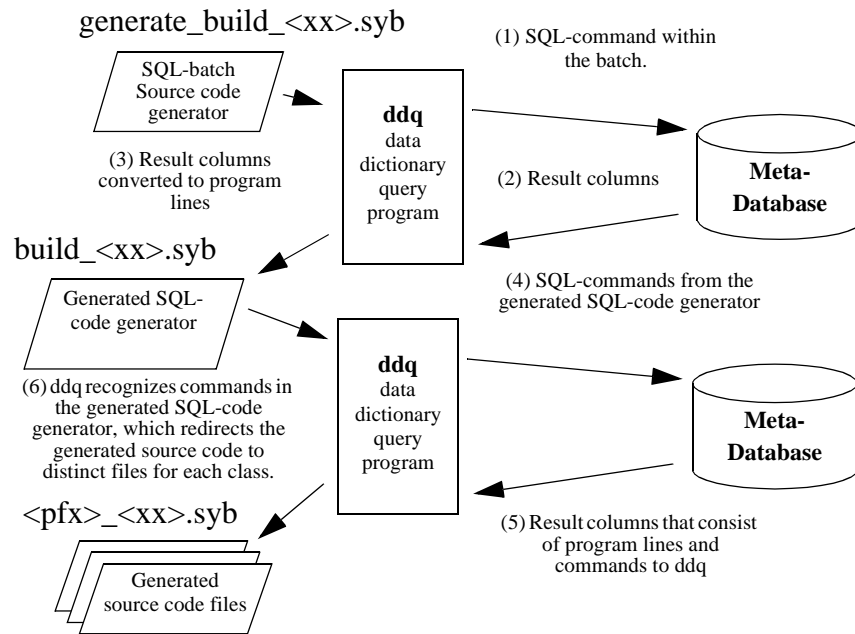


FIGURE 27. Principle of declarative 2-step source code generation.

Figure 28 shows the fraction of the generated source code generator from the first step which in turn generates the code in Figure 26. The first step instantiates data from the classes in the domain model, and the second step instantiates data from the attributes for each class. `allattribute` is a view on a table that caches all attributes for all classes in the inheritance hierarchy.

```
select '/* Table declaration for class Company */,'
go
select 'create table co_company ('
go
select '  ' + a.name + ' ' + a.declaration + ','
from allattribute a
where a.class_highid = 1101
and a.class_lowid = 112011375
order by name
select char(8) + char(8) + ')'
go
```

FIGURE 28. Small portion of the generated SQL-code generator.

The select of `char(8)+char(8)+')'` backspaces two characters and overwrites the comma behind the last attribute declaration with a `)`.

Figure 29 shows a simple version of the 2-step source code generator for the table declaration in Figure 26.

```
select "select '/* Table declaration for class "+c.name+" */,'"
"go"
,"select 'create table "+
      c.prefix+"_"+lower(c.name)+" ("
"go"
,"select ' '+a.name+' '+a.declaration+''"
,"from allattribute a"
,"where a.class_highid = "+convert(varchar,c.highid)
," and a.class_lowid = "+convert(varchar,c.lowid)
,"order by name"
,"select char(8)+char(8)+''"
"go"
from class c
where c.gensqlflag = 'T'
order by c.prefix
go
```

FIGURE 29. Source code generator for the code in Figure 26.

The source code generator shown in Figure 29 consists of 17 lines and about 440 characters. The generated source code consists of 312 lines and 6350 characters for the benchmark domain model, which means a generation factor of 18 for the number of lines and 14 for the number of characters.

10.4 Measurements from the development environment

The source code generation techniques described are an efficient way to reuse software implementation knowledge. The larger the domain models are, the larger the generation factors become. This has been verified by measurements. Table 3 shows domain model measurements from the benchmark application and one version of the ProCAD application developed in cooperation with ABB STAL.

Table 3: Domain model measurements.

| Domain model | classes | leaf classes | relationships | attributes | database columns |
|--------------|---------|-----------------|---------------|------------|---------------------|
| benchmark | 27 | 18 | 16 | 45 | 292 |
| ProCAD | 52 | 40 | 46 | 429 | 1582 |

Five types of implementation functionality were generated using different sets of source code generators. They are listed in the left column of Table 4. The product model database (PMDB) stored procedures are used for

accessing and updating the object structures in the database from the CAD-system. The PMDB triggers are used for maintaining referential integrity in the database.

The CAD/DB interface procedures belong to the product model interface layer in the CAD-system (Figure 16 on page 68). They are implemented in AutoLISP which is an integrated application programming language within AutoCAD. They provide the domain model-specific library of LISP-functions for manipulating objects in the product model database.

The CAD/UI interface belongs to the same layer, and implements a form-based browser interface for objects in the PMDB, using AutoLISP and the built in form layout declaration language of AutoCAD.

The bottom two rows show measurements from test implementations done in Smalltalk.

Table 4: Source code generator measurements.

| | Number of source code generator scripts | Number of procedure generators | Number of lines | Number of characters |
|---------------------------|--|--------------------------------------|--------------------|-------------------------|
| PMDB stored procedures | 27 | 43 | 1563 | 57492 |
| PMDB triggers | 4 | 4 | 201 | 6629 |
| CAD/DB interface | 19 | 22 | 582 | 17012 |
| CAD/UI interface | 20 | 21 | 691 | 21933 |
| Smalltalk PMDB | 39 | | 3089 | 91969 |
| Smalltalk UI | 7 | | 420 | 13900 |

Table 5 shows measurements on the generated source code. The figures in parentheses are from ProCAD. Observe that these generated implementations only provide the basic object management functionality. Application-specific functionality is written manually using calls to these generated high-level domain specific software layers, and requires skills in both the domain and the CAD-system.

Table 6 relates measurements on the generated source code to the size of the source code generators. The PMDB implementation generated for the benchmark domain model consists of about 2-3 times as much source code as the source code generators themselves. For a domain model of the size of ProCAD, this factor is more than 10.

Table 5: Generated source code measurements. Benchmark (ProCAD).

| | Number of generated source code files | Number of generated procedures | Number of lines of code | Number of characters of code |
|---------------------------|--|--------------------------------------|-------------------------------|------------------------------------|
| PMDB stored procedures | 41 (85) | 180 (560) | 5538 (26126) | 132428 (687511) |
| PMDB triggers | 18 (40) | 54 (120) | 1052 (2612) | 29587 (78692) |
| CAD/DB interface | 38 (82) | 240 (738) | 4452 (11153) | 44342 497588 |
| CAD/UI interface | 37 (84) | 158 (406) | 2537 (15598) | 79218 625816 |
| Smalltalk PMDB | 90 (104) | | 19170 (53882) | 561523 (1941603) |
| Smalltalk UI | 18 (40) | | 5353 (23144) | 150588 (757606) |

Table 6: Generated source code size / source code generator size.

| | Factor of generated source code files | Factor of generated procedures | Factor of lines of code | Factor of characters of code |
|---------------------------|--|--------------------------------------|-------------------------------|------------------------------------|
| PMDB stored procedures | 1,5 (3,1) | 4 (12) | 3,5 (16) | 2,3 (12) |
| PMDB triggers | 4,5 (10) | 13,5 (30) | 5,2 (13) | 4,5 (12) |
| CAD/DB interface | 2,0 (4,3) | 11 (34) | 7,6 (19) | 2,6 (30) |
| CAD/UI interface | 1,8 (4,2) | 7,9 (19) | 3,7 (23) | 3,6 (29) |
| Smalltalk PMDB | 2,3 (2,7) | | 6,2 (17) | 6,1 (30) |
| Smalltalk UI | 2,6 (5,8) | | 14 (62) | 11 (54) |

The Smalltalk PMDB implementation was made to demonstrate that implementations for the domain models can be generated in a traditional object-oriented programming language. The generated Smalltalk User Interface is a straightforward browser application, with one form for each leaf class with attribute and relationship editors for each attribute and relationship that belongs to the class. Here, too, most of the functionality was factored out and placed in the GUI-DBI layer (Figure 17 on page 69). The generation factor for forms is proportional to $|C|*|A|$, where $|C|$ is the number of classes in the domain model, and $|A|$ the number of attributes. The functions in the CAD/DB interface does a simple forwarding of the calls to the stored procedures in the product model database. Other interface functionality was factored out into general procedures placed in the CAD-DBI software layer (Figure 16 on page 68).

10.5 Summary and conclusions

Source code generators that take data from one meta-database table can be written in one step. Source code generators that need to combine data from two meta-database tables, for instance the class and attribute tables, can be implemented in two steps.

In a two-step source code generation, the first step generates a source code generator with instantiated data from the first table. The second step instantiates data from the second table and produces the final source code.

The size of the generated source code divided by the size of the source code generators is called the generation factor. To implement the basic object management functionality of a product model database, a CAD/DB and CAD/UI interface, source code generators for about 90 types of procedures were written. They consisted of about 3000 lines of source code.

For the benchmark domain model with 27 classes, 16 relationships and 45 attributes, the generation factor is about 2-3. For a domain model with 52 classes, 46 relationships and 429 attributes the generation factor is larger than 10.

Thus using formal domain models, a meta-database, and the source code generation techniques described in this chapter can lead to increases in productivity of an order of a magnitude for the implementation phase.

Testing whether a source code generator produces correct code can be done by generating an implementation from one design space spanning benchmark domain model, and testing this implementation thoroughly.

Software maintenance is made significantly easier. If an error is detected or an improvement must be made in the generated source code, the code generator is rewritten and the old source code replaced by new, regenerated code. The generation factor for a domain model can be larger than 60 for certain types of source code.

If several applications use the same source code generator, the cost of the testing effort can be shared amongst these applications.

Part III Theoretical Framework for the Information System Platform

11 Introduction to Part III

Model based KBS for the design of complex artifacts need to be based on an information system that takes care of the basic information management issues of the models on which the KBS operates. The same aspects that apply to the development of complex information systems need to be addressed. These include proper design, implementation, test, validation and continuous maintenance by a staff that will change many times during the lifetime of the KBS.

Maintenance issues are more complex for KBS than for traditional information systems. Thus if the basic problems with information systems maintenance are not solved in a satisfactory manner, maintenance of a large scale KBS will soon become unmanageable.

One way to handle the maintenance problem is to restrict the utilization of the design space of the underlying information system platforms such that the object system management of a KBS implementation can be generated automatically from a domain model which is maintained in a meta-database.

As described in Part II, the underlying information system platforms will change as software industry provides new better solutions.

The KBS-user companies will need software industry to provide new source code generators for the new platforms in the same way as standardized programming languages need new machine code generators for new processors.

There must be a way to validate new source code generators such that the customer companies are guaranteed that their KBS will be correctly implemented on the new platform. The validation procedure must also provide measurable evidence that the KBS-application running at the user company will benefit in terms of performance from the shift to the new technology.

Making the source code generators pass the validation procedure is also a well defined goal for software engineering companies developing source code generators for new platforms.

The following chapters will introduce some fundamental concepts in information system design that constitute the core of a theory that enables a qualitative and quantitative validation procedure of source code generators for database and user interface application generators.

12 Concepts and Notation from Infological Theory

Infological theory has its roots in Börje Langefors' work [Langefors 66] [Langefors 93]. A major contribution to the understanding of the nature of information and data in the context of communication with humans is concisely expressed in his infological equation:

$$(EQ\ 1) \quad I = i(D, S, t)$$

where I is the information (or knowledge) produced from the data D and the pre-knowledge S of a person, by the interpretation process i during the time t . In the general case, S in the equation is the result of the total life experience of the individual.

This chapter presents a conceptual framework for the information system part of KBS and PMS applications.

The other well-known definition of information relates to the context of information transmission over a channel with a limited bandwidth. Section 12.6 provides definitions of the unit “bit” from information (transmission) theory to clarify the difference between the semantically oriented information unit “e-constellation”, used here.

12.1 A quantifiable infological framework

The infological frame-work presented briefly here was developed by Bo Sundgren [Sundgren 73]. His framework differs from other object system modelling languages in the respect that it allows explicit numerical quantification of the amount of information stored in a certain object system model. This is achieved by defining the term elementary constellation as the smallest possible entity of information. The term elementary message is used for denoting a message that can transfer one elementary constellation from a sender to a recipient. These features provide the necessary theoretical basis that enable numerical comparison of the information quantity in different object models measured in their number of e-constellations. It also enables comparison and evaluation of particular user interface designs⁴⁵, in terms of the hypothetical measures of

45. The user interface is intended for visualization and interaction with the contents of a particular knowledge- or data-base where the “schema” (or domain-model) of the knowledge- or data-base can be described in terms of an infological model.

accessibility and operability as defined in chapter 15, "An Information-oriented Task Description Language".

The central concepts in the infological framework are presented in Sundgren's thesis. The purpose of this chapter is to give a reader, unfamiliar with infological theory, an understanding of the important concepts.

12.2 Data and information

The terms data and information are central axioms for infological theory, and it is therefore in place to introduce them:

Definition : If a person intentionally arranges one piece of reality to represent another, we shall call the former arrangement data, and we shall say that the arranged piece of reality is a medium, which is used for storing the data.

The concept of information is more difficult to define in one single sentence. It is easier to describe some of its important properties. Information exists only in the mind of a human being as a part of that person's mental frame of reference. By a frame of reference, we mean the collection of concepts, definitions, laws of logic, empirical laws and perceived, deduced or deducible knowledge belonging to the mind of that reference person *P* at a particular time. A person's frame of reference will change continuously, depending on what new knowledge he/she acquires, and what is currently in focus in his/her conscious mind.

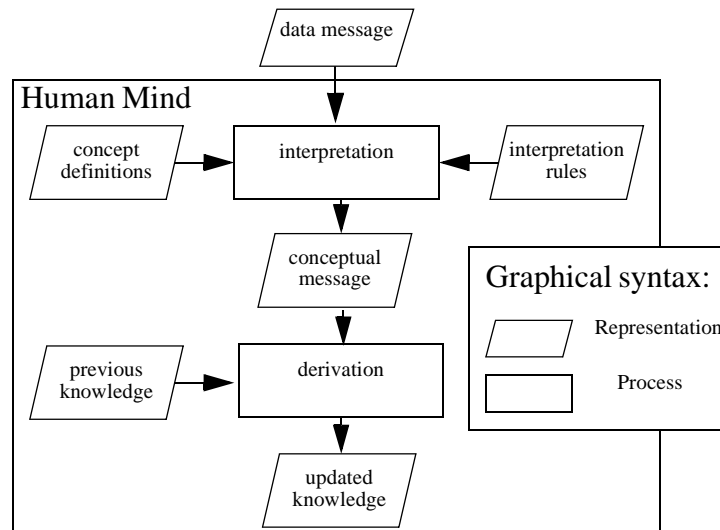


FIGURE 30. Transformation of data into information. From [Sundgren 73] page 24.

Figure 30 shows a model of processes and representations needed for transforming a data message into information that can be assimilated into the body of knowledge of a particular person *P*'s mind. The concept definitions are *P*'s understanding of the referents in a particular data message. The interpretation rules define how lexical and syntactical structures in the data message are mapped to the semantics. The result of the interpretation is a conceptual message that carries the semantics of the data message. If the conceptual message was not known before by *P*, it will be information, which may lead to an update of his/her former knowledge structures⁴⁶.

12.3 Infological model

A model is an abstraction, a simplifying representation, of some aspects of reality. It is created in order to facilitate analysis, planning, and decision-making within a particular domain. A model ought to be the result of a homomorphic mapping of one subset of reality to another. An infological model is a formal description of a certain target domain⁴⁷. It is defined with a set of generators. Each generator consists of a fundamental basis in terms of objects, properties, object relations etc. and a set of generation rules that together form a formal definition of the domain of a large closed world. The infological model is the result of analysis of the target domain. Figure 31 on page 100 shows the relations between a domain of reality, infological theory and a particular infological model in the process of developing a data-base (or KBS) application.

There are many similarities between formulating an infological model and knowledge acquisition (Figure 2 on page 31 and Figure 31 on page 100). Reality corresponds to the target domain for the particular application. The conceptual model was described in section 5.1.3 on page 33. The infological model is a formalized model of a conceptual model expressed using infological theory. A domain model or meta-model⁴⁸ is used for the same purpose as an infological model, but may be expressed in languages with a less well-founded theoretical basis, and may in some cases describe additional modelling aspects, for instance dynamic behavior. The database implementation model contains implementation-specific details and user interface specifications. This model is to represent the concepts available in the data definition language (DDL) of a target database or file system. In many cases the database implementation model is stored in the database itself.

46. Data messages are useful during human information processing even if they do not carry new information for a particular person *P*. The effect of receiving a data message will be to recall its contents to the conscious mind. In other terms it is a reactivation of the appropriate knowledge structures (chunks) previously stored in long term memory (LTM) in working memory (WM). (See Appendix E page 218 for further details.)

47. Infological theory was developed for facilitating the design and implementation of data-bases. A superset of traditional data analysis should to be made for the design of KBS.

48. Recall section 5.1.4 on page 33.

Once the infological model is specified, pieces of reality can be represented as data messages according to some chosen representation. These are transferred to the database during the implementation and operation of the database.

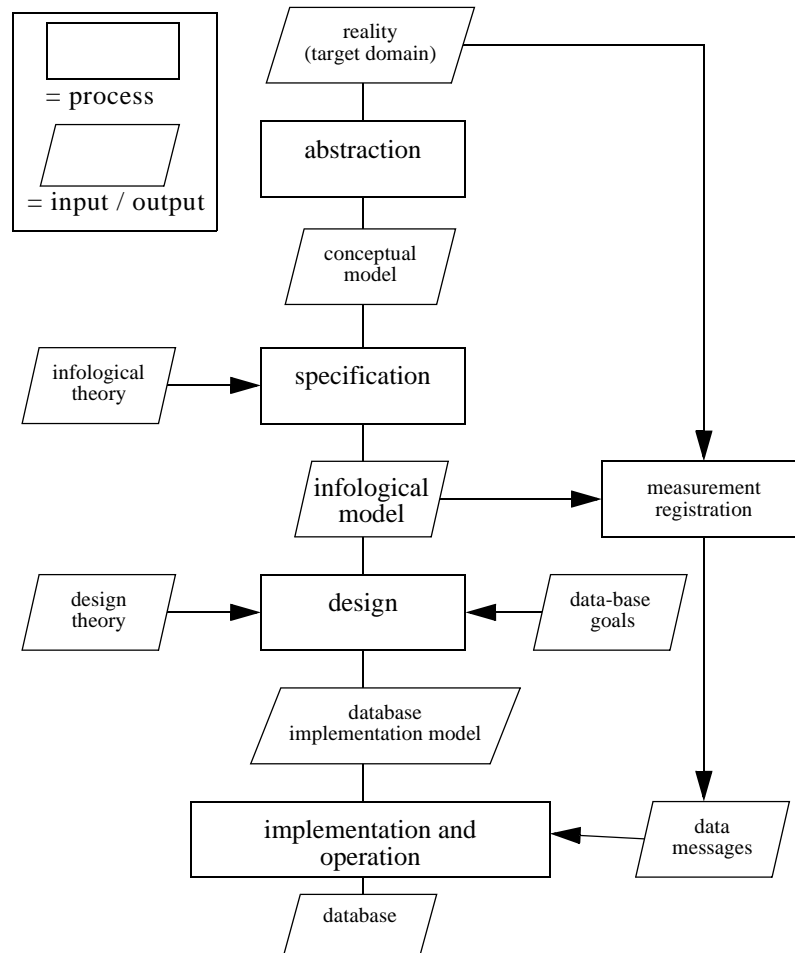


FIGURE 31. The homomorphic mapping of a slice of reality into a data-base model. From [Sundgren 73] p 32.

The rest of this chapter will focus on the infological frame-work for specifying an infological model.

12.4 Infological object system concepts

In infological theory, we try to model the real world without regard to database implementations. The reader is encouraged to forget about the common computer science concepts of object systems for a moment,

despite their obvious similarities with the terminology used here.

In the infological theory, there is a distinction between fundamental concepts, which are informally defined, and derived concepts, which are defined in terms of fundamental concepts. The theoretical frame-work can be divided into two areas, the object system part and the information sphere. Within the object system part there are four basic fundamental concepts: object o , property p , object relation r and time t . These define the derived concept elementary constellation, (or e-constellation) that can be either of property type $\langle o, p, t \rangle$ or relational type $\langle \langle o_1, \dots, o_n \rangle, r, t \rangle$, where r is a relation of order n and $\langle o_1, \dots, o_n \rangle$ is a n -tuple of objects. Most object relations are binary. Orders above three are very rare, since complex relations can usually be broken down into more primitive ones.

12.4.1 Object

An object o can be either atomic or compound. Atomic objects are fundamental, i.e formally unspecified. Compound objects are formally defined in terms of other entities⁴⁹. Important events in the life-cycle of an object are its creation, changes and destruction. There is a problem with identifying objects. How can we tell one from another, if they have exactly the same properties? Objects are identified with respect to their properties. Artificial objects are usually given an object identifier (oid) or identifying property, such as a name, a registration number etc. The identifying property usually belongs to the object until it is destroyed.

12.4.2 Property

A property p distinguishes something interesting about a particular object o . P is the set of properties for one particular infological model. Properties are generative under a closure restriction:

If p_1, \dots, p_n are elements in the set of properties, P , of a particular infological model, and g is an element in the set of property generation rules, G , for the same model, and $g(p_1, \dots, p_n) = p_{n+1}$, then $p_{n+1} \in P$.

A P-generator is any subset of P which together with the property generation rules, generates P . A P-basis is a minimal P -generator.

12.4.3 Object-relation and object group

The set R is the set of object-relations for one particular infological model. Similarly, as for properties, object relations are generative under a closure restriction. There is an R-generator, and an R-basis.

49. The terms "entity" is used as a general term for more specific object system entities such as object, property, e-constellation etc.

An object group $O(p)$ is generated by all objects o that have a certain property p . Object types, having a stable set of properties constitute a classification.

12.4.4 Attribute

A set of properties is defined to be an attribute A , if there is an object group $O(p)$ for which A is relevant, according to a set of more precise definitions not presented here. Intuitively, an attribute is what we usually expect it to be. The elements v_i are called the values of A .

An identifying attribute A is a single valued attribute that distinguishes objects from each other. A generally identifying attribute A uniquely identifies single objects o within all object groups $O(p)$.

12.4.5 Attribute and relational e-constellation types

Definition : If x is an n -tuple $\langle O_1(p), \dots, O_n(p) \rangle$ of object groups $O_i(p)$, and y is an attribute or an object relation, then the pair $\langle x, y \rangle$ is called an elementary constellation type. x is called the object component and y the predicate component of the e-constellation type.

There are two special cases of elementary constellation types. If O is an object group, and A is an attribute then the pair $\langle O, A \rangle$ is called an attribute e-constellation type (aect). If $\langle O_1, \dots, O_n \rangle$, is an n -tuple of object groups, and R is an n -ary object relation, then the pair $\langle \langle O_1, \dots, O_n \rangle, R \rangle$ is called a relational e-constellation type (rect).

Definition : If $\langle x, y \rangle$ is a valid e-constellation type, and t is a time, and there is a valid e-constellation $\langle x_i, y_j, t \rangle$ corresponding to $\langle x, y \rangle$, then $\langle x, y, t \rangle$ is said to be a valid time version of the e-constellation type $\langle x, y \rangle$.

A valid e-constellation type has a formal definition that in its essence tells it to be the type of some e-constellations that have valid self-contained meanings in a particular domain. e-constellation types constitute a basis for the design of the database implementation model, where the number of time versions⁵⁰ of each type is an important design parameter.

12.5 Information entities

Information entities are the components of a message which must be provided to in order to successfully transfer information to a user.

50. Could also be described as the number of instances of each type, using computer science terminology.

12.5.1 Reference

The basic, formally undefined concept within the information sphere of the infological frame-work is reference ρ . A reference points to a target which can be any instance of a fundamental or derived object system concept.

A reference may be either explicit or implicit. An explicit reference refers directly to its target. An implicit reference is given as a reference expression $\text{expr}(\rho_1, \dots, \rho_n)$ that is isomorphic with $g(e_1, \dots, e_n)$, where g is a generation rule that tells how to generate the target entity given the object system entities⁵¹ e_1, \dots, e_n .

Among explicit references we distinguish between:

- universal names, which refer to a unique object system entity, independently of the infological context, i.e. the environment where the reference occurs.
- local names, which are unique in a particular infological context.
- quasi-names, which are non-unique references, tentatively used where the infological context requires a unique name. Further interaction may resolve the target or possible targets of the quasi-name.

ambiguous references which form the complement of the other classes.

References ρ can be classified according to the type of their target. If it is inferable that the target of ρ belongs to category x , where x is an object system entity (e.g. an object, property, e-constellation etc.), we say that ρ is an x -reference. If for ρ there is any valid infological context making ρ an x -reference, then ρ is a potential x -reference.

12.5.2 Message

A message is used to transfer information. Messages can be broken down into elementary messages which transfer one elementary piece of information, e.g. an e-constellation. The definition of a complete elementary message is:

Definition : If $\langle x, y, z \rangle$ is a reference expression, where x is a n -tuple of locally unique object identifiers, y refers locally uniquely to a generating property⁵² or generating object relation, and z is a locally unique time reference, then $\langle x, y, z \rangle$ is called a complete elementary message, a complete e-message. x is called the object component, y the predicate component and z the time component of the e-message.

51. The terms "object system entity" and "information entity" are used as general terms for the more specific ones defined in the object system part and information sphere.

12.5.3 Message types / elementary concepts

Messages can be ordered under message types.

Definition : If x is an n -tuple of object group references, and y is an attribute reference or an object relation reference, then the pair $\langle x, y \rangle$ is called an elementary message type (e-message type) or an elementary concept (e-concept). x is called the object component and y the predicate component of the e-message type.

There are two special cases of e-concepts. If $p(O)$ is an object group reference, and $p(A)$ is an attribute reference then the pair $\langle p(O), p(A) \rangle$ is called an attribute e-message type (**aemt**). If $\langle p(O_1), \dots, p(O_n) \rangle$, is an n -tuple of object groups, and $p(R)$ is a reference to an n -ary object relation, then the pair $\langle \langle p(O_1), \dots, p(O_n) \rangle, p(R) \rangle$ is called a relational e-message type (**remt**).

An e-concept (e-message type) $\langle x, y \rangle$ is defined as meaningful if and only if $\langle x, y \rangle$ refers to a valid e-constellation type. When we design and use data-bases we need to talk about instances of the same e-concept. Sundgren defines one important variant⁵³ of instances as different time versions of an e-concept. A meaningful time version of an e-concept is defined in the following way:

Definition : If $\langle x, y \rangle$ is a meaningful e-concept, and t refers to a time, and there is a specific meaningful e-message $\langle x_i, y_i, t \rangle$ corresponding to $\langle x, y \rangle$ then $\langle x, y, t \rangle$ is said to be a meaningful time version of the e-concept $\langle x, y \rangle$.

For static systems, the time component in e-constellations may be omitted.

When developing product models, it would induce a significant overhead and complexity if the time component had to be specified for each e-constellation.

Appendix G presents a detailed example that shows how the concepts in this chapter can be applied to a simple product model of a car braking system. For the e-constellations presented in Figure 53 on page 230, the time component is omitted.

52. A generating property is a property which is a member of the explicitly specified property generator of the particular infological model. Similarly a generating object relation is an object relation which is a member of the object relation generator. Recall that the infological model is specified in terms of generators which are formed from two parts: a basis of object system entities and sets of formal generation rules. A rule in a property generator can, for instance, define that **if** $\langle o, \text{person}, t \rangle$ and $\langle o, \langle \text{weight}, X \rangle, t \rangle$ and $(X > 90)$ **then** $\langle o, \text{heavy}, t \rangle$.

53. There are other variants of instances of e-concepts. For instance $\langle o_1, \langle A \rangle \rangle$, where o_1 is a specific object.

12.6 Transmission oriented information theory

Information theory is a very important engineering discipline for applications in digital communications [Haykin 88]. The definitions in this section are included to provide an understanding of the information transmission theoretic measure of information - the binary unit “bit”.

In 1948, Claude Shannon published a landmark paper entitled “A Mathematical Theory of Communication.” [Shannon 48]. The paper provided the foundation of a scientific core for mathematical modeling and analysis of communication systems, widely known as “Information Theory”. Given an information source and a noisy channel, information theory provides limits on (1) the minimum number of bits per symbol required to fully represent the source, and (2) the maximum rate at which reliable communication can take place over the channel.

Shannon separated his models of communication from the semantic aspects, which is apparent in the following quote:

*“The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. Frequently the messages have **meaning**; that is they refer to or are correlated according to some system with certain physical or conceptual entities.*

These semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual message is one selected from a set of possible messages.

The system must be designed to operate for each possible selection, not just the one which will actually be chosen, since this is unknown at the time of design.”

Suppose we have a discrete source emitting symbols every unit of time (signaling interval). The source output is modelled as a discrete random variable S , which takes on symbols from a fixed finite alphabet

$$(EQ\ 2) \quad A = \{s_0, s_1, \dots, s_{K-1}\}$$

with probabilities

$$(EQ\ 3) \quad P(S = s_k) = p_k, \quad k = 0, 1, \dots, K-1$$

The set of probabilities must (of course) satisfy the condition

$$(EQ\ 4) \quad \sum_{k=0}^{K-1} p_k = 1$$

We define the amount of information gained after observing the event $S = s_k$, which occurs with probability p_k , as the logarithmic function

$$(EQ\ 5) \quad I(s_k) = \log_2\left(\frac{1}{p_k}\right)$$

The unit of information is called the bit⁵⁴ (from binary unit). When we have $p_k = 1/2$, we have $I(s_k) = 1$ bit. Hence, one bit is the amount of information that we gain when one of two possible and equally likely events occurs.

(EQ 5) exhibits the following important properties:

$$(EQ\ 6) \quad I(s_k) = 0 \text{ for } p_k = 1$$

obviously, if we are absolutely *certain* of the outcome of an event, even before it occurs, there is *no* information gained.

$$(EQ\ 7) \quad I(s_k) \geq 0 \text{ for } 0 \leq p_k \leq 1$$

The occurrence of an event $S = s_k$ either provides some or no information, but never brings about a loss of information.

$$(EQ\ 8) \quad I(s_k) > I(s_i) \text{ for } p_k < p_i$$

That is, the less probable an event is, the more information we gain when it occurs.

$$(EQ\ 9) \quad I(s_k s_i) = I(s_k) + I(s_i) \text{ if } s_k \text{ and } s_i \text{ are statistically independent.}$$

The information of two symbols in sequence is the sum of the information of each symbol.

Given (EQ 3) and (EQ 5), the mean value of $I(s_k)$ over the source alphabet A is

$$(EQ\ 10) \quad H = \sum_k p_k \log_2(1/p_k)$$

H is the entropy of a discrete memory-less source with alphabet A, and it is a measure of the average information content per source symbol.

54. Note that the term “bit” (of course) also is used to denote a binary digit in a sequence of 1s and 0s.

13 A Design Space Spanning Benchmark Domain Model

This chapter presents a benchmark domain model that has been designed to cover the design space of the primitives used for developing large scale product modeling systems. It is placed before chapter 14, "Primitives for Domain Models" to give an example to instantiate the primitives on.

The benchmark domain model covers the domain of an engineering company that designs products that can be represented by hierarchical article structures (part-of relationship) with references to included articles (uses relationship). It has a single inheritance hierarchy (isa-relationship), and includes a one to many relationship to a superclass where the instances can be of different subclasses. This type of relationship is common in product models, for instance when a part-of relationship can own components of slightly different types.

The benchmark domain model also covers hierarchical network activity planning. The nodes of the planning graph represent activities and the edges represent deliverables that are output from one activity and input to another.

The article structure and network planning graphs are very common in product models. Large and complex instances of graphs for benchmarking purposes can be efficiently documented as a recursive algorithm using a random number generator with a known seed and thus number sequence.

13.1 Object model diagram of the benchmark domain model

Figure 32 on page 108 shows an object model diagram of the benchmark domain model. The example presented in Section 9.5 on page 72 explains the syntax of the diagram and the entity *DatabaseObject*.

DesignationObject inherits from database object. Its attribute *ref_id* serves as a human readable identifying attribute and *name* provides a descriptive name for the object. The subclass *QualitySecuredObject* manages quality assurance information in the attributes *checked_by*, *approved_by*, *release* and *status*.

Benchmarks on long series of such browsing paths will be able to distinguish access times in different interface configurations with statistical significance. See Appendix F on page 223 for a browsing path example.

Object system entities in the benchmark domain model can be expressed in infological terminology. For instance $\langle \textit{Product}, \textit{delivery_date} \rangle$ is an attribute e-constellation type (**aect**).

$\langle \langle \textit{Company}, \textit{Department} \rangle, \textit{company_departments} \rangle$ is a relational e-constellation type (**rect**).

13.2 Overview of the benchmark domain model

This section presents some of the details of the benchmark domain model that provide examples for the concepts described in more detail in the following chapters.

An instance of the benchmark domain model typically contains one *Company* that has several *Departments* through the 1-N relationship *company_departments*. Each *Department* has many products, and each *Product* may hold a set of *ProductData* instances through the 1-N relationship *product_productData*.

The class *ProductData* shown in Figure 32 contains attributes of the basic data types of the CORBA⁵⁵ interface definition language (IDL) [CORBA 91]. A user interface compiler must generate attribute editors for each of these data types. The *ProductData* class is useful for statistical measurements of attribute value manipulation times for different implementations of [attribute editors](#).

Each product may have one main *Article* through the 1-1 relationship *product_mainArticle*. Articles may own other articles through the 1-N relationship *owner_owns*. They may also include other article definitions in the assembly through *IncludedArticle* objects which implement an M-N relationship between articles. The attribute *quantity* specifies the number of included articles in the assembly.

The *Company* holds all its *Drawings* through the 1-N relationship *company_drawings*. Figure 33 shows an example of a form-based [object editor](#) for instances of the class *Company*. The editor provides access to the attribute values of all inherited attributes from *DatabaseObject.highId* down to *DesignationObject.name*. Instances of the 1-N relationships *company_departments* and *company_drawings* can be edited through the listbox implementation of relationship set editors identified with the labels *Departments* and *Drawings*.

The path *Company.departments->Department.products->Product.mainArticle->Article* represents a typical hierarchical browsing path over 1-N relationships that needs to be efficiently implemented by a user interface

55. Common Object Request Broker Architecture

compiler. Figure 34 shows an example of a hierarchical browser for this path. Users can easily click their way down the hierarchy, and read and edit the values of the attributes *ref_id* and *name* for instances of *Company*, *Department*, *Product* and the products corresponding mainArticle directly. The classes *Product* and *Article* have the attributes *delivery_date* and *quantity*. These have their own labelled attribute editors in the browser.

The image shows a window titled "Company" with a menu bar (Object, Edit, Info, Window, Help). The window is divided into several sections. On the left, there are input fields for attributes: HighId (41), LowId (105979477), DtAdded (04/18/1996 14:44:37.000), DtModified (04/21/1996 18:58:58.000), Created_by (nobody), Modified_by (nobody), Ref_id (co1), and Name (Company1). On the right, there are two lists: "Departments" with one entry "de1 Department1" and "Drawings" with three entries "dr1 Drawing1", "dr2 Drawing2", and "dr3 Drawing3". Each list has buttons for Create, Add, Edit, Remove, and Delete. At the bottom left, there is a "Select" checkbox and buttons for OK, Store, and Cancel.

FIGURE 33. Example of an object editor window for instances of the class Company within the benchmark domain model.

The image shows a window titled "CompanyBrowser" with a menu bar (File, Edit, Create, Export, Info, Window, Help). The window is divided into several sections. At the top, there are four input fields for Ref_id and Name, each with a "Store" button: Ref_id c1, Name Company1; Ref_id d1, Name Department1; Ref_id p1, Name Product1; Ref_id a, Name Article1. Below these are four lists: "Departments" (d1 Department1, d2 Department2, d3 Department3), "Products" (p1 Product1, p2 Product2, p3 Product3, p4 Product4), "ProductData" (t1 For test t1, t2 For test t2), and "Owns" (a1 Article a1, a2 Article a2, a3 Article a3). There is also a "MainArticle" section with a list containing "a Article1" and a "Browse" button. At the bottom right, there is an "Includes" section.

FIGURE 34. Example of a form-based browser window for the path Company down to Article in the benchmark domain model.

Figure 35 shows an example of a form-based hierarchy browser for an Article and its hierarchical article structure implemented through the 1-N relationship *owner_owns*. By selecting an article in the relationship set editor labelled by Articles, the browser sets the object editor to the right to display this article. Through the relationship set editor labelled "Includes", the user has access to instances of IncludedArticle related to the current

article through the 1-N relationship *article_includes*.

FIGURE 35. Example of a form-based browser window for the hierarchical structure Article->owner_owns.

Figure 36 shows a graphical 2D view of the same article structure as displayed in the Articles *relationship editors* in Figure 35. The different article instances are accessible through the rectangular *object editors* in the nodes of the graph. The editors are click sensitive and provide popup menus with available operators. The attribute values for *ref_id*, *name* and *quantity* can be accessed through *attribute editors* that cover the top left, top right and bottom right corner of an object editor. For interpreting the information in this 2D view correctly, the user has to have learned the interpretation rules that the location of the attribute values determines the attribute constellation type to which the displayed value belongs.

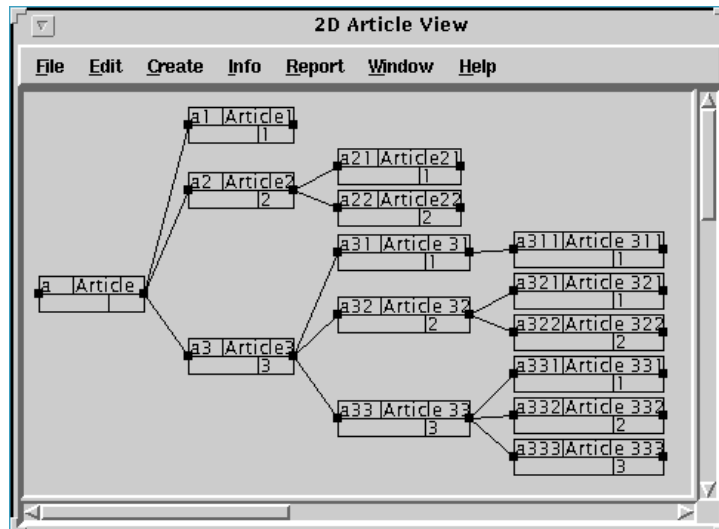


FIGURE 36. Example of a 2D browser for hierarchical 1-N relationships.

In the benchmark domain model each product can have a main DesignActivity connected through the 1-1 relationship *product_mainActivity*.

There is one special 1-N relationship *product_features*, which has the special feature that instances on the N-side may be any of the subclasses for the class C. When creating a new instance on the N-side of this kind of relationship, the user must provide information about which subclass to instantiate. A user interface compiler providing support for implementing this feature must be able to detect this case.

Appendix A has tables with additional details of the benchmark domain model.

The next chapter describes a theory for calculating information quantity in complex object structures. The theory enables interactive calculations of the type presented in Figure 37.

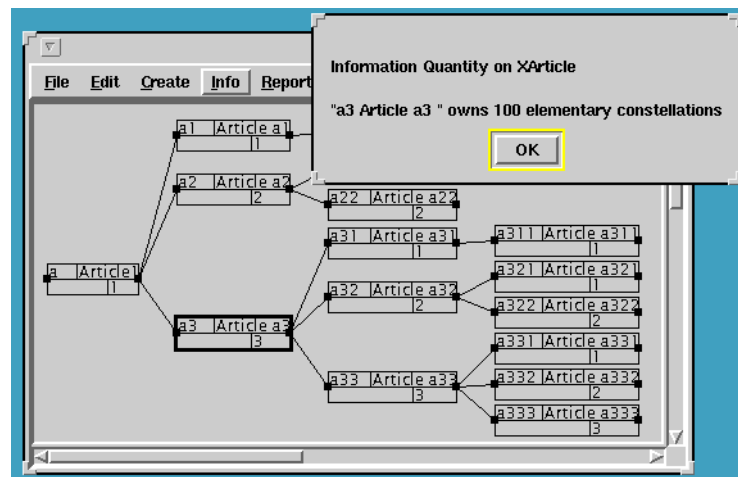


FIGURE 37. Interactive calculation of the information quantity owned by an Article object. In this view 40 e-messages are visible that represent e-constellations owned by “a3 Article a3”.

14 Primitives for Domain Models

This chapter describes fundamental primitives for the design of domain models and how these primitives relate to infological theory. The primitives are a subset of the ones described in section 9.7 "The meta-database domain model" on page 76. They are selected to minimize complexity and be straightforward to implement in relational- and object-oriented databases using declarative source code generators.

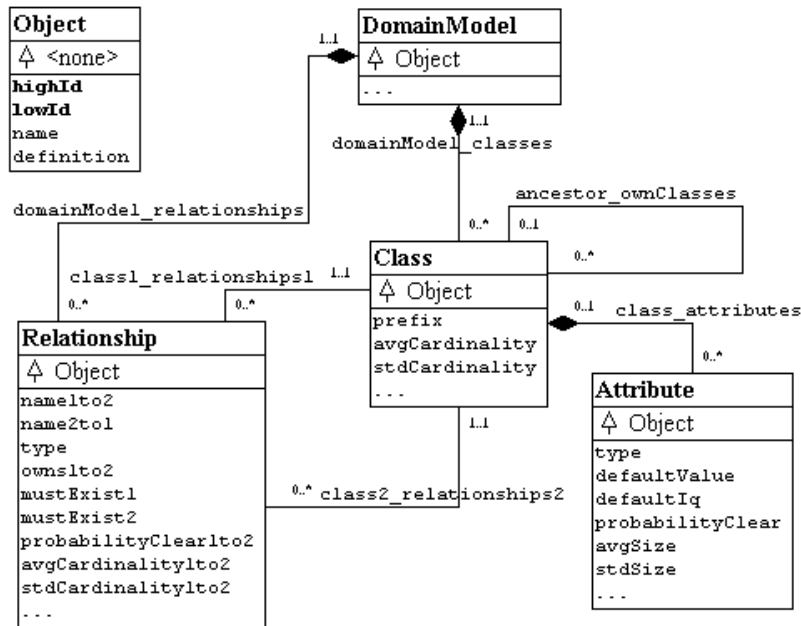


FIGURE 38. View of the meta-database domain model, showing domain model meta-data used for performance calculations.

Figure 38 shows a subset of the domain model meta-data in the meta-database that is used for information quantity calculations and prediction of database- and user interface performance requirements. In this chapter, the concepts Class, Attribute, Relationship of types 1-1, 1-N & M-N will be described in more detail and related to infological theory.

Primitive operations on classes, attributes and binary relationships are also

defined. Besides the function, each operation has a description of how it influences the information content of an instantiated domain model calculated in its quantity of e-constellations. The primitive operations are atomic. That is, the implementation must make sure that they are either performed completely or not at all.

One of many benefits of the exact quantification measures of instantiated domain models is that it makes it possible to compare the size and complexity of different product models and domain models from different engineering domains. This comparison mechanism can, for instance, support project planning and cost estimation for developing PMSs and KBSs for new domains.

14.1 Class

A class specifies a classification of object types within the domain that are important enough to have their own name, definition and possibility to be accessed as one unit. Our meta-database definition of the domain model concept *Class* was given in section 9.7.5 on page 77. Table 7 describes class attributes that should be defined in the domain model. Table 8 specifies class operations that must be available in the delivery system, and Table 9 operations that must be available on instances of a class.

Table 7: Class attributes.

| Name | Description |
|-----------------------|---|
| <i>name</i> | A descriptive name of the class. |
| <i>definition</i> | A definition of the class in one or a few sentences. |
| <i>prefix</i> | A short string prefix that uniquely identifies the class within the domain model. |
| <i>avgCardinality</i> | Estimated average number of instances of this class. |
| <i>stdCardinality</i> | Standard deviation of cardinality. |

Table 8: Class operations.

| operation name | Description |
|--------------------------|--|
| <i>C.new(oc)</i> | Create an new instance of a class, maintaining all referential integrity- and cardinality constraints of the domain model. oc is an ordered collection with objects needed to fulfill the cardinality constraints imposed by relationships for instances of class C. |
| <i>C.instanceAt(oid)</i> | Return the instance with the given object-identifier oid. |

Table 8: Class operations.

| operation name | Description |
|------------------------------|--|
| <i>C.allAttributes()</i> | Return the set of all attributes that belong to the class, including inherited attributes. |
| <i>C.allRelationships1()</i> | Return a set of all relationships for which the class is on side 1. |
| <i>C.allRelationships2()</i> | Return a set of all relationships for which the class is on side 2. |

Table 9: Class instance operations.

| operation name | Description |
|----------------------|---|
| <i>o.localIq()</i> | Return the local information quantity for the object. See section 14.1.5 on page 116. |
| <i>o.iq()</i> | Return the information quantity owned by the object including its part-of relationships. See section 14.1.6 on page 116 |
| <i>o.canDelete()</i> | Return true if the object can be deleted without violating any cardinality constraints defined in the relationships. |
| <i>o.delete()</i> | Delete the object and all objects in its owned part-of relationships, while maintaining referential integrity. See section 14.1.7 on page 117 |

14.1.1 C.new(oc)

Create a new instance of a class, maintaining all referential integrity- and cardinality constraints of the domain model. *oc* is an ordered collection with references to objects which must exist in a relationship to instances of *C* in order to fulfill the cardinality constraints of *C*'s relationships. If the cardinality constraints do not require instances of *C* to be owned or referenced by other objects, *oc* is an empty collection.

Information quantity change: The added number of e-constellations are calculated as the aggregate of all defined default attribute values for the class plus the information quantity in the relationships to the supplied objects in *oc*. If *C* has owned objects through a part-of relationship which must have at least one object to fulfill the cardinality constraints, these objects are created automatically and their information quantity added.

(EQ 15) shows how to calculate the added information quantity for the new instance.

14.1.2 C.allAttributes()

Return the set of all attributes that belong to the class.

$$(EQ\ 11) \quad C.allAttributes() = \left(\bigcup_{\forall AC \in ancestors(C)} AC.allAttributes() \right) \cup C.attributes()$$

where `ancestors(C)` returns a union of the owner superclass and mix-in super classes that are inherited through the priority ordered M-N relationship implemented by *AncestorSubClass* in Figure 38 on page 113.

14.1.3 C.allRelationships1()

Return the set of all relationships for which C is class1.

(EQ 12)

$$C.allRelationships1() = \left(\bigcup_{\forall ac1 \in ancestors(C)} ac1.allRelationships1() \right) \cup c.relationships1()$$

14.1.4 C.allRelationships2()

Return the set of all relationships for which C is class2.

$$(EQ\ 13) \quad C.allRelationships2() = \left(\bigcup_{\forall ac2 \in ancestors(C)} ac2.allRelationships2() \right) \cup c.relationships2()$$

14.1.5 o.localIq()

The local information quantity *localIq()* for an object o of class C is calculated as the sum of the information quantities held by all attributes (inherited and local) plus all relationship references.

(EQ 14)

$$o.localIq() = \sum_{\forall a \in class(o).allAttributes()} o.iq<a>() + \sum_{\forall r21 \in class(o).allRelationships2()} o.iq<r21>()$$

where `o.iq<a>()` returns the information quantity of the attribute value. (Section 14.2). The information quantity returned by `o.iq<r21>` is given in (EQ 22), (EQ 23) on page 122 and (EQ 33) on page 125.

14.1.6 o.iq()

The information quantity (iq) is calculated as the local information quantity *localIq()* (EQ 14) for an object plus the information quantity contained in the part-of structures in owned relationships. Relationships can by definition only be owned in the 1to2 direction. See Section 14.3 on page 119.

$$(EQ\ 15) \quad o.iq() = o.localIq() + o.ownedIq()$$

$$(EQ\ 16) \quad o.ownedIq() = \sum_{\forall r12 \in class(o).allRelationships1()} o.iq<r12>()$$

14.1.7 o.delete()

Delete an instance of the class and all objects in its owned (part-of) relationships, forcing the maintenance of all referential integrity- and cardinality constraints of the domain model. Warnings to the user has to be issued before this operation is performed, for instance by checking with the testing operation *canDelete*.

A delete operation can never leave the product model in an illegal state. A bad design of cardinality constraints on the relationships may lead to a rippling deletion of a whole product model. The information quantity change is $-o.iq()$ before the delete operation is performed, if no cardinality constraints enforce deletions of associated objects. These restrictions also apply to the delete operations on relationships specified later in this chapter.

14.2 Attributes

An attribute defined in the domain model documents an attribute e-constellation type that belongs to objects of a particular class.

Sometimes during the development of a domain model we would like to document some concepts as attributes that do not map directly to an attribute e-constellation type. An example of such an attribute is a long text field containing a textual definition, comment or remark. To handle such informal attributes in our theoretical framework, it is possible to specify a default information quantity that should be added to the information quantity of the object if the attribute has a value; see Table 12

If there is a need to extract the information quantity of a textual sentence, the knowledge engineer has to write a parser that translates the text into a parse tree, where the nodes are instances of classes in the domain model and the information quantity can be calculated according to the framework from the root of the parse tree.

Table 10 specifies the attributes of an *Attribute* specified in the domain model. Table 12 specifies the operations that should be available for each attribute on objects of a particular *Class*.

Table 10: *Attribute* attributes.

| Name | Description |
|-------------------|--|
| <i>name</i> | A descriptive name for the attribute. |
| <i>definition</i> | A definition of the attribute in one or a few sentences. |

Table 10: *Attribute* attributes.

| Name | Description |
|-------------------------|---|
| <i>type</i> | The data type of the attribute, if it is not inherited from the TypeDef, or Property. See Figure 21 on page 82. |
| <i>defaultValue</i> | A default value, if it is not inherited from the TypeDef, or Property. |
| <i>defaultIq</i> | Default information quantity for an attribute value. Usually 1. |
| <i>probabilityClear</i> | Probability that this attribute has been cleared or never got a value set by the user. |
| <i>avgSize</i> | Estimated average size in a convenient measure. Usually the number of characters in a character representation of the value. The average size can be used for calculating the estimated average time for a user to enter a value. |
| <i>stdSize</i> | Standard deviation of size. |

Table 11: *Attribute* operations on class C.

| operation name | Description |
|-------------------------------------|---|
| <i>C.default<attname>()</i> | Return a default value for the attribute. |
| <i>C.defaultIq<attname>()</i> | Return the <u>default information quantity</u> for an attribute that has a value. Is 1 in most cases. |

Table 12: *Attribute* operations on objects of class C.

| operation name | Description |
|-------------------------------|--|
| <i>o.get<attname>()</i> | Return the current value of the attribute. If the value was never set and no default value has been defined, raise an exception where the user has the option to select or set a value. If the exception functionality is not implemented, the source code generator should report an error. |
| <i>o.iq<attname>()</i> | Return the information quantity of the attribute. If the attribute has a value, the default implementation should return the <i>defaultIq</i> specified in the domain model, else it should return 0. |
| <i>o.has<attname>()</i> | Return true if the current attribute value carries information, i.e. <i>iq<attname>()</i> would return a non zero value. |

Table 12: *Attribute* operations on objects of class C.

| operation name | Description |
|----------------------------------|--|
| <i>o.create<attname>()</i> | Set the attribute value to the default value specified in the domain model. |
| <i>o.delete<attname>()</i> | Sets the attribute value to undefined, or a value that makes <i>o.iq<attname>()</i> return 0. |
| <i>o.set<attname>(v)</i> | Set the attribute value to the supplied value v. If the value violates the defined value domain, raise an exception where the user has the option to supply a legal value. |

14.3 Relationships

Section 9.7.6 on page 77 gave a short description of how relationships are documented in our domain models. Relationships are binary since this considerably simplifies the theory and also the transformation of domain models to implementations. It is well-known to all practitioners of information systems modeling that ternary relationships and higher can be transformed to a set of binary relationships by introducing an extra common class. Having relationships of higher degree in the domain model modelling language is a matter of convenience.

In our domain modelling language, a relationship has two ends, 1 and 2. A 1-N relationship always has the N side on the side denoted 2. This design rule puts no restrictions on expressiveness, but significantly simplifies the implementation of source code generators.

The classes at the 1-and 2-side of the relationship are called class1 and class2. A binary relationship can be viewed and implemented as attributes in class1 and class2. The name of the binary relationship attribute in class1 is called name1to2, and the name in class2, name2to1. Relationship operations have a naming convention that includes these names (see Table 14 on page 121). <r12> is substituted by name1to2 and <r21> by name2to1. <rXX> can be substituted by either <r12> or <r21>.

A relationship may be a part-of or association relationship. For part-of relationships objects of class2 are the parts and objects of class1 are owners of the parts. This direction convention simplifies the information quantity analysis of an instantiated domain model significantly. Part-of relationships have the *Relationship* attribute owns1to2 set to true.

The difference between part-of and reference relationships is that the information quantities of objects in part-of relationships are accumulated in the 2to1 direction. That is, the information quantity held by objects of class2 for a part-of relationship belongs to the object of class 1.

In a reference relationship, the information quantity of the relational e-constellation is owned by the instance of class2. This design decision was

made since in foreign key implementations of 1-N relationships in relational databases the foreign key is placed in the table on the 2-side.

Table 13: *Relationship* attributes.

| Name | Description |
|-----------------------------|---|
| <i>name</i> | An identifier for referring to the relationship. From our experience, a good naming convention is to name the relationship “<name2to1>_<name1to2>”. |
| <i>definition</i> | A definition of the relationship in one or a few sentences. |
| <i>name1to2</i> | The name used when referencing the relationship from class1. Substitutes <r12> in the operation names. |
| <i>name2to1</i> | The name used when referencing the relationship from class2. Substitutes <r21> in the operation names. |
| <i>owns1to2</i> | Set to TRUE if the relationship is a part-of relationship. |
| <i>type</i> | The type can be 1-1, 1-N or M-N. |
| <i>mustExist1</i> | Flag set to true, if there must exist an instance of class1 for each instance of class2. |
| <i>mustExist2</i> | Flag set to true, if there must exist at least one instance of class2 for each instance of class1. |
| <i>probabilityClear1to2</i> | Probability that an instance of class1 does not participate in the relationship. |
| <i>avgCardinality1to2</i> | Estimated average cardinality of the set of instances of class2 for instances of class1 that have this relationship. |
| <i>stdCardinality1to2</i> | Standard deviation for the <i>avgCardinality1to2</i> . |

The *Relationship* attributes only store statistics for the direction 1to2. The corresponding statistic measures *probabilityClear2to1*, *avgCardinality2to1* are constrained by a set of equations:

$$(EQ\ 17) \quad \text{avg}(|R|) = (1 - p_{0_{1to2}}(R)) \times \text{avg}C_{1to2}(R) \times \text{avg}(|C_1|)$$

$$(EQ\ 18) \quad \text{avg}(|R|) = (1 - p_{0_{2to1}}(R)) \times \text{avg}C_{2to1}(R) \times \text{avg}(|C_2|)$$

$$(EQ\ 19) \quad p_{0_{2to1}}(R) = 1 - \left(\frac{\text{avg}(|R|)}{\text{avg}(|C_2|) \times \text{avg}C_{2to1}(R)} \right)$$

$$(EQ\ 20) \quad \text{avg}C_{2to1} = \frac{\text{avg}(|R|)}{\text{avg}(|C_2|) \times (1 - p_{0_{2to1}}(R))}$$

where R denotes the *Relationship*, $\text{avg}(|R|)$ the average number of relational e-constellations, $p_{0_{1to2}} = \text{probabilityClear1to2}$, $\text{avg}C_{1to2} = \text{avgCardinality1to2}$, $\text{avg}(|C_1|)$ is *avgCardinality* for class 1.

For 1-1 and 1-N relationships *avgCardinality2to1* is always 1. For M-N relationships *probabilityClear2to1* and *avgCardinality2to1* are constrained by (EQ 21).

$$(EQ\ 21) \quad (1 - p_{0_{2to1}}(R)) \times \text{avg}C_{2to1}(R) = \frac{\text{avg}(|R|)}{\text{avg}(|C_2|)}$$

14.4 Relationship operations

The operations described in this section should be provided for both class1 and class2 and operate on a (virtual or real) attribute implementation of the relationship. 1-1, 1-N, and M-N relationships have the same set of operations to provide compatibility for algorithms. In the operation naming convention $c.\langle\text{opname}\rangle\langle\text{rXX}\rangle(\langle\text{parameters}\rangle)$ c can be either class1 or class2, $\langle\text{opname}\rangle$ is the operation name and $\langle\text{rXX}\rangle$ the name of the relationship reference attribute in direction 1to2, or 2to1 that is name1to2 or name2to1.

Table 14: 1-1 *Relationship* operations in both directions.

| operation name | Description |
|---|---|
| $o.\text{all}\langle\text{rXX}\rangle()$ | Return a collection holding the object on the other side of the 1-1 relationship. If there is no object, return an empty collection. |
| $o.\text{size}\langle\text{rXX}\rangle()$ | Return an integer holding the size of the collection returned by $c.\text{all}\langle\text{rXX}\rangle()$ |
| $o.\text{get}\langle\text{rXX}\rangle()$ | Return the object on the other side of the relationship. |
| $o.\text{iq}\langle\text{rXX}\rangle()$ | Return the information quantity owned by the relationship. |
| $o.\text{has}\langle\text{rXX}\rangle()$ | Return true if the relationship holds an object. |
| $o.\text{canAdd}\langle\text{rXX}\rangle(\text{oid})$ | Return true if oid can be added to the relationship without violating any cardinality constraints. |
| $o.\text{create}\langle\text{rXX}\rangle(\text{oc})$ | Return a new object of the opposite class, which is created and added to the relationship. oc is an ordered collection with references to objects in other relationships in which the new object must participate to fulfill the cardinality constraints. |
| $o.\text{add}\langle\text{rXX}\rangle(\text{oid})$ | Add an existing object of the opposite class to the relationship if allowed by cardinality constraints. |

Table 14: 1-1 *Relationship* operations in both directions.

| operation name | Description |
|--|---|
| <code>o.set<rXX>(oid)</code> | Force set the object on the opposite side to oid, with brutal maintenance of cardinality constraints. |
| <code>o.remove<rXX>(oid)</code> | Remove the relationship to the supplied object if it exists. This operation is provided for compatibility with 1-N and M-N relationships. |
| <code>o.removeAll<rXX></code> | Remove the relationship to the related object if there is one. |
| <code>o.delete<rXX>(oid)</code> | Delete the supplied object if it is contained in the relationship. This operation is provided for compatibility with 1-N and M-N relationships. |
| <code>o.deleteAll<rXX>(oid)</code> | Delete the object on the other side of the relationship if possible. |

The following subsections describe some 1-1 relationship operations that calculate or influence the information quantity of the product model.

14.4.1 **o.iq<rXX>()**

Return the calculated owned information quantity for the relationship according to the following conditions.

(EQ 22) $\text{not}(\text{o.has}\langle\text{rXX}\rangle()) : 0$

(EQ 23) $\text{o.has}\langle\text{r21}\rangle() : 1$

(EQ 24) $\text{o.has}\langle\text{r12}\rangle() \ \& \ \text{not}(\text{owns1to2}(\text{r})) : 0$

(EQ 25) $\text{o.has}\langle\text{r12}\rangle() \ \& \ \text{owns1to2}(\text{r}) : \text{o}_2.\text{iq}()$

where o_2 is the object on the 2-side of the relationship.

14.4.2 **o.create<rXX>(oc) -> oid**

Create an object of the opposite class and include it in the relationship if there is not already an object in the relationship. If the operation succeeds, the information quantity change is the number of e-constellations owned by the newly created object. oc is an ordered collection with any objects that the newly created object must have relationships to in order to fulfill the cardinality constraints.

14.4.3 **o.add<rXX>(oid)**

Add an existing object of the opposite class to the relationship if possible. Cardinality restrictions may prevent an add operation to be completed. If

the operation is successful the information quantity change is 1.

14.4.4 **o.set<rXX>(oid)**

Force set the object on the opposite side to oid (if legal) with “brutal” maintenance of all referential integrity constraints. Brutal means removing or deleting any previous object in the relationship for both o and oid, and forcing all cardinality constraints which may lead to the deletion of several other objects attached by cardinality constraints. This is a way to avoid the need for a special garbage collection algorithm. The set-operation may take **nil**⁵⁶ as its parameter value if **nil** is available in the implementation language.

Information quantity change:

(EQ 26) $\text{owns1to2}(r) \ \& \ \text{not}(\text{o.canAdd}\langle\text{rXX}\rangle()) : \text{o.iq}() - \text{o}_2.\text{iq}()$

(EQ 27) $\text{owns1to2}(r) \ \& \ \text{o.canAdd}\langle\text{rXX}\rangle() : \text{o.iq}()$

(EQ 28) $\text{not}(\text{owns1to2}(r)) \ \& \ \text{o.canAdd}\langle\text{rXX}\rangle() : 1$

(EQ 29) $\text{not}(\text{owns1to2}(r)) \ \& \ \text{not}(\text{o.canAdd}\langle\text{rXX}\rangle()) : 0$

Where r is the relationship. o is the instance referenced by **oid**. o₂ is the previous instance of class2 held in the relationship. owns1to2(r) is a flag set by the domain model designer and tells which relationships are part-of and which that are “only” associations. The predicate canAdd<rXX> returns true if the current state of the objects and the cardinality constraints would allow an o.add<rXX>(oid) operation to succeed.

14.4.5 **o.remove<rXX>(oid)**

Remove the supplied object from the relationship, if possible. This operation is provided for compatibility reasons. The information quantity change is -1, if the operation was successful.

14.4.6 **o.removeAll<rXX>()**

Remove the relationship to the related object if there is one. The information quantity change will be -1 if there was a related object.

14.4.7 **o.delete<rXX>(oid)**

Delete the supplied object if it is contained in the relationship. This operation is provided for compatibility with 1-N and M-N relationships. DeleteAll does the same thing, but needs no reference. The Information quantity change is

(EQ 30) $-\text{o}_2.\text{iq}()$

⁵⁶. An object representing the NULL-object.

where o_2 is the deleted instance referenced by oid.

14.4.8 **o.deleteAll<rXX>()**

Delete the object on the other side of the relationship if possible. If there is an instance on the other side of the relationship, the information quantity change is the same as for `o.delete<rXX>(oid)`.

14.5 1-N & M-N Relationship operations

1-N & M-N relationships have the same set of operations as 1-1 relationships, but the semantics is adjusted for the augmented cardinality of these relationships.

Table 15: 1-N Relationship operations in both directions.

| operation name | Description |
|---------------------------------------|---|
| <code>o.all<rXX>()</code> | Return an ordered collection holding all objects on the other side of the relationship. If there are no objects, return an empty collection. |
| <code>o.size<rXX>()</code> | Return an integer holding the size of the collection returned by <code>o.all<rXX>()</code> . |
| <code>o.get<rXX>()</code> | Return the first object on the other side of the relationship. If there is no object return nil. |
| <code>o.iq<rXX>()</code> | Return the information quantity owned by the relationship. |
| <code>o.has<rXX>()</code> | Return true if the relationship holds at least one object. |
| <code>o.create<rXX>(oc)</code> | Return a new object of the opposite class, which is created and added to the relationship. oc is an ordered collection with references to objects in other relationships in which the new object must participate to fulfill the cardinality constraints. |
| <code>o.add<rXX>(oid)</code> | Add an existing object of the opposite class to the relationship. |
| <code>o.set<rXX>(oid)</code> | The operation is done in three steps. For a part-of relationship, first delete all objects. For a reference relationship first remove all objects. Then remove any object from oid that occupies the place which o will take. Finally add the object referenced by oid to be the single object on the other side. |
| <code>o.remove<rXX>(oid)</code> | Remove the supplied object from the relationship, if possible. |

Table 15: 1-N *Relationship* operations in both directions.

| operation name | Description |
|--|---|
| <code>o.removeAll<rXX>()</code> | Remove all related objects from the relationship. |
| <code>o.delete<rXX>(oid)</code> | Remove the supplied object from the relationship and delete it. |
| <code>o.deleteAll<rXX>(oid)</code> | Delete all objects on the other side of the relationship. |

The following subsections describe some 1-N and M-N relationship operations that calculate or influence the information quantity of the product model.

14.5.1 **`o.iq<rXX>()`**

Return the information quantity within the relationship that is owned by `o`. There are 3 different cases, which are described in (EQ 31) to (EQ 33).

$$(EQ\ 31)\ owns1to2(r) \ \& \ <r12> : \sum_{\forall o_2 \in o.all<r12>()} o_2.iq(i_2)$$

$$(EQ\ 32)\ not(owns1to2(r)) \ \& \ <r12> : 0$$

$$(EQ\ 33)\ <r21> : o.size<r21>()$$

14.5.2 **`o.create<rXX>(oc) -> oid`**

Create a new instance and add it to the relationship. `oc` is an ordered collection with any objects that the newly created object must have relationships to, in order to fulfill the cardinality constraints. The information quantity change is the number of e-constellations owned by the newly created object. Return an `oid` to the new instance.

14.5.3 **`o.add<rXX>(oid)`**

Add an existing object of the opposite class to the relationship if possible. For 1-N relationships, cardinality restrictions may prevent the add operation to be completed. If the operation is successful, the information quantity change is 1.

14.5.4 **`o.set<rXX>(oid)`**

Simulates the `o.set<rXX>` behaviour of an 1-1 relationship. For a part-of relationship, it works as a `deleteAll<rXX>()` followed by an `add<rXX>(oid)`-operation. For a reference relationship it works as a `removeAll<rXX>()` followed by an `add<rXX>(oid)`-operation. The information quantity change can be derived from the `deleteAll`, `removeAll` and `add` operations.

14.5.5 o.remove<rXX>(oid)

Remove the supplied object from the relationship, if possible. The information quantity change is -1 if the operation was successful.

14.5.6 o.removeAll<rXX>()

Remove all objects in the relationship. The information quantity change is $o.size<rXX>()$.

14.5.7 o.delete<rXX>(oid)

Delete the supplied object if it is contained in the relationship. The Information quantity change is:

$$(EQ\ 34) - o_2.iq()$$

where o_2 is the deleted instance referenced by oid.

14.5.8 o.deleteAll<rXX>()

Delete all the objects on the other side of the relationship if possible. The information quantity is given in (EQ 35).

$$(EQ\ 35) \quad \sum_{\forall o_2 \in o.all<r12>()} o_2.iq(i_2)$$

14.6 Higher degree relationships

Some small percentage of the relationships within a domain model may have to be of a higher degree than binary, for instance ternary $\langle\langle O1, O2, O3 \rangle, R \rangle$. A ternary relationship is transformed into a class with three connected binary relationships. Their cardinality constraints may make it impossible to create the simulated relationship-instance with one atomic create<rXX>(oc)- and two add<rXX>-operations, without violating the integrity of the product model database for a moment. In these cases the objects required to fulfil the cardinality constraints have to be supplied in the ordered collection oc that is a parameter of the create<rXX>(oc)-operation.

If instances of ternary relationship constellation types are to be calculated as 1 e-constellation instead of 3, an adjustment in the equations for information quantity change must be made.

In practice, however, most ternary and higher degree relationships will require some attributes, for instance a timestamp when the relationship was created, and thus they must be transformed into a class anyway.

14.7 Cardinality calculation examples

Here follow two simple examples of statistical calculations on the benchmark domain model (Figure 32 on page 108) that summarize the terminology introduced.

14.7.1 1-N relationship department_products.

We have a sample benchmark database with one company (c1) that owns four departments (d1, d2, d3, d4) thus $Department.avgCardinality = 4$. Department d1 has no products. Department d2 owns products p1 and p2. Department d3 owns products p3 and p4. Department d4 owns products p5, p6, p7, p8 and p9. Thus $Product.avgCardinality = 9$.

Since there is one department d1 of the four departments that has no product, $department_products.probabilityClear1to2 = p0_{1to2} = 1/4$. The relationship attribute $department_products.avgCardinality1to2$ is defined to be the average cardinality of products that are owned by the departments that have one or more products. Thus $avgC_{1to2} = (2+2+5)/3 = 3$. The relationship attribute $department_products.stdCardinality1to2$ is the standard deviation. Thus $stdC_{1to2} = \sqrt{(2^2+2^2+5^2-3^2*3)/3} = \sqrt{2}$.

From (EQ 17) we derive : $avg(|R|) = (1-1/4) * 3 * 4 = 9$, there are nine relational e-constellations of type *department_products* in our database. In a 1-N relationship, $avgCardinality2to1$ is 1 per definition.

From (EQ 19) we derive : $p0_{2to1} = 1 - 9/(9*1) = 0$.

14.7.2 Hierarchical 1-N relationship owner_owns.

Figure 36 on page 111 shows the sample population of 15 articles. The relationship cardinality instances are ordered by left-to-right, top-down.

$$p0_{1to2} = 9/15 = 0.6.$$

$$avgC_{1to2} = (3+2+3+1+2+3)/6 = 2.33.$$

$$stdC_{1to2} = \sqrt{(3^2+2^2+3^2+1^2+2^2+3^2-2.33^2*6)/6} = 0.75$$

$$\text{From (EQ 17) : } avg(|R|) = (1-0.6) * 2.33 * 15 = 14$$

$$\text{From (EQ 19) : } p0_{2to1} = 1 - 14/(15*1) = 0.067$$

This kind of statistics can be used by a user interface compiler for selecting view implementation classes for various types of views (Section 18.3.4).

15 An Information-oriented Task Description Language

This chapter introduces a language for information-oriented task descriptions which can be used as a declarative input specification for a heuristic user interface design algorithm.

Section 15.1 introduces some hypothetical measures on user interfaces for object-oriented systems that are a step towards the design of a global goal function for user interface configurations for complex KBS-applications. These will contain hundreds of different forms and graphical browsing views. The introduction motivates the detailed declarative descriptions of the tasks.

The task descriptions describe HOW the information will be used in the PMS or KBS.

15.1 Some hypothetical measures on KBS user interfaces

In KBS-applications which use product models with thousands of components, the user interface for creating and maintaining the models should be optimized for performing the most frequent tasks.

An example of an improving step in such an optimization could be to change the representation of the object editors presented in the 2D Article View in Figure 36 on page 111 to present additional e-messages, for instance by adding attribute editors for the *dtModified* attribute in the lower left corner of the *Article* object editor nodes.

The added information might be needed for performing a certain task. This may save the user many interaction operations that otherwise would be needed to get the necessary information entities on the screen. The views picture the contents of the knowledge-base. To know how to design each type of view, we have to know how it is going to be used, which means knowing the user's tasks.

We assume that the user works according to the ten principles of the model of a human information processor [Card et al. 83]. See Appendix D.

At system design time, the predictable behaviour of a user while completing a task has a number of free and bound "design variables".

The rationality principle decomposes the “high-level” behavior of a user into different influencing components :

P8. Rationality Principle. A person acts so as to attain his goals through rational action, given the structure of the task and his inputs of information and bounded by limitations on his knowledge and processing ability:

$$\begin{aligned} &\text{Goals} + \text{Task} + \text{Operators} + \text{Inputs} + \text{Knowledge} + \\ &\text{Process-limits} \Rightarrow \text{Behavior (EQ 44)} \end{aligned}$$

The design of the user interface can influence the form of Operators, Inputs, and user’s Knowledge⁵⁷. The Task, and Goals must have been found in earlier stages of the knowledge acquisition process.

Once sufficient Tasks, Goals, semantic Operations and Inputs have been elicited for the KBS-application, the syntactic and lexical implementation of Operators and Inputs provided by the user interface can be designed, with guidance from the model human information processor and its ten principles. A numerical “goodness”-estimate for the UI-design will finally become parameterized formulas that try to minimize the operation time and user remembering effort in terms of primitive motor operations and the number of chunks needed in long term memory, i.e. the amount of knowledge necessary for operating the interface. See Appendix E.

Such calculations will have much in common with GOMS-analysis⁵⁸. Examples show that the number of parameters necessary to get a useful estimate of any realistic user interface will make the calculation work difficult without computer support. A fixed user interface software architecture (UISA), such as the one that is presented in Part IV, should ease the implementation of such estimation support.

During the prototyping phases in PMS and KBS development, the interaction schemes used by the experts who maintain the domain knowledge models will be unknown. The schemes (for the user’s Behavior in P8) will very much depend on which KB-views are available in the user interface (i.e. how Inputs are provided to the user) and how commands (i.e. Operators) are made available, and it might take several prototype iterations with modified views before a satisfactory solution is found. After empirical studies of the experts a better understanding of how they interact

57. Knowledge in this context means what is needed for handling the user interface. In the design stage of a KBS we assume that there is no problem in training the user how to operate any user interface style.

58. GOMS-Analysis (Goals, Operators, Methods, Selection Rules) is an analysis-technique that has been applied on word-processing tasks (i.e. editing text instead of knowledge-bases). It showed less than 40% root mean square error in predicting the time to perform certain editing tasks for skilled typists.[Card et al. 83] p. 428-429. The GOMS model is extensively explained in chapter 5 of [Card et al. 83]. A shorter description can be found in [Kieras 88].

with the knowledge-base will emerge. This may be input to a reformulation of task descriptions and a redesigned user interface with less average operation time for performing certain tasks. Considering the effort spent on developing and maintaining product models and knowledge bases, such a redesign could be a good investment.

A fixed structure of the UISA like the one presented in Part IV enables implementation of the automatic collection of statistics from the user interface. This will show which user interface components are most frequently used. Statistics may also indicate targets for improvements and short-cuts.

Below, we propose the hypothetical measures of accessibility and operability and indicate how they could be calculated on instances of user interfaces developed with the proposed UISA.

Note that the expression of these measures in equations is a tool for a compact presentation of how things relate. The main purpose of using equations is to explain what the measures mean and to suggest a framework for evaluating user interfaces designed with our UISA. Detailed experiments and tuning with parameters must be conducted before they can give any precise information on how to improve a user interface configuration.

15.1.1 Access time and access costs

When working interactively with a large product model or knowledge-base, we want to minimize the user's access time for finding elementary constellations e that are modelled within the knowledge-base K . The access time depends on the current state s of the display which denotes the current configuration of views v_i on the screen, and the different information access paths provided by available browsers and views specified in the user interface configuration U . Access time $at(K, U, s, e)$ can be defined as the time it takes a model human information processor to locate an e-message m in a view v_i on the screen that carries the e-constellation e . In many situations there is no view v_i on the display that contains m . The user must issue some commands or browse along a path of views through the knowledge-base to find a view v_i such that $m \in v_i$ (see the example in Appendix F on page 223).

In display state 1 of Appendix F, for instance, there is only one view available which visualizes the structure of an aircraft product model. In addition to the display state s , the access time depends on what browsing path the user selects since there may be many possible paths in a knowledge-base with a complex meta-model.

The behavior of a user when "walking" a view path will depend on his/her preferences and previous knowledge. There might be short-cuts available in the interface, but if the user does not know them he, will not use them. This phenomenon is captured in the rationality principle P8. Take for instance an

operator issued by a command that creates a view v_5 that visualizes the part of the project plan where a certain geometry part is defined (see display state 4 and 5 in Appendix F). This operator could be applied both by selecting a command from a menu or by issuing a keyboard shortcut. The access time will depend on whether the user knows and remembers the keyboard short-cut and uses it, or if he prefers to select the command from the probably slower menus.

In a complex user interface with hundreds of operators it seems to simple to just calculate the optimal access times for getting a quantitative estimate of the “goodness” of the interface. A perhaps better but less easy to implement way is to assign some kind of access cost $c(e,s)$ that reflects the cost to access e given the display state s for a particular user. This cost can be a synthesis of the available possibilities of applying operators, the users preferences and knowledge about them and possibly some more factors that are individual for each user. The exact value of this cost is not important, but it is important to know how it can be influenced. To give an extreme example, the cost $c(e_{95},s_{23})$ of accessing the e-constellation e_{95} given display state s_{23} could be significantly reduced by providing a special function key that pops up a view that displays an e-message that carries e_{95} given that the user is taught exactly how to use this function key. However, the number of such specialized function keys or keyboard short-cuts will become impossible for the user to remember. There must be a balance in the interface design between best possible access times and difficulty to learn and remember. The user interface should, for instance, be consistent because this reduces the amount of specialized interface knowledge that a user has to acquire and remember. Each individual access time and access cost is not very interesting in itself but together they constitute important influencing factors on the impression and attitude that the user gets while working with a particular interface.

It seems tedious to calculate some kind of quantitative measure of this impression based on access costs. However, it seems worthwhile to formulate how parts such as individual access costs add to the whole global impression of the interface since such a formulation can guide a user interface designer in deciding where and what to improve.

15.1.2 The task and its information set

Why do we need access to particular e-constellations that are modelled in a knowledge-base? This question is usually impossible to answer for all cases at the time when the user interface is designed. However if there is a particular task that a knowledge-base user has to perform, it is possible to identify different e-constellation types that will have to be read, or modified in order to complete the task.

Many tasks are complicated and the number of e-constellations to be read and modified for their completion can be large. A good user interface

provides fast access to those e-constellations that are needed frequently and it provides some access to the ones that are less frequently needed. The interface should be balanced to find some kind of optimum. Before that is possible, some kind of measure on the access frequency of different types of e-constellations has to be found. There might be other influencing factors on why it is important to access a particular e-constellation.

Some e-constellations are used as travelling paths during the browsing of a knowledge base. The structure of a product model, for example, may be “walked” many times during the design of a product. Therefore the components of the e-constellations that define this structure may be accessed very frequently. The structural model of the car braking system in Figure 47 on page 224 can, for instance, be used as an index view to access e-constellations that hold information about individual brake tubes and disk brakes. The task of designing a car braking system may need this information very frequently. For a task T a hypothetical importance weight $iw(e,T)$ could be defined for each e-constellation that is needed for completing T . Elementary constellations modelled in the knowledge-base that are not related to T have $iw(e,T)$ set to zero.

The importance weight should reflect a relative importance between different individual e-constellations. The sum of importance weights could, for instance, be normalized to a constant N , where N is the total number of e-constellations used for completing the task.

$$(EQ\ 36) \quad \sum_{\forall e \in K} iw(e, T) = N$$

It may be tedious to estimate a figure for individual importance weights manually, but a partial ordering of their importance could be made by an expert by pair-wise comparison, or ordering of e-constellations into groups that are pair-wise compared. From the results of the comparison the individual importance weights can be computed by an algorithm.

There are other ways to define the importance weights. See the definition on page 138.

15.1.3 Accessibility

Accessibility is a hypothetical measure of how easy a user can access e-constellations e in the knowledge-base K through a particular user interface design. Since only small portions of the knowledge-base can be displayed at any one time, the user will frequently have to browse through it in order to locate e-messages that carry the needed e-constellations.

The accessibility $acc(T,K,U,s)$ can be hypothetically calculated for a particular task T on a knowledge-base instance K given the user interface configuration U which defines the different information access paths provided by available browsers and views and the current state s of the

display. The task T will determine which elementary constellations e in the knowledge-base are important for it and assign them a corresponding importance weight $iw(e, T)$. The access cost to bring an e-message carrying e to the display surface will reduce the influence of e 's importance weight on the accessibility. The total accessibility will be the sum of each e-constellation's importance weight divided by its associated access cost.

$$(EQ\ 37) \quad acc(T, K, U, s) = \sum_{\forall e \in K} \frac{iw(e, T)}{c(e, U, s)}$$

As mentioned earlier, this equation does not say anything precise but gives an idea of how changes in the interface could be made in order to increase accessibility.

15.1.4 Operability

In an incremental environment, the operability of a user interface is a **hypothetical measure** of how easy the information in the knowledge-base can be manipulated through it. Operations on the knowledge-base may be thought of as assembling, changing and deleting models consisting of e-constellations. Operability is measured with respect to a task T in the same way as accessibility, but here we put operation importance weights $opiw(\mathbf{ectr}, \mathbf{op}, T)$ on the different operations \mathbf{op} that are available on an object playing a reference in the elementary constellation type role \mathbf{ectr} ⁵⁹.

At a certain display state s , the operation cost is $opc(o, \mathbf{op}, U, s)$, where o is an object in the knowledge base, U is the user interface configuration, containing definitions of all forms and browsing views available for task T , and how they can be reached from any display state s .

(EQ 38) defines the operability for task T on a general knowledge-base K with the user interface configuration U and the current display in state s .

(EQ 38) $oper(T, K, U, s) =$

$$\sum_{\forall o \in K} \left(\sum_{\forall \mathbf{ectr} \in \mathbf{ectrs}(o, T)} \left(\sum_{\forall \mathbf{op} \in \mathbf{ops}(o, \mathbf{ectr}, U, s)} \frac{opiw(\mathbf{ectr}, \mathbf{op}, T)}{opc(o, \mathbf{op}, U, s)} \right) \right)$$

where $\mathbf{ectrs}(o, T)$ returns a set of all elementary constellation type roles for instances o of a particular class within task T . $\mathbf{ops}(o, \mathbf{ectr}, U, s)$ returns a set of all operations available on o for \mathbf{ectr} in the user interface configuration

59. A description of the domain modeling concept \mathbf{ectr} is given in section 15.7 on page 137

U at display state s . $\text{opiw}(\text{ectr}, \text{op}, T)$ returns the importance weight of op for ectr in task T . It may simply be 1 if the operation is needed for T and 0 if not. A more detailed cost model may compute opiw from usage statistics taken from logged prototype sessions.

The operability is heavily determined by the user interface configuration U . The operation cost $\text{opc}(o, \text{op}, U, s)$ is a combined function of the available browsing paths from display state s with forms and browsers available for T within U , and of the individual users knowledge about how to walk them from display state s and how to issue commands that apply operations on objects.

15.2 Object model of the task description concepts

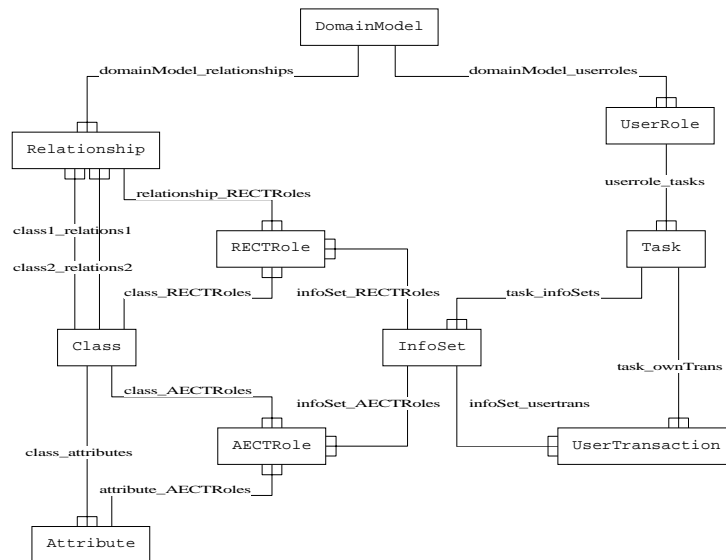


FIGURE 39. Object model diagram of the task description language.

Figure 39 shows how the primitives used for the task-description language are related. The users of a product modeling system are assigned user roles to which a set of tasks belong.

Examples of *UserRoles* from our benchmark domain model are *ExecutiveDirector*, *ChiefDesigner*, *ProjectManager* and *Designer*. Examples of *Tasks* related to the product modeling system for, for instance the *ChiefDesigner* are *ArticleStructuring*, *ArticleChecking* and *ArticleApproval*.

The ArticleStructuring task requires flexible read and modification access to the information in the product model, while the ArticleChecking task only should provide modification access to the *checkedBy* attribute value.

How the information is accessed in the different tasks is documented in one or many *InfoSets*. An information set contains descriptions of how attribute e-constellations and relational e-constellations are going to be used. These e-constellation access descriptions are instances of *AECTRole* and *RECTRole*.

Some product model modifications need to be organized into atomic transactions in order to keep the product model (or KBS) in a legal state. Such transactions are documented in a *UserTransaction* that relates to one *InfoSet* instance that describes the roles of the involved e-constellations.

15.3 UserRole

A user role describes responsibilities for a particular type of user/actor within the system.

Table 16: *UserRole* attributes.

| Name | Description |
|-------------------|---|
| <i>name</i> | A descriptive name for the user role. |
| <i>definition</i> | A definition of the user role in a few sentences. |

15.4 Task

A task within a domain model defines a standard task that requires access to a specified set of information.

Table 17: *Task* attributes.

| Name | Description |
|-------------------|---|
| <i>name</i> | A descriptive name for the task. |
| <i>definition</i> | A definition of the task in one or a few sentences. |

15.5 UserTransaction

Defines an infoset with modifiable e-constellations that are written to the

database in one user transaction.

Table 18: *UserTransaction* attributes.

| Name | Description |
|-------------------|---|
| <i>name</i> | A descriptive name for the user transaction. |
| <i>definition</i> | A definition of the user transaction in one or a few sentences. |

15.6 InfoSet

Information sets describe sets of information that play certain roles during the completion of a task or particular user transaction. The members of information sets are pairs $\langle ecrole, ectype \rangle$, where *ecrole* is a role identifier, and *ectype* is an attribute- or relational e-constellation type ($\langle O, A \rangle$ or $\langle \langle O1, O2 \rangle, R \rangle$).

Table 19: *InfoSet* attributes.

| Name | Description |
|-------------------|--|
| <i>name</i> | A descriptive name for the information set. |
| <i>definition</i> | A definition describing the purpose of the info set. |

15.7 Elementary constellation type role, ECTRole

Attribute and relationship e-constellation types were defined in section 12.4.5 on page 102. An instance **ectr** of ECTRole defines the presence of an e-constellation for a particular purpose within a task. ECTRole is a common superclass holding some shared attributes for AECTRole and RECTRole.

Table 20: *ECTRole* attributes.

| Name | Description |
|-------------------------|---|
| <i>importanceWeight</i> | Relative importance of an e-constellation in this ectyperole for the task. Used for specifying a priority on which e-messages to provide the best access for in the user interface. |
| <i>ecrole</i> | Identifies a role that an e-constellation plays within a user transaction or information set. |

The following two ECTRole attributes are discussed in more detail.

importanceWeight : Another definition for which the importanceWeight number gets a meaning is;

Definition : The importance weight is the probability that the information in the e-constellation will have an influence on the information created by the user if he/she reads (and interprets) an e-message for it during the completion of the task.

ecrole: An identifier that describes the role of an e-constellation of a certain type within a user transaction. The identifier is needed when two or more instances of the same elementary constellation type are used for different purposes. This may, for example, occur when two Article-instances from the benchmark domain model are to be connected in the hierarchical relationship owner_owns. Two **aect**'s <Article,ref_id> with the ecroles **ownerRef** and **partRef** are used to provide the references to the owner and the part. At runtime, the attribute e-constellation <anOwnerArticle, <ref_id, anOwnerArticleRefId>> provides a selectable⁶⁰ reference to *anOwnerArticle* which has the ecrole **ownerRef** and <aPartArticle, <ref_id, aPartRefId>> provides a selectable reference to *aPartArticle* and has the ecrole **partRef**. The operation completed within the user transaction creates the relational e-constellation <<anOwnerArticle, aPartArticle>, owner_owns>.

During prototype development it may be hard to identify and give a name to the ecroles for individual e-constellations. The default **ectr** is to represent an instance of the class and the default ecrole name is <Class><occurrence> or just <occurrence>, where <occurrence> is an integer counting the number of distinct occurrences of the e-constellation type within the same information set.

15.8 Attribute e-constellation type role AECTRole

An AECTRole documents access rights for a certain ecrole within the information set; see Table 21.

Table 21: *AECTRole* attributes.

| Name | Description |
|-------------------|---|
| <i>name</i> | Name of the format <Classname>.<attributename>. |
| <i>definition</i> | A comment describing the purpose of this AECTRole within the infuset. Can be omitted in most cases. |

60. See section 19.2 "The selection manager" on page 160.

Table 21: *AECTRole* attributes.

| Name | Description |
|-----------------------------|--|
| <i>accessRights</i> | Access rights to the attribute e-constellation in this e-role, for the user role that performs its task. Can be R,C,U,D for Read, Create, Update and Delete. Can be used by interface compilers when selecting interface functionality or menu choices for operations. |
| <i>identificationWeight</i> | A key weight. The probability that the user can uniquely identify an object for the purpose of the task by looking at the value of an e-constellation of this attribute e-constellation type, given no other information about the particular object. |

15.9 Relationship e-constellation type role RECTRole

A RECTRole specifies a role that a relationship may play in a certain task.

Table 22: *RECTRole* attributes.

| Name | Description |
|---------------------|---|
| <i>name</i> | Name in format<Classname>.<side><nameXtoX> where <side> specifies which end of the relationship the RECTRole is viewed from. |
| <i>definition</i> | A comment describing the purpose of the rect -role within the infoSet. Can be omitted if obvious. |
| <i>accessRights</i> | Access rights to the relation e-constellation in the rect -role for the user role that performs its task. Can be R,C,A,R,D for Read, Create, Add, Remove and Delete. Can be used by interface compilers when selecting interface functionality or menu choices for operations. |

15.10 Summary

The information-oriented task description language contains the primitives UserRole, Task, UserTransaction and InfoSet. A task may own many InfoSets, but the information affected by a particular UserTransaction is always specified in one InfoSet.

Each InfoSet holds a set of e-constellation type roles (ECTRoles) that may be of either of its subclasses AECTRole or RECTRole.

The function of the ECRole is to uniquely identify e-constellations of the same e-constellation type that are used for different purposes within the same InfoSet.

16 The Benchmark Task Descriptions

The benchmark domain model and task descriptions are selected to be a representative example that can be extended with details to cover a fully operational set of functional features necessary for efficient implementation of PMS-development platforms.

This chapter is an informal introduction to the task descriptions for the benchmark domain model. Appendix B page 190 shows a more detailed example of a task description.

A benchmark product model database is managed by four user roles. ExecutiveDirector, ChiefDesigner, ProjectManager and Designer. Figure 40 on page 143 shows the responsibility areas for different user roles within the domain model. The information in these areas is created, updated and deleted by the user assigned to that user role.

16.1 ExecutiveDirector tasks

16.1.1 DepartmentManagement

Involves the creation, deletion and update of attribute values in instances of Company and Department.

16.1.2 Product Inspection

Involves browsing the information from a Company instance down to Product and ProductData.

16.2 ChiefDesigner tasks

In addition to the tasks presented below, a chief designer can do all tasks that a ProjectManager and Designer can do.

16.2.1 Product Management

Involves the creation, deletion and update of attribute values in instances of Product, ProductData, C and its subclasses.

16.2.2 Article Checking

Involves browsing of article structures, marking them as checked by entering the login-id in the checked_by field.

16.2.3 Article Approval

Involves browsing of article structures, and marking them as approved, by entering the login-id in the approved_by field.

16.3 Project Manager tasks

The project manager is responsible for creating a hierarchical design activity plan for a certain product.

16.3.1 ProjectMonitoring

Involves browsing of article and activity structures.

16.3.2 ActivityPlanning

Involves creation and deletion of DesignActivity objects, and building substructures having their corresponding deliverables.

16.4 Designer tasks

A designer makes the detailed design decisions for articles and their structures.

16.4.1 Article Structuring

Involves interactive creation of articles and article structures.

16.4.2 Article Design

Involves interactive assignment of attribute values for articles, and IncludedArticles.

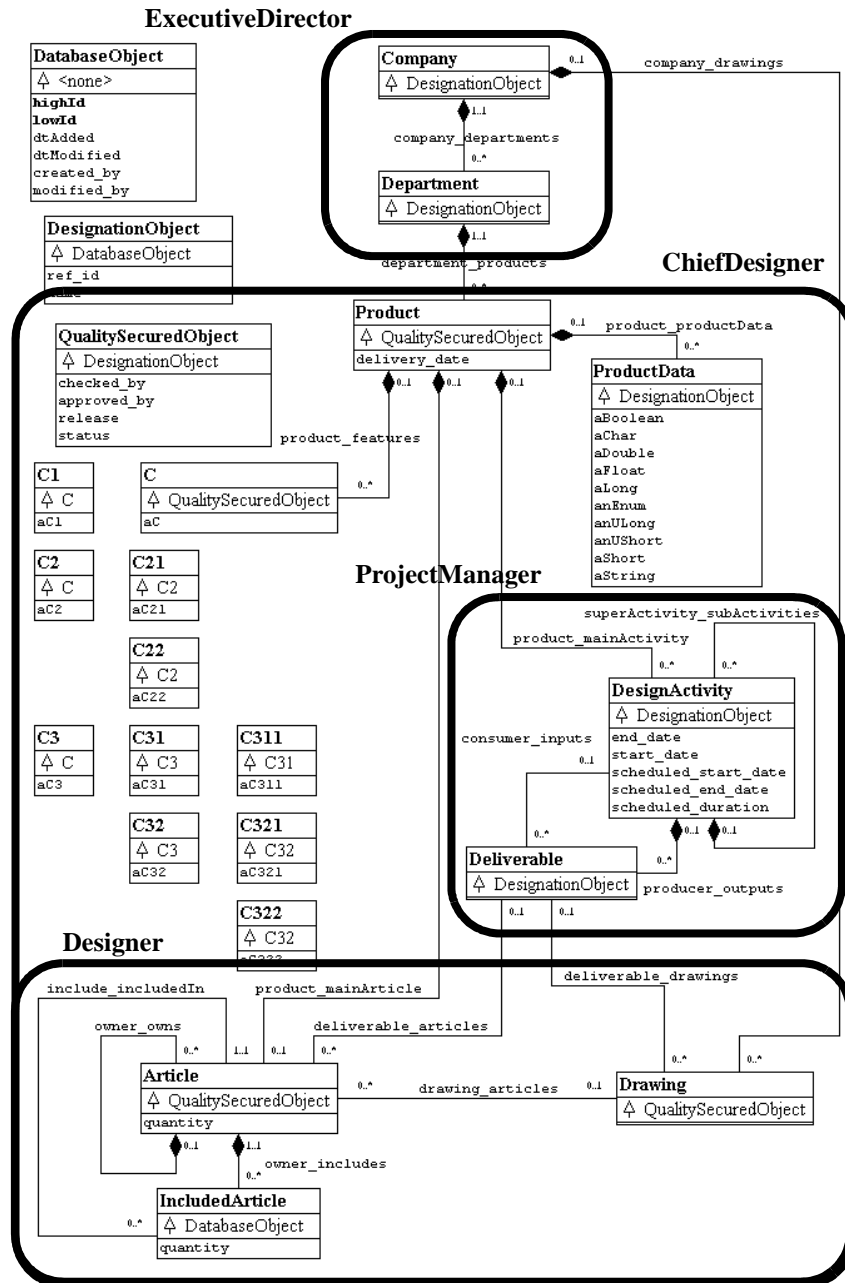


FIGURE 40. Responsibility areas for the different user roles in the benchmark domain model.

16.5 Specifying a functionality feature checklist

To be able to verify and validate source code generators, a complete specification of the functionality that these must generate must be created and published.

Our development environment contains source code generators for the product model database, the CAD-PMDB interface, and for graphical user interface browsers.

For the PMDB and CAD-PMDB interface, a functionality feature check list should contain specifications of the naming rules, the functionality, exceptions and returned error and status messages for operations with the semantic functionality listed in Table 8 to Table 15 in chapter 14.

A functionality feature checklist for user interface source code generators should contain the same type of specifications for operations with the semantic functionality listed in Table 30-Table 45 in Appendix C. There each functionality has at least one instance of its implementation in the benchmark domain model. To make a feature list easier to comprehend, each functionality should be exemplified by one instance from the benchmark domain model.

A source code generator company, implementing source code generators according to such a published feature list, can check that their source code generation system covers all functionalities by thorough testing of the exemplified instances from the benchmark domain model.

Part IV Theoretical
Framework for
Object-Oriented
User Interface
Configurations

17 Introduction to Part IV

When a domain model has been specified, it must be tested with realistic, instantiated test cases. If a domain expert is to be able to develop such test cases without extensive support from a software- or knowledge engineer, he/she must be provided with a user interface that is easy to operate and provides full access to all types of object-, attribute-, and relationship operations described in chapter 14.

A straightforward form based object-oriented browser application for the benchmark domain model, with one form for each leaf class that provides access to all possible operations on objects of that class, requires 18 forms, containing 292 attribute editors and 32 relationship editors with a total of $(18*17+292*9+32*13) = 3350$ customized event procedures in a traditional UI-toolkit development environment (see Table 35, 37, 40). For product modeling systems with complex domain models, it is obvious that such user interfaces cannot be implemented manually on that abstraction level if the prototype development cycle is to remain incremental.

Chapters 18 and 19 together with Appendix C describe the user interface software architecture (UISA). This UISA is a framework which enables automatic generation of most of the domain model-dependent source code for handling for instance a straightforward form-based user interface. In the ProCAD system it is used for implementing the CAD/UI interface.

The UISA has also been tested with a Smalltalk implementation of an instance browser for the benchmark- and ProCAD domain models. Most of the domain model-dependent source code is generated automatically. The graphical layout of the different user interface objects on the forms still has to be adjusted manually to get a comfortable prototype to work with.

Measurements from source code generators and generated source code that implements the above described functionality for the benchmark and ProCAD domain models are reported in section 10.4 on page 88. These show that the source code generators are a one-time programming effort that consist of less than a 1/10 of the source code than is actually generated from a domain model of the size of ProCAD for each prototype iteration.

Chapter 20 discusses what is needed to provide better prediction capabilities directly from domain models and models of user interface configurations. The relationship to some other object-oriented domain modeling languages is described, and other UISA-architectures and Meta-CASE-tools are discussed. The Part ends with some questions for future work and conclusions.

18 The User Interface Software Architecture

18.1 Introduction

Figure 41 shows a system model of the user - knowledge-base interaction. The knowledge-base (KB) contains an object-oriented model of the real system and the knowledge needed for solving the associated problem. The user interface (UI) visualizes the information and knowledge in this model on the screen in various representations.

The user looks at the representations and gets help to recall his own mental models of the different objects from long term memory into working memory. Using graphical keys for the recall of relevant knowledge supports and speeds up the user's reasoning processes.

The user enters text and commands via the keyboard and issues commands and changes the focus of operation with the mouse. Key strokes and mouse events are transformed to messages by the window system which sends them to the particular user interface object in the current focus⁶¹ of the object-oriented user interface state model. The user interface objects in this model pass control to relevant routines in the UI-KB interface which in turn carry out the semantic operations on the knowledge-base.

The above scenario can be viewed as an information-processing system including the three interacting components; knowledge-base, user interface and user. The basic entities to transfer between KB, UI and user are elementary messages⁶² that carry elementary entities of information (e-constellations). The purpose of a good UI is to enable a high possible average bandwidth for elementary message transfer.

Each component in the information processing system of Figure 41 can be described with a theoretical model. Having such models, we can estimate the time needed for performing a certain operation on the KB e.g. transferring information from the user to the KB or for the user to access an entity of information in the KB given a certain user interface design.

61. The term "current focus" is often used in the context of windowing systems for referring to the currently selected object or object pointed to by the cursor or mouse pointer.

62. The concept of elementary messages for analysis of information contents is a part of the infological theory that was developed by Sundgren [Sundgren 73]. It represents minimal units of information, where information is data that gets a semantic interpretation in the receiver of the message. See chapter 12 on page 97 to recall the infological theory.

A typical performance measure could be an estimate of the time it takes for the user to find an information entity in the KB by following an interaction path. The path includes the average times to move the mouse to certain mouse sensitive areas, to bring up menus, to select the right choice, and for the system to visualize the information views on the screen.

See Appendix F for an example of such user interaction. This type of calculation might become very important, since engineers and experts in the future will spend man-years on KB interaction via a particular UI.

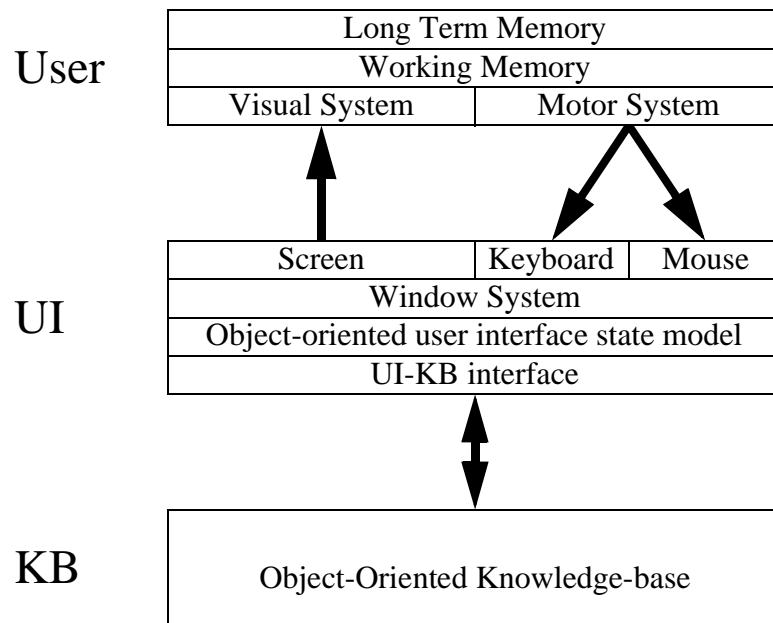


FIGURE 41. A systems and subsystems model of the user-user interface - knowledge-base interaction.

The theoretical model of the KB which was presented in chapter 14 is a set of concepts that can be used to describe object-oriented, and to some extent frame-based software systems. It is compatible with suggested implementation technology for object-oriented databases [Kim 90][Cattell et al. 96] and object-oriented languages [Wegner 87]. There are of course other models for object-orientation such as prototypes and delegation [Ungar et al. 87] that could be used, but the majority of research efforts and promising commercial systems adhere to the class-instance distinction that is explicit in this model.

The model of the user is adopted from Card, Moran and Newell [Card et al. 83] (Appendix D on page 215). A description of the nature of human memory is given in Appendix E page 218.

The abstract model of the UI is described in section 18.2. A strong

guideline for the design of the UISA has been to have a direct mapping between concepts in the user interface model and concepts in the knowledge base which have their formal definition in the domain model.

The design has also been generalized and synthesized from theoretical studies and practical experience with various window systems⁶³.

The notational convention for the following sections are as follows. Static concepts that are not changed in the delivery system are denoted by capital letters. Dynamic concepts that are created and may change in the delivery system have lower-case letters. A variable holding an instance of a concept is denoted by *italics*. A unique particular instance of a concept is subscripted.

The following notation is used for concepts in the domain model of the object-oriented knowledge-base. Knowledge-base object *kbo*, Object identifier *oid*, Class *C*, Attribute *A*, attribute e-constellation *aec*, Relationship *R*, Relational e-constellation *rec*. Detailed descriptions of these concepts were given in chapter 12 and chapter 14.

18.2 Components of the UISA

In this chapter, the following components of the UISA are introduced: Window instance *w*, Window type *WT*, window implementation class *W*, user interface object *uio*, object editor instance *oe*, object editor type *OET*, object editor implementation class *OE*, field instance *f*, field class *F*, attribute editor instance *ae*, attribute editor implementation class *AE*, relationship1 editor instance *r1*, relationship1 editor implementation class *R1*, relationshipN editor instance *rn*, relationshipN editor implementation class *RN*, link editor instance *le*, link editor implementation class *LE*, view instance *v*, view implementation class *V*.

Some statistical parameters and operations on the above listed components are described in Appendix C.

Chapter 19 describes some other important components in the UISA, namely the display manager *DM*, selection manager *SM*, clipboard manager *CM* and transaction manager *TM*.

In Figure 42 an object model diagram of the components described in this chapter is shown. Instance configurations of this model implement the object-oriented user interface state model layer in the UI shown in Figure 41.

63. The following have been studied: KEE Pictures [Intellicorp 87], Microsoft Windows Version 3 [Microsoft 90], X-Window system and Xt [Young 89]. To some extent: ET++ [Weinand et al. 88], The Model-View-Controller [Krasner&Pope 88].

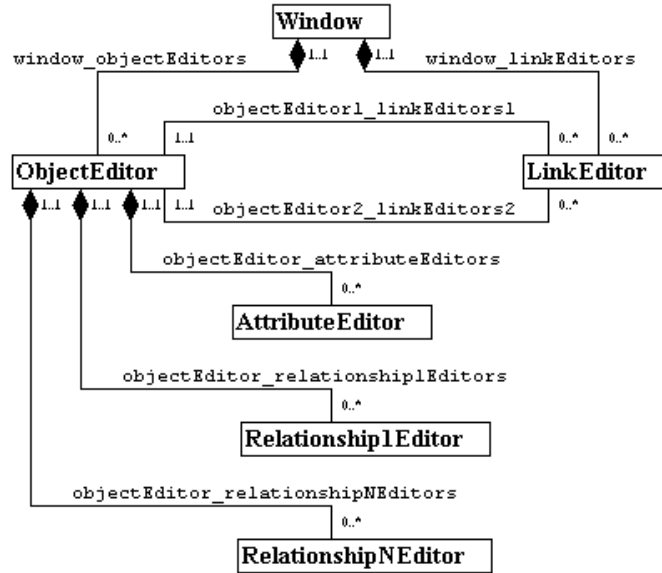


FIGURE 42. Object model diagram showing relationships between different user interface objects *uio* in the domain model of the user interface state.

Each window instance *w* has a reference to a knowledge-base object that serves as a root for the information visualized in the window. A window may own a set of object editors *oe* that provide access to other *kbo*'s related to the root *kbo*.

An object editor can own a set of attribute editors *ae*, a set of relationship1 editors *r1* and a set of relationshipN editors *rn*. An attribute editor provides visualisation and editor functionality for attribute e-constellations that belong to the *kbo* held by the object editor *oe*. The relationship editors provide similar functionality, but for relational e-constellations in which the *kbo* participates. A common name for editors owned by an object editor is field or field editors *f*.

In windows providing visualisation of graph and network structures in the knowledge base, a link editor *le* is used for editing *kbo*'s that represent a link between two other *kbo*'s. When providing 2D visualisations of instances of the benchmark domain model, instances of *IncludedArticle* which implements an M-N relationship between *Articles* can be represented by link editors. The same applies to *Deliverable* where the M-N relationship is between activities.

18.2.1 User interface object (*uio*)

All classes in Figure 42 reference a *kbo* and are called user interface objects *uio*. An *uio* keeps knowledge about how to map the e-constellation representation of a *kbo* in the knowledge base to an e-message representation on the screen. The state of a *uio* contains, for instance, window-coordinates and height and width of the user interface representation. *Uio*'s are generated dynamically. They only exist as long as they are accessible from the workstation display⁶⁴. Most *uio*'s have functionality for providing the user with access to related *kbo*'s to the *kbo* they represent. This functionality is implemented in cooperation with the DisplayManager DM. Other tasks of a *uio* include handling selections and logging of operations issued by the user in cooperation with the SelectionManager SM and TransactionManager TM.

18.2.2 Object Editor (*oe*)

An object editor owns attribute- and relationship editors that may represent a subset of the attributes and relationships that are defined for a particular object class. An object editor can be single within an window, see for instance Figure 33 on page 110. This Company object editor owns eight attribute editors and two relationshipN editors.

The *kbo* of object editors within the same window may be set interactively from relationship editors when the e-message of a **rec** is selected. Thus a chain of object editors linked by relationship editors may reflect a browsing path from an object of one class via a relationship to objects of another⁶⁵ class, and so on, see Figure 34 on page 110. This browser window owns 4 object editors, where the first owns two attribute editors and one relationshipN editor.

18.2.3 Object Editor Type (OET)

An object editor owns attribute- and relationship editors that may represent a subset of the attributes and relationships that are defined for a particular object class. An object editor type (OET) is a predefined configuration of an object editor, that may be instantiated many times in the same window. Figure 35 on page 111 shows a hierarchical article browser that has four instances of the same Article-OET.

64. If it is a requirement to be able to save the current screen layout, *uios* can of course be made persistent. This feature is valuable since it often takes time to set up the views needed for performing a certain task. This functionality can also be accomplished by saving some of the *uio* state, such as coordinates and regenerating the *uios* from them.

65. In the case of hierarchical relationships with both ends in the same class, chained object editors may represent objects at different levels in the hierarchy.

18.2.4 Object Editor implementation Class (OE)

In the case of representing *kbos* in 2D views the object editor implementation class OE must provide a fast creation and initialization of instances. A 2D view of an article structure, see for instance Figure 36 on page 111, may contain hundreds of object editors that ought to be displayed in fractions of a second. Object editor implementation classes are arranged in a class-hierarchy that allows sharing of the implementation.

18.2.5 Field editor (*fe*)

A field editor *fe* visualizes one attribute e-constellation $\langle o, \langle p, v \rangle \rangle$, or one or several relational e-constellations $\langle \langle o_1, o_2 \rangle, r \rangle$. The encoding of the e-message within the field, must enable the user to recall its e-concept. We recall from section 12.5.3 on page 104 that if $\rho(O)$ is an object group reference where the object group is a class *C*, and $\rho(A)$ is an attribute reference, then the pair $\langle \rho(O), \rho(A) \rangle$ is called an attribute e-message type (**aemt**) or attribute e-concept. In the implementation of an **aemt** in a form-based browser (See Figure 43), $\rho(O)$ is derived from the title of the window, or the shape or position of the object editor. $\rho(A)$ is usually the label or prompt in front of the edit field for the value, but it could also be the location of the field, if the user gets special training to remember this.

In the case the field is a relationship1- or relationshipN editor that represents relational e-constellations, we derive from section 12.5.3 on page 104 that if $\langle \rho(O_1), \rho(O_2) \rangle$, is an 2-tuple of object groups, and $\rho(R)$ is a reference to an binary object relation, then the pair $\langle \langle \rho(O_1), \rho(O_2) \rangle, \rho(R) \rangle$ is called a relational e-message type (**remt**). In the implementation of a **remt** in a form-based browser, $\rho(O_1)$ is derived in the same way as $\rho(O)$ for an **aemt**. $\rho(R)$ is derived from the label or prompt in front of the relationship editor which displays *R.name1to2*, or *R.name2to1*, depending on if O_1 represents *R.class1* or *R.class2*. $\rho(O_2)$ is implicit in the users knowledge of the domain model. Objects on the other side of relationship *R* when viewed from class C_1 can only be of class C_2 (or any of C_2 's subclasses).

An e-message type can be codified by visual features such as colour, shape, texture or relative location. This is particularly evident in the symbol libraries of functional CAD-diagrams (Figure 11 on page 57). If the user has knowledge of this graphical syntax, there is no need to textually spell out the name of the attribute. Smart graphical encodings can save considerable amounts of screen space and improve readability⁶⁶. When a user task requires fast access to any entity of information in a large

66. Graphical encodings used in geographical maps are good examples of efficient utilization of limited space. They use colour coding for entity types such as red for roads, and blue for water. Red blobs of different shapes mark the existence of cities and represent intervals for the value of the attribute "POPULATION". Different sorts of coloured bar-charts and diagrams are other good examples.

quantity, a compact representation will save many user interaction operations. The user might work with a particular kind of information representation for several hundreds of hours and product models and future knowledge-bases will contain large numbers of objects. Therefore some initial effort on learning an efficient graphical encoding will yield benefits in the long run.

FIGURE 43. Information entities in a form-based object editor.

$\underline{aemt} = \langle p(\text{Company}), p(\text{Company.name}) \rangle$

$\underline{remt} = \langle \langle p(\text{Company}), p(\text{Drawing}) \rangle, p(\text{company_drawings}) \rangle$

18.2.6 Field editor implementation class (FE)

The field editor implementation class FE provides the mechanism for displaying and modifying the attribute values and relational e-constellations reflected in field-instances *fe*. FE's are organized in class hierarchies in the same way as OE's and other user interface object classes. Window system programming toolkits (e.g. Xt [Young 89][Rao 87] ET++ [Weinand et al. 88]) usually have a library of useful components for field editor classes. These libraries implement the basic functionality for push-buttons, sliders, edit-fields, list-boxes and so on. There are, of course, a large number of commercial frameworks where some are included in modern window-based operating systems that provide much of the functionality presented here.

In addition to implementing graphical layout and user interaction, our FEs, also provide an interface to the UI-KB layer shown in Figure 41, which stores and retrieves attribute values from *kbos* and translates user commands to operations on its relationships. This UISA functionality is provided in higher abstract implementation classes. The knowledge engineer selects an FE that suits the need for graphical layout and interaction functionality. The UI-KB interface that adapts the FE-functionality to work on objects from a particular domain model can be generated from the domain model in the meta-database.

18.3 2D user interface support

In a knowledge base for engineering, we typically want to represent complex hierarchical and network structures by 2-dimensional graphs. In this case, we need to introduce two other types of user interface objects, namely link editors and views.

18.3.1 Link Editor (*le*)

An link editor *le* represents a **rec** $\langle\langle o_1, o_2 \rangle, R\rangle$ by a graphical link between two object-editors. This **rec** is typically implemented so that the two *kbo*'s $\langle o_1, o_2 \rangle$ that it connects have each other's object identifiers stored in one of their instance variables or in a local set- or collection-oriented abstract data type that is referenced through the operations or specializations of the operations presented in section 14.4 on page 121. A link must in most cases be explicitly expressed in the user interface in the form of a *le*. This allows the user to interact with the link even if it is not represented as a *kbo* in the knowledge-base. The *le* can be manipulated and transforms the user's commands to the corresponding relationship operations on the underlying *kbo*'s.

Another task for link editors is to represent an information-rich relationship between two classes that have been transformed into a class in the domain model. (Recall the discussion in section 14.3 on page 119 and section 14.6 on page 126). The information represented by the link-editor is then stored in its own *kbo*.

18.3.2 Link Editor implementation class (LE)

How a **rec** is visualized and how the operations can be expressed using, for example, menus or drag-drop, is determined in the link editor implementation class LE. LE specifies common properties and functionality of all its link-instances *le*. Each unique link editor *le_i* within a view *v* must have a unique priority that allows a topological sort among the links. This priority is used by a 2D view *v* to manage the spatial rendering order. It may be a computationally derivable value or a value stored in one of the objects $\langle o_1, o_2 \rangle$ of the **rec**.

18.3.3 View (*v*)

A view is a special type of window in which object editors *oe* and sometimes link editors *le* are displayed. A view-instance *v* is always generated from one special root *kbo*. A view in some way visualizes the environment of the root *kbo*, with respect to some relationship or relationships. The window in Figure 36 on page 111 is an example of a 2D view that displays the hierarchical relationship *owner_owns*. The layout starts from the root *kbo* with *ref_id* "a" and *name* = "Article".

In a view, the spatial dimensions (x,y) should be assigned a semantics. In Figure 36 the x-dimension expresses that an Article *kbo* represented by an *oe* to the left is an *owner* of an Article to the right. The y-dimension is sorted with *owner* membership as the first index, and the alphabetical value of *ref_id* as the second.

An example of a very simple view is the sorted table shown in Figure 51 on page 227. In the figure, each object editor *oe* contains two attribute editors *ae* which identify their **aemt** by their spatial location in the x-dimension. The y-dimension expresses an alphabetical sorting order by a symbolic name.

The explicit assignment of semantics and sorting orders to the different dimensions in the view help the user to more rapidly locate an *oe* of interest. It also preserves the graphical structure of the view or its “picture”. Pictures are easy to remember and if their structure does not change too radically over time, the user can use them as mental memory structures to remember facts or as support for recalling the access-path to certain information during interactive browsing. See for instance [Gärdenfors 89a][Gärdenfors 89b] for a deeper discussion on conceptual spaces.

18.3.4 View implementation class (V)

A view implementation class V provides an algorithm for displaying some relationship among *kbo*s. Some two-dimensional views may require more complicated graph-drawing algorithms. A common and useful 2D-view for visualizing important properties of a plan is the PERT diagram (Program Evaluation and Review Technique [Schaffer et al. 65]), see Figure 52 on page 228. This view expresses the ordering relation and time requirements for different activities. Algorithms for drawing such graphs automatically are known [Battista et al. 89]. The implementation of such algorithms are typically stored as methods in abstract view-classes higher up in the view implementation class hierarchy. A PERT diagram can be used to display a level of the hierarchical network design activity graph in the benchmark domain model.

There is a rich source of literature available on graph-drawing algorithms [Eades&Tamassia 89]. However, not all drawing techniques are well-suited for the human visual perception system [Hubel 88]. Care should be taken when selecting one since some algorithms do not preserve any spatial sorting order. A minor change among the relations in the knowledge-base may cause a drastic change in the graphical layout. This will disturb the user’s possibilities to acquire a mental model of the KB.

As mentioned, there are many benefits when organizing view implementation classes into a class-hierarchy that inherits methods and functionality. The basic drawing algorithm for a certain type of view can be implemented higher up in the hierarchy and specializations, such as

choosing OE's and LE's, are added in the subclasses. The knowledge engineer only has to code the very small parts that are unique for his particular application view.

19 The User Interface Managers

The user interface display manager (DM), coordinates the refreshing of windows. This is necessary when the e-constellations represented in the product model are changed and need to be represented with new e-messages which transfer the new state of the model in the KB correctly to the user. Section 19.1 describes the workings of the display manager.

The selection manager (SM), is a global resource from which user interface objects can request data and status information about the current selection. A well-defined protocol for selection management is essential to make this type of interface operational. The system operates according to the object-action paradigm. Legal operations and operations enabled on menus may depend on what is currently selected. Section 19.2 explains the workings of the selection manager together with formulas and description of tests how to calibrate statistics for selection times.

The clipboard manager (CM) supports copy and paste of complex object structures.

The user interface transaction manager (TM) is responsible for logging user interactions and triggering refreshes coordinated by the display manager. In advanced user interface implementations, it is also responsible for the undo-facility. The transaction manager is described in Section 19.4.

19.1 The display manager

The display manager has some additional duties compared to traditional display managers. It must preserve a correct view of the KB on the screen. One situation might be where there are n windows on the screen that display the same *kbo*. If the user modifies a *kbo* represented in the window w_1 , the other windows w_2 to w_n have to be refreshed or at least marked as invalid if an automatic refresh would induce too much overhead. The DM keeps track of which object editors represent which *kbo*s, so a targeted refresh can be performed. This is done by maintaining the *kbo-oe* relationship in a fast global data structure so that DM can access it by indexing on *kbo* and retrieve all object editors that need to be refreshed.

Some other responsibilities for the display manager are:

- 1) Window editor resource management, finding the fastest way to provide an editor for a *kbo* for which editing is requested. This can be done by maintaining a cache of hidden editors or reusing existing open ones.

- 2) Save and load of display states or working contexts, including window placement for performing a certain task.
- 3) Calculation of visible e-messages from a snapshot of a window placement configuration, and “accessibility” measures given a particular task.
- 4) Coordination for the selection marking of selected *kbo*’s that are represented in several windows.

19.2 The selection manager

The purpose of the selection manager is to enable the user to select and refer to constituents of an elementary constellation before issuing the operation that either assembles, modifies or disassembles it.

Elementary constellations are either attribute e-constellations $\langle o, \langle A, v \rangle \rangle$ or binary relational e-constellations $\langle \langle o_1, o_2 \rangle, R \rangle$. The operations are performed by selecting the appropriate objects *o*, and then issuing an operation to a user interface object that represents an attribute- or relationship e-constellation type.

The service of the selection manager is also useful for any command that needs a set of parameters in the form of references to objects, see Table 39.

The selection manager (SM) cooperates with the display manager (DM) to maintain a visual feedback of the current selection. When we have multiple windows displaying the same *kbo*, the selection should be graphically indicated in all the views. When the selection is cleared, the graphical indications have to be removed. The mechanism for this is important for the ease of performing operations on *kbo*’s and its implementation is rather complicated. The UISA allows this function to be inherited and thereby relieves the knowledge engineer from the burden of implementing it.

Table 23: Basic operations on *uios* that involve the selection manager.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| setSelectionOwner oid - - | Form 2D | - | Clear any previous selections, and make the <i>uio</i> with its selected object the single owner of the selection. |
| addSelected oid - - | Form 2D | - | Add an object to the selection held by the selection manager. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| removeSelected oid - - | Form 2D | - | Remove an object from the selection held by the selection manager. |
| clearSelections oid - - | Form 2D | - | Clear all selections held by the selection manager. |

Table 24: Some useful *Selection Manager* methods.

| Name | Description |
|-------------------------------------|--|
| <i>SM.getSelectionCollection()</i> | Return the kbo's held by the current selection in an ordered collection. |
| <i>SM.getSelectionSize()</i> | Return the size of the ordered collection of kbo's that would be returned by <i>getSelectionCollection()</i> . |
| <i>SM.matchSelection(classColl)</i> | Match the selection against a the ordered collection of classes in classColl. Return ordered collection of kbo's if the match was successful, else return nil. |

19.3 The clipboard manager

The purpose of the clipboard manager is to efficiently support copy and paste operations of large and complex object structures. Some kbo's may own very large substructures and the clipboard manager must generate new object identifiers for all objects in the substructure for each copy that is pasted back into the KB. This requires knowledge of how the domain model is structured and knowledge of which relationships should be broken from a copy, which ones to maintain, and which ones to deep copy.

19.4 The transaction manager

The transaction manager is a global application resource that keeps track of the begin, commit and rollback of operations issued through user interface objects. It also manages the logging of information from these operations for later statistical analysis.

The reason for dividing interaction with a product model into distinct user transactions is to avoid leaving the product model in an inconsistent state. Some operations such as storing certification information for a substructure should not be completed if not all information about the certification is available. The completion of such a user transaction may move the substructure into another state where, for instance, update operations are inhibited.

Appendix C page 191 describes a set of operations on user interface objects in detail. It also describes data to log from user sessions which can be used for analysis.

To conclude, experience with practical implementation of the user interface software architecture has shown it valuable to factor out common functionality from the user interface components and place it into a display-, selection-, clipboard- and transaction manager.

This approach increases the functionality that can be provided directly in an automatically generated user interface and decreases the size of the source code generator scripts.

20 Discussion

Efficient development environments for product modeling systems can benefit from using knowledge developed in research programmes of KBS for design, CASE, databases, object-oriented programming, user interfaces, and to some extent cognitive psychology.

The value of this work is the integration of knowledge from these different domains to a coherent whole that solves a practical industrial problem.

20.1 Prediction properties of the domain modeling language

To our knowledge, there is no theory published yet that enables prediction of the performance of a generated browser user interface application from a domain model, measured in average user interaction- and system response times for various operations on instantiated object structures from an object-oriented domain model. The reason is probably the complexity of such a theory. To make such predictions for form-based and 2D user interface configurations, prediction functions for average completion times for more than 100 different types of operations (listed in Appendix C) must be included. These prediction functions need statistics from the population of the database to be able to compute realistic estimates. The average time to select an object in a 2D view showing an article structure (Figure 36 on page 111) will, for instance, be different if only a small fraction of the average number of objects is visible, so that scrolling must be applied. Theory for calculating user interaction times for a well-defined sequence of actions is available in the model human processor (Appendix D page 215).

Theory for calculating statistics of the population of a database implementation of an ER-model is given in the paper “A Temporal Statistical Model for Entity-Relationship Schemas” presented by J-L Hainaut, at ER11 [Hainaut 92].

It presents a statistical treatment of what can be derived from an ER-model given some basic statistics on the populations of its concepts Entity type, Group, Attribute, Value Domain, Role, Relationship Type, and Spaces.

The paper provides 40 equations that describe the relations between populations of instances of the above-mentioned concepts for the ER-model. (EQ 17) - (EQ 21) in chapter 14 describe similar relations for the conceptual framework in this thesis. It should be possible to extend the

proposed framework to a similar level of detail based on the work done by Hainaut.

The strength of our model compared to the ER-model is the ability to calculate an exact information quantity for an instantiated model, since the domain model itself implicitly specifies to which object a relational e-constellation belongs. To perform a similar calculation for the ER-model, additional design decisions must be made.

Another advantage of our model is that some implementation-oriented design decisions are already implicit in the modeling language, which simplifies the implementation of source code generators.

20.2 Other object-oriented domain modeling languages

On the design level, the concepts of Object-Oriented modeling of static instance structures is fairly well understood today. There are many commercial tools that provide support for object-modeling and design. Many of these are based on the methods that were made popular through [Coad&Yourdon 90], [Rumbaugh 91] and [Jacobsson et al. 92].

Another approach which provides some different abstraction techniques on the design level, oriented towards the behavior of object systems, is the OORAM-method [Reenskaug 96]. It uses collaboration views for roles that objects take, that declare message paths between the roles. A message path enters a role through a port that specifies the message protocol that the role has with the connected role in the specific collaboration view. What represents the concept of a class in the other models is a synthesis of attributes and message protocols from the union of all roles in all collaboration views that a certain type of object may play. This approach seems to be more orthogonal with respect to reuse of collaboration patterns.

EXPRESS has an object-based flavour and is a fundamental part of the STEP standard ISO 10303, "Product Data Representation and Exchange" [STEP 92]. EXPRESS models have a similar expressive power for static object modeling as our domain modeling language [STEP 92a]. It is straightforward to generate EXPRESS code from our meta-database that contains unique functions for each entity type for automatic information quantity calculation.

Interface definitions for our domain models can be generated in the interface definition language (IDL) for CORBA. In addition, the logic for the basic object manipulation functionality can be generated automatically.

The ODMG-93 standard [Cattell et al. 96] specifies the object definition language (ODL) which has a simple mapping to the concepts defined in chapter 14. One difference is that in ODL the abstract data type implementation for the relationship must be specified. In our domain modeling language the selection of ADT-implementation is left to the

source code generators, but it is easy to augment the relationship objects in the meta-database with an attribute that gives a hint to the source code generator what implementation to select. It should be straightforward to write source code generators that generate information quantity calculation functions for object-oriented databases that implement the ODMG-93 standard.

AMOS (Active Mediator Object System) is the multi-database research platform at the Engineering Database and Systems Laboratory [FahlRischSköld 93]. The AMOS data model is functional and object-oriented. It is based on the IRIS data model [Fishman et al. 89], which in turn is based on the functional data model of DAPLEX [Shipman 81].

There are three basic constructs in the AMOS data model: objects, types and functions. Functions can be either stored, derived or external. Our domain modeling primitives have mappings to AMOS types, stored and derived functions. Classes are mapped to types in a type hierarchy which inherit functions that implement class attributes (Table 7 on page 114) and class operations (Table 8). Attributes and relationships are implemented using functions. In AMOS, functions are first-class objects which can have stored and derived functions themselves. Thus attribute attributes (Table 10 on page 117), attribute operations (Table 12) and relationship attributes (Table 10) are mapped to functions of the built in type *function*. The corresponding AMOS type for each class in a domain model is provided with a set of functions that implement the class instance operations (Table 9). Each attribute in a domain model is mapped to a set of functions on its corresponding type which provides the attribute operations (Table 12) on objects of that type. Relationships are mapped to a set of stored and derived functions that implement the basic relationship operations (Tables 14,15 on pages 121,124).

From what we know about the emerging SQL3-standard, domain models implemented according to our meta-database design will be straightforward to implement.

20.3 Other work on user interfaces

To my knowledge little theoretical work has been done on user interfaces that are tightly integrated with the application domain model and perform quantifiable predictions on user interface performance.

Common interface support systems and user interface development toolkits (e.g. [Schaufler 88][Rao 87]) are too isolated from the application to the support rapid developments of KBS-applications. They provide little or no support for representing complex structures in the form of graphs. They have all kinds of functionality for developing an attractive graphical layout, but what is really needed for KBS development is support for an efficient information transfer in the final interface.

At the UISA-level, there are numerous approaches for visualizing the internal workings of an object-oriented software system. One of the well-known approaches is the Smalltalk Model-View-Controller (MVC), class hierarchy framework [Krasner&Pope 88]. In short, it divides the responsibility for a user interface into three abstract superclasses (Model, View and Controller) which then can be specialized for the current application:

The *model* represents the data structure of the application. In our perspective we could compare it with a static instance structure in the KB.

The *view* handles all graphical tasks. It requests data from the model and displays it. A view can contain a hierarchy of subviews. The view handles clipping of graphical objects at window edges and transformations of the subviews. The views *v* in our architecture are mainly intended to display object editors *oe* that reflect KB-instances and their instantiated relationships.

The *controller* takes care of the user input events and transforms them into messages to both the *view* and the *model*.

In our architecture there are no controllers. The windows *w* or views *v* take care of the conversion of user interaction events from the underlying window system and convert them to messages *M*. The coordinates of mouse events and the current dialog focus determines which UI-object will receive the message.

There are many commercial application frameworks that implement much of the functionality of an UISA. To be able to compare them one has to study the commercial products in detail.

The main difference between our approach and others we have studied, is that ours makes specific assumptions about the domain model in the sense that the UI implementation is generated from a domain model. This allows the UI to work on any instance-configuration that adheres to the domain model.

The view instances *v* in our UISA are automatically generated from the instance structure in the KB or PMDB, using a layout algorithm. Other object-oriented approaches often assume a more static instance structure of the application and the graphical layout of the windows has to be done by hand or possibly with the help of a UIMS.

The purpose of the UI is to provide an isomorphic picture of the KB. When the KB changes, its appearance in the UI will change accordingly. This can only be obtained if the UI is generated from the KB at both schema level (meta-model level) and instantiation level (a particular model).

The user or knowledge engineer is relieved from time-consuming non-productive layout tasks. Considering the large amount of redundant graphical information that otherwise would have to be maintained and the potential size of future knowledge-bases, this is an important property.

Many approaches to object-oriented user interface development systems, seem to have technology push as a major source of inspiration. As new features are possible to implement they become used. Our approach, besides being influenced by object-oriented programming systems, object-oriented databases demands and window systems is largely inspired by cognitive psychology [Card et al. 83], and what is known about the human visual system [Hubel 88]. It should be possible to utilize the large parallel information processing and memory capabilities of human vision in a better way if the user interfaces are adapted according to what is known about it. The idea is that as much as possible of the graphical contents in views of the KB should carry semantically meaningful information of interest to the user, i.e. the graphical layout should reflect some semantic information that is valuable to recall. Our hypothesis is that the possibility to recall facts in the form of pictures will significantly increase the knowledge engineers' and users' abilities to use and reuse the contents of a large toolbox or knowledge-base. However, this requires that the graphical syntax and its corresponding user interaction facilities are tuned towards the optimal for the human information processing system.

Appendix D describes a model of a human information processor that can be used for theoretical analysis.

The UISA described is a synthesis of theory and practically useful techniques which are necessary for meeting the requirements of future user interface support systems for interactive development of domain model-based KBS/PMS. The division of the responsibility for generating views among different classes decreases the coupling between necessary software components and facilitates the implementation.

20.4 Meta-CASE-tools

A meta-CASE tool is a tool for development of CASE-tools. Using the terminology of this thesis, a CASE-tool can be seen as a product modeling system for software. The OOCASE-tool [Johansson 93] and the meta-database was, for instance, used for the development of the meta-database itself.

Scandinavia has a long tradition in information systems modelling and the concept of a meta-CASE-tool emerged at SYSLAB⁶⁷ around 1983 from the need to experiment with tools for different methodologies for system design [Bubenko 92]. SYSLAB developed a meta-tool, RAMATIC. This tool was later further developed by the Swedish Institute for Systems Development (SISU).

The need to find objective evaluation methods for comparing various CASE-supported graphical design techniques and development

67. Systems Development and Artificial Intelligence Laboratory, Department of Computer and System Sciences, Stockholm University

methodologies has lead to the development of metrics on domain models for CASE-tools [Rossi et al. 95].

20.5 Future work

There are several research questions that need further work.

- * Improved source code generators for forms that contain a large number of fields.
- * Source code generators for graphical 2D and 3D user interfaces.
- * Development of a mathematical framework for computing usage statistics from logged user interaction sessions, which provide decision support for how to improve the user interface of a prototype system.
- * Development of the language for task descriptions so that estimations of usage statistics can be entered and used for improved automatic generation of user interface configurations.
- * Development of the benchmark domain model to cover more design features, especially for the task descriptions.
- * Development of a theory that provides an evaluation function of user interface configurations for applications that have many user roles and a large number of task descriptions.

21 Conclusions

The conclusion from practical experience with developing and maintaining a product modeling system is that the development and long-term maintenance effort for a product modeling system is quite demanding for an engineering company that has development of a specialized complex non-software product as its main business goal.

The overall development and maintenance effort can be decreased if it is divided in an intelligent way between the product developing engineering companies, specialized PMS-development consultant companies that work as mentors during the development of the system, software engineering companies and suppliers of CAD-software and core software technology such as compilers, operating systems and database engines.

This division of effort must be done in such a way that clear borders of responsibility are provided between different areas of expertise. The borders must be designed and standardized so that many companies can be formed around these areas of expertise and have a large enough market to be profitable and long-lasting.

The architectural framework suggested in the thesis is believed to be a major improvement compared to the current state of the art. The design of the development platform enables product developing engineering companies and specialized PMS-development consultant companies to focus on the development of a valid domain model for the PMS-application.

Source code generators for different database and user interface platforms can be developed independently by different software engineering companies and tested and compared using a standardized validation test on a standardized benchmark domain model. Outlines of what to include in such a standard are given in Chapter 14, section 16.5 and Appendix C.

If the design of the meta-database can be standardized and the standard is accepted by software industry, this would have an accelerating effect on the development of product modeling systems. In the same way that standardized programming languages and the development of compiler technology relieved application developers from knowing the details of assembly programming for a particular processor, this new level of abstraction will relieve the domain experts and system developers from having to deal with many of the implementation details of a particular database, CAD-system or window operating system.

References

-
- [Agrawal et al. 90] R. Agrawal, N.H. Gehani, J. Srinivasan, "OdeView: The Graphical Interface to Ode", *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 23-25, 1990.
- [Andrews et al. 87] T. Andrews, C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, (OOPSLA87)*, Oct. 1987.
- [Andrews et al. 90] T. Andrews, C. Harris, K. Sinkel, "The Ontos Object Database", *Ontologic Inc.*, 1990.
- [Ansell 88] H. Ansell, "An Application of the SPEKS Software for Load Spectrum Handling on a Transport Aircraft", *SAAB-SCANIA*, reg nr TUKH R-3705, 1988.
- [Atkinson et al. 89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, "The Object-Oriented Database System Manifesto", *Proceedings of 1st International Conference on Deductive and Object-Oriented Databases (DOOD 89)*, Kyoto, Japan, Dec. 1989.
- [Battista et al. 89] G.D. Battista, E. Pietrosanti, R. Tamassia, I.G. Tollis, "Automatic Layout of PERT Diagrams with X-PERT", *Proceedings of 3rd International Workshop on Visual Languages*, 1989.
- [Bancilhon&Kim 90] F. Bancilhon, W. Kim, "Object-Oriented Database Systems: In Transition", *ACM SIGMOD Record*, Vol. 19, No. 4, Dec. 1990, pp. 49-53.
- [Beeri 89] C. Beeri, "Formal models for Object-Oriented Databases", *Proceedings of 1st International Conference on Deductive and Object-Oriented Databases (DOOD 89)*, Kyoto, Japan, Dec. 1989, pp. 370-395.
- [Brown et al. 89] D.C. Brown, B. Chandrasekaran, "Design Problem Solving, Knowledge Structures and Control Strategies", *Morgan Kaufman Publishers, Inc.* San Mateo, California, 1989.

- [Buchanan et al. 90] B.G. Buchanan, D. Bobrow, R.Davis, J. McDermott, E.H. Shortliffe, "Knowledge-Based Systems", in [Traub et al. 90], 1990, pp. 395-416.
- [Bubenko 92] J.A. Bubenko jr, "On the Evolution of Information Systems Modelling - A Scandinavian Perspective", Report No. 92-023-DSV, Dept. of Computer and Systems Sciences, Stockholm University, Sweden.
- [Bundy 84] A. Bundy, "Intelligent Front Ends" in "Expert Systems", *Pergamon Infotec Ltd*, 1984.
- [Bylander 87] T. Bylander, B. Chandrasekaran, "Generic Tasks for Knowledge Based Reasoning: the "Right" Level of Abstraction for Knowledge Acquisition", *International Journal of Man-Machine Studies*, Vol. 26, pp. 231-243, 1987.
- [Cattell et al. 96] Cattell, R.G.G. et al, "The Object Database Standard: ODMG-93, Release 1.2", *Morgan Kaufmann Publishers, Inc.*, San Fransisco, Carlifornia, ISBN 1-55860-396-4.
- [Card et al. 83] S. Card, T. Moran, A. Newell, "The Psychology of Human Computer Interaction", *Hillsdale, New Jersey: Erlbaum*, 1983.
- [Chalmers 82] A.F. Chalmers, "What is this thing called Science?", second edition, *Open University Press*, ISBN 0-335-10107-0.
- [Chen 76] P.P.S. Chen, "The entity-relationship model: Towards a unified view of data", *ACM Transactions on Systems*, March 1976, pp. 9-36.
- [Chandrasekaran 86] B. Chandrasekaran, "Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design." *IEEE Expert*, Vol. 1, No. 3, pp. 23-30.
- [Chandrasekaran 90] B. Chandrasekaran, "Design Problem Solving: A Task Analysis", *AI Magazine*, Vol. 11, No. 4, 1990.
- [Coad&Yourdon 90] P. Coad, E. Yourdon, "Object-Oriented Analysis", *Prentice-Hall*, 1990.
- [CORBA 91] Object Management Group, "The Common Object Request Broker: Architecture and Specification", *OMG Publications*, <http://www.omg.org>.
- [Dahlbom 95] B. Dahlbom (Ed.), "The Infological Equation - Essays in Honor of Börje Langefors", *Gothenburg Studies in Information Systems*, Report 6, March 1995.
- [Dahlbäck 89] N. Dahlbäck, "A Symbol Is Not A Symbol", *Proceedings of 11th IJCAI, Detroit*, Aug., 1989.

-
- [Duff 86] C.B. Duff, "Designing an Efficient Language", *BYTE*, Aug. 1986, pp. 133-139.
- [Eades&Tamassia 89] P. Eades, R. Tamassia, "Algorithms for Automatic Graph Drawing: An Annotated Bibliography", Technical Report CS-89-09, Dept. of Computer Science, Brown Univ. 1989.
- [ElmasriNavathe 94] R. Elmasri, S.B. Navathe, "Fundamentals of Database Systems", Benjamin/Cummings, Redwood City California, 1994. ISBN 0-8053-1748-1.
- [Eriksson 89] H. Eriksson, "A Study in Domain-Oriented Tool Support for Knowledge Acquisition", Licentiate Thesis 181, Dept. of Computer and Information Science, Linköping University, 1989.
- [Eriksson 91] H. Eriksson, "Meta-Tool Support for Knowledge Acquisition", PhD Thesis 244, Dept. of Computer and Information Science, Linköping University, 1991.
- [Eshelman 88] L. Eshelman, "MOLE: A Knowledge Acquisition Tool for Cover-and-Differentiate Systems, in [Marcus 88a].
- [EXPRESS 88] Information Modeling Language EXPRESS, ISO TC184/SC4/WG1 Report N268, August 1988.
- [FahlRischSköld 93] G. Fahl, T. Risch, M. Sköld, "AMOS - An Architecture for Active Mediators", in *Proceedings Workshop on Next Generation Information Technologies and Systems (NGITS'92)*, Haifa, Israel, June 1993.
- [Fishman et al. 89] D.H. Fishman et al: "Overview of the Iris DBMS", in W. Kim, F.H. Lochovsky (eds.): *Object-Oriented Concepts, Databases and Applications*, *ACM Press, Addison-Wesley*, 1989.
- [Fredriksson 86] B. Fredriksson, "Advanced numerical methods for analysis and design of aircraft structures", *International Journal of Vehicle Design*, Vol. 7, No. 3/4, 1986.
- [Forgy 81] C.L. Forgy, "OPS5 User's Manual", Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July, 1981.
- [Gero 90] J.S. Gero (ed.) "Applications of Artificial Intelligence in Engineering V, Vol. 1 Design", *Springer-Verlag Berlin, Heidelberg*, Proceedings Fifth International Conference on Applications of AI in Engineering, Boston, USA, July 1990.
- [Goldberg&Robson 83] A. Goldberg, D. Robson, "Smalltalk-80, The Language and its Implementation", *Addison-Wesley*, Reading, Mass. 1986.

- [Goldfarb 90] C.F. Goldfarb, "The SGML Handbook", *Clarendon Press, Oxford*, 1990.
- [Gyssens et al. 90] M. Gyssens, J. Paredaens, D.V. Gucht, "A Graph-Oriented Object Model for Database End-User Interfaces", *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 23-25, 1990.
- [Gärdenfors 89a] P. Gärdenfors, "Induction, Conceptual Spaces and AI", *Philosophy of Science*, Vol. 57, 1990, pp. 78-95.
- [Gärdenfors 89b] P. Gärdenfors, "Mental Representations, Conceptual Spaces and Metaphors", Draft, Department of Philosophy, University of Lund, Sweden, 1989.
- [Hammer et al. 81] M. Hammer, D. McLeod, "Database Description with SDM: A Semantic Database Model", *ACM Transactions on Database Systems*, 1981.
- [Harmon 89] P. Harmon (editor) "How DEC is living with XCON", *The Monthly Newsletter for Managers and Developers of Expert Systems, from Cutter Information Corp*, Vol. 5, No.12, 1989.
- [Haarslev&Möller 90] V. Haarslev, R. Möller, "A Framework for Visualizing Object-Oriented Systems", *Proceedings ECOOP OOPSLA 90*, 1990, pp. 237-244.
- [Hainaut 92] J.-L. Hainaut, "A Temporal Statistical Model for Entity-Relationship Schemas", in *Proceedings of the 11th International Conference on the Entity-Relationship Approach, ER92*, Karlsruhe, Germany, October 92. Lecture Notes in Computer Science, Vol. 645, *Springer-Verlag*, ISBN 3-540-56023-8, pp. 79-96
- [Haykin 88] S. Haykin, "Digital Communications", *John Wiley & Sons*, 1988, ISBN 0-471-62947-2.
- [Helander 88] M. Helander (editor), "Handbook of Human-Computer Interaction", *North-Holland*, 1988.
- [Hickman 90] L.J. Hickman, "Towards A Convergence In Knowledge-Based And Conventional Software Engineering: Tools, Methods, Techniques", in *Proceedings of Software Engineering 90*, Brighton, July 1990, pp. 161-181.
- [Horner&Brown 90] R. Horner, D.C. Brown, "Knowledge Compilation Using Constraint Inheritance", in [Gero 90], pp. 161-174.
- [Hubel 88] A. Hubel, "Eye, Brain and Vision", *Scientific American Library*, 1988.
- [Intellicorp 87] KEE Version 3.1 Manuals, *Intellicorp*, 1987.

-
- [Jacobsson 87] I. Jacobsson, "Object Oriented Development in an Industrial Environment", *Proceedings OOPSLA 87*, 1987.
- [Jacobsson et al. 92] I. Jacobsson, M. Christerson, P. Johsson, G. Övergaard, "Object-Oriented Software Engineering - A Use Case Driven Approach", *Addison-Wesley*, Reading, MA, 1992.
- [Johansson 89] O. Johansson, "A Perspective on Engineering Database Research", Report LiTH-IDA-R-89-14, Dept. of Computer and Information Science, Linköping University, Sweden.
- [Johansson 91] O. Johansson, "Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases", *Linköping Studies in Science and Technology*, Thesis no 283, Linköping University, 1991, ISBN 91-7870-792-7, ISSN 0280-7971.
- [Johansson 93] O. Johansson, "PRODUKTMODELLER - Objektorienterad datamodellering", Tutorial given at Workshop on Product Models (PM-93), Linköping, 26-27 October, 1993. Published in PM-93, ISBN 91-7871-212-2, 1993.
- [Johansson 94] O. Johansson, "Using an Extended ER-model based Data Dictionary to Automatically Generate Product Modeling Systems", in *Applications of Databases, First International Conference, ADB-94, Vadstena, Sweden, June 21-23, 1994, Proceedings. Lecture Notes in Computer Science, Vol. 819, Springer-Verlag, ISBN 3-540-58183-9, pp. 42-61.*
- [Kieras 88] D.E. Kieras, "Towards a Practical GOMS Model Methodology of User Interface Design", in [Helander 88] pp. 135-158.
- [Kim 90] W. Kim, "Object-Oriented Databases: Definition and Research Directions", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 3, September 1990.
- [Krasner&Pope 88] G.E. Krasner, S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, Vol. 1, No. 3, Aug-Sept 1988.
- [Langefors 66] B. Langefors, "Theoretical Analysis of Information Systems.", *Lund: Studentlitteratur*, 1966.
- [Langefors 93] B. Langefors, "Essays on Infology, Summing up and Planning for the Future", *Gothenburg Studies in Information Systems*, Department of Information Systems, University of Gothenburg, Report 5, Augusti 1993.

- [Laird et al. 87] J.E. Laird, A. Newell, P.S. Rosenbloom, "SOAR: An Architecture for General Intelligence", *Artificial Intelligence*, Vol. 33, 1987, pp. 1-64.
- [Ledbetter&Cox 85] L. Ledbetter, B. Cox, "SOFTWARE-ICs: A plan for building reusable software components", *BYTE Magazine*, June 1985, pp. 28-36.
- [Lindsay&Norman 77] P.H. Lindsay, D.A. Norman, "Human information processing", *Academic Press*, 1977.
- [Lippman 92] S.B. Lippman, "C++ Primer, 2nd Edition", *Addison Wesley*, 1992, ISBN 0-201-54848-8.
- [McDonald&Buja 90] J.A. McDonald, A. Buja, "Painting multiple views of complex objects", *Proceedings ECOOP OOPSLA 90*, 1990, pp. 245-257.
- [Marcus 88a] S. Marcus (ed.), "Automated Tools for Knowledge Acquisition", *Academic Press*, 1988.
- [Marcus et al. 88] S. Marcus, J. Stout, J. McDermott, "VT: An Expert Elevator Designer that Uses Knowledge-Based Backtracking", *AI Magazine*, Spring 1988, pp. 95-111.
- [Marcus et al. 89] S. Marcus, J. McDermott, "SALT: A Knowledge Acquisition Language for Propose-and-Revise Systems", *Artificial Intelligence*, 39, 1989, pp. 1-37.
- [McDermott 82] J. McDermott, "R1: A Rule-Based Configurer of Computer Systems", *Artificial Intelligence*, Vol. 19, 1982, pp. 39-88.
- [McDermott 88] J. McDermott, "A taxonomy of problem solving methods", in [Marcus 88a].
- [Melton 95] Melton, J. (ed.), ANSI SQL3 Papers SC21 N9463 - SC21 N9467, ANSI SC21 Secretariat, New York, USA, 1995.
- [Microsoft 90] Microsoft Corp, "Microsoft Windows 3.0 Software Development Kit", Reference manuals.
- [Minsky 75] M. Minsky, "A Framework for Representing Knowledge", in P.H. Winston, Ed., "The Psychology of Computer Vision", *McGraw Hill*: New York, 1975.
- [Mittal&Araya 86] S. Mittal, A. Araya, "A Knowledge-Based Framework for Design", *AI Expert*, 1986, pp. 856-865.
- [Mittal et al. 86] S. Mittal, C. Dym, M. Morjaria, "PRIDE: An Expert System for the Design of Paper Handling Systems", *IEEE Computer*, Vol. 19, No. 7, 1986, pp. 102-114.
- [Mostow 85] J. Mostow, "Towards Better Models Of The Design Process", *AI Magazine*, Spring, 1985, pp. 44-57.

-
- [Musen 89] M.A. Musen, "Conceptual models of interactive knowledge acquisition tools", *Knowledge Acquisition*, No. 1, pp. 73-88, 1989.
- [Navathe 92] S.B. Navathe, "The Next Ten Years of Modeling, Methodologies, and Tools", in *Proceedings of the 11th International Conference on the Entity-Relationship Approach, ER92*, Karlsruhe, Germany, October 92. Lecture Notes in Computer Science, Vol. 645, *Springer-Verlag*, 1992, ISBN 3-540-56023-8, pp. 3-6.
- [Newell 82] A. Newell, "The Knowledge Level", *Artificial Intelligence*, Vol. 18, 1982, pp. 87-127.
- [Norman 83] D.A. Norman, "Some observations on mental models". In D. Gentner, A. Stevens (eds.) *Mental Models*, Hillsdale NJ: *Erlbaum*, 1983.
- [Orsborn 91] K. Orsborn, "Using Knowledge-Based Technique in Systems for Structural Design", *Proceedings of the 1991 International Conference on Computational Structures Technology*, Edinburgh, August, 1991.
- [Orsborn 93] K. Orsborn, "Structural Design Systems Using Knowledge-Based Techniques, Applications to Damage Tolerance Design of Aircraft Structures", Linköping Studies in Science and Technology, Thesis No. 400, Linköping University, 1993,
- [Rao 87] R. Rao, S. Wallace, "The X Toolkit", *Proceedings of USENIX Conference Summer 88*.
- [Reenskaug 96] T. Reenskaug, P. Wold, O.A. Lehne, "Working With Objects, The OOram Software Engineering Method", *Manning Publications Co.*, ISBN 1-884777-10-4.
- [Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", *Prentice-Hall, Eaglewood Cliffs*, 1991, ISBN-0-13-630054-5.
- [Rossi et al. 95] M. Rossi, S. Brinkkemper, "Metrics in Method Engineering", *Proceedings of Advanced Information Systems Engineering, CAiSE'95*, Jyväskylä, Finland, June 12-16, 1995, Lecture Notes in Computer Science Vol. 932, *Springer-Verlag*, 1995, ISBN 3-540-59498-1, pp. 200-216.
- [SAAB 90] SAAB Scania Aircraft Division, "CAFE - Computer Aided Fracture Analysis, Users Manual", *SAAB Aircraft Division*, Linköping.

- [Sandewall 78] E. Sandewall, "Programming in an interactive environment: the "Lisp" experience.", *ACM Computing Surveys*, Vol. 10, No. 1, pp. 35-71, 1978.
- [Sandewall 88] E. Sandewall, "Future Developments in Artificial Intelligence", Invited talk, *Proceedings of European Conference on Artificial Intelligence (ECAI)*, Munich, August 1988.
- [Sandahl 87] K. Sandahl, "Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems", Licentiate Thesis 127, Linköping University, Dept. of Computer and Information Science, 1987.
- [Schank 82] R.C. Shank, "Dynamic Memory", *Cambridge University Press*, Cambridge, 1982.
- [Schaffer et al. 65] L.R. Schaffer, J.B. Ritter, W.L. Meyer, "The Critical- Path Method", McGraw-Hill, 1965.
- [Schaufler 88] R. Schaufler, "The X11/NeWS Design Overview", *Proceedings of USENIX Conference*, Summer 1988, pp. 22-35.
- [Shaw et al. 90] M.L.G. Shaw, J.B. Woodward, "Modeling expert knowledge", *Journal of Knowledge Acquisition*, Vol 2, pp. 179-206, 1990.
- [Shan 90] Y.P. Shan, "MoDE: A UIMS for Smalltalk", *Proceedings ECOOP OOPSLA 90*, pp. 258-268, 1990.
- [Shannon 48] C.E. Shannon, "A Mathematical Theory of Communication", *Bell System Technical Journal*, Vol. 27, pp. 379-423 and pp. 623-656, 1948.
- [ShannonWeaver 49] C.E. Shannon, W. Weaver, "The Mathematical Theory of Communication", *University of Illinois Press*, 1949, ISBN 0-252-72548-4.
- [Shipman 81] D.W. Shipman: "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981.
- [Smith&Smith 77] J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", *ACM Transactions on Database Systems*, Vol. 2, No. 2, 1977.
- [Staley et al. 86] S.M. Staley, D.C. Anderson, "Functional specification for CAD databases", *Computer Aided Design*, Vol. 18, No. 3, pp. 132-138, 1986.
- [Steele 84] G. L. Steele jr, "Common LISP: The Language", *Digital Press*, 1984, ISBN 0-932376-41-X.

-
- [Steels 84] L. Steels, "Second Generation Expert Systems", *Conference on Future Generation Computer Systems*, Rotterdam. Reprinted in *Journal of Future Generation Computer Systems*. Vol.1, No. 4, *North-Holland, Pub. Amsterdam*, June, 1985.
- [Steels 89a] L.Steels, "Expert Systems Terminology", AI Memo no 89-8, VUB AI LAB, Excerpt from a session at the workshop on foundations of expert systems, held at Paco De Gloria, Portugal, March, 1989.
- [Steels 90] L. Steels, "Components of Expertise", *AI Magazine*, Vol. 11, No. 2, Summer 1990.
- [STEP 92] International Standard ISO 10303 - Industrial automation systems and integration - Product data representation and exchange.
- [STEP 92a] International Standard ISO 10303 - Industrial automation systems - Product data representation and exchange. Part 11: Description methods: The EXPRESS language reference manual.
- [Stroustrup 93] B. Stroustrup, "The C++ programming language, second edition", Addison Wesley, 1993.
- [Sybase 94] "SYBASE SQL Server, Transact-SQL User's Guide", Release 10.0, Sybase Inc., 1994.
- [Sybase 94a] "SYBASE SQL Server Reference Manual. Volume 1 Commands, Functions and Topics", Release 10.0, Sybase Inc., 1994.
- [Sun 89] Sun Microsystems, "OpenWindows 1.0 Manual Set", Rev 50 26, May 1989.
- [Sundgren 73] B. Sundgren, "An Infological Approach to Data Bases", National Central Bureau of Statistics, Sweden, and University of Stockholm, Dept. of Administrative Information Processing, *Beckmans Tryckerier AB*, Stockholm 1973.
- [Sundgren 77] B. Sundgren, "The elements of largeness - A contribution to the theory of information systems.", In R.A. Buckingham (ed) *Education and Large Information Systems*, Proceedings from IFIP TC3 och TC8 Conference North-Holland Publishing Company, 1977.
- [Sundgren 89] B. Sundgren, "Conceptual Modelling as an Instrument for Formal Specification of Statistical Information Systems", National Central Bureau of Statistics Sweden, 1989:18.

- [Sundgren 92] B. Sundgren, "Databasorienterad Systemutveckling", Studentlitteratur, 1992, ISBN 91-44-35991-8 (in Swedish).
- [Tong 87] C. Tong, "AI in Engineering Design", *Artificial Intelligence in Engineering*, Vol. 2, No. 3, 1987.
- [Traub et al. 90] J.F. Traub, B.J. Grosz, B.W. Lampson, N.J. Nilsson (eds), "Annual Review of Computer Science", *Annual Reviews Inc*, Palo Alto California, Vol. 4, 1990.
- [Tullis 88] T.S. Tullis, "A system for evaluating screen formats: Research and application" in H.R.Hartson, D. Hix (eds) "Advances in Human-Computer Interaction Vol 2", *Ablex Publishing Corp., NJ*, 1988, pp. 214-286.
- [Wegner 87] P. Wegner, "Dimensions of Object-Based Language Design", *Proceedings of OOPSLA 87*, 1987, pp. 168-182.
- [Weinand et al. 88] A. Weinand, E. Gamma, R. Marty, "ET++ - An Object-Oriented Application Framework in C++", *Proceedings of OOPSLA 88*, pp. 46-57.
- [Whitewater 90] Whitewater Group, Inc, "Actor User's Manuals (Version 2.0)", Whitewater Group Inc, 1800 Ridge Avenue, Evanston, IL 60201, USA, Fifth printing, May, 1990.
- [Ungar et al. 87] D. Ungar, R.B. Smith, "Self: The Power of Simplicity", *Proceedings of OOPSLA 87*, 1987, pp. 227-242.
- [Young 89] D.A. Young, "X-Window Systems, Programming and Applications with Xt", *Prentice Hall*, New Jersey, 1989.
- [Zdonik 90] S.B. Zdonik, D.M. Maier, "Readings in Object-Oriented Database Systems", *Morgan Kaufmann Publishers, Inc*, San Mateo, California, 1990.

Appendixes

Appendix A The Benchmark Domain Model

An overview of the suggested benchmark domain model is given in chapter 13. This appendix provides the following additional documentation :

Table 25: List of documentation of the benchmark domain model.

| Documentation type | | Page no |
|---|-----------|------------|
| Graphical overview of the benchmark domain model. | Figure 44 | 183 |
| Class list | Table 26 | 184 |
| Relationship list | Table 27 | 185 |
| Attribute list | Table 28 | 186 |

Table 26: Classes from the benchmark domain model.

| Name | Pfx | SuperClass | Definition |
|---------|-----|-----------------------|--|
| Article | ar | QualitySecured Object | An article specifies a single part or an assembly of other articles. Other articles may either be owned directly by the article, or included from assembly structures owned by other articles. |
| C | c0 | QualitySecured Object | Abstract/instantiable superclass for the class hierarchy that inherits the product_features relationship. |
| C1 | c1 | C | First instantiable leaf class C1 in the subclass hierarchy of C. |
| C2 | c2 | C | Abstract/instantiable class. |
| C21 | ca | C2 | Instantiable leaf class. |
| C22 | cb | C2 | Instantiable leaf class. |
| C3 | c3 | C | Abstract/instantiable class. |
| C31 | cc | C3 | Abstract/instantiable class. |
| C311 | cd | C31 | Instantiable leaf class. |
| C32 | ce | C3 | Abstract/instantiable class. |
| C321 | cf | C32 | Instantiable leaf class. |
| C322 | eg | C32 | Instantiable leaf class. |
| C33 | ch | C3 | Abstract/instantiable class. |
| C331 | ci | C33 | Instantiable leaf class. |
| C332 | cj | C33 | Instantiable leaf class. |
| C333 | ck | C33 | Instantiable leaf class. |
| Company | co | DesignationObject | A company. |

Table 26: Classes from the benchmark domain model.

| Name | Pfx | SuperClass | Definition |
|-----------------------|-----|-----------------------|--|
| DatabaseObject | db | (none) | DatabaseObject is the superclass of all classes that are stored in the database. It holds a globally unique object identifier (lowId,highId) that serves as a key. |
| Deliverable | dl | DesignationObject | Defines a deliverable that is produced by a DesignActivity. |
| Department | de | DesignationObject | A department within the company that is responsible for a set of products. |
| DesignActivity | da | DesignationObject | Specifies an activity that may take a set of input deliverables and produce a set of output deliverables. |
| DesignationObject | do | DatabaseObject | Abstract class for objects with a human readable designation. |
| Drawing | dr | QualitySecured Object | A drawing that depicts a set of articles. |
| IncludedArticle | ia | DatabaseObject | Relates the inclusion of an article in the assembly of another article. |
| Product | pr | QualitySecured Object | One of several products that are designed within a department. |
| ProductData | pd | DesignationObject | Class with attributes of the basic types of CORBA IDL. |
| QualitySecured Object | qs | DesignationObject | Defines objects that need quality assurance. |

Table 27: Relationships from the benchmark domain model.

| Relationship | type | owns 1to2 | Definition |
|---------------------|------|--------------|---|
| company_departments | 1-N | T | Holds departments related to a company. |
| company_drawings | 1-N | T | A master index for the drawings within a company. |

Table 27: Relationships from the benchmark domain model.

| Relationship | type | owns 1to2 | Definition |
|-----------------------------|------|--------------|---|
| consumer_inputs | 1-N | F | Defines inputs consumed by DesignActivities. |
| deliverable_articles | 1-N | F | Defines articles within a deliverable. |
| deliverable_drawings | 1-N | F | Defined drawings within a deliverable. |
| department_products | 1-N | T | Holds products owned by a department. |
| drawing_articles | 1-N | F | Relates a set of articles to a drawing. |
| include_includedIn | 1-N | F | Relates articles that include an article. |
| owner_includes | 1-N | T | Holds included article reference objects for an article. |
| owner_owns | 1-N | T | Holds articles that are owned by other articles. |
| producer_outputs | 1-N | T | Specifies deliverables produced by design activities. |
| product_features | 1-N | T | Relationship from product to class C, which is inherited to C's subclasses. |
| product_mainActivity | 1-1 | T | Relates one main activity to a product. |
| product_mainArticle | 1-1 | T | Relates one main article to a product. |
| product_productData | 1-N | T | Holds fictitious product data for a product. |
| superActivity_subActivities | 1-N | T | Holds a sets of subactivities that make up a super activity. |

Table 28: Attributes of the benchmark domain model.

| Name | ClassName | Type | Definition |
|----------|-------------|---------|---|
| aBoolean | ProductData | boolean | CORBA IDL basic typedef boolean. |
| aC | C | boolean | An attribute of class C on the 4th inheritance level. |

Table 28: Attributes of the benchmark domain model.

| Name | ClassName | Type | Definition |
|---------|-------------|---------------|--|
| aC1 | C1 | boolean | Attribute of class C1 on the 5th inheritance level. |
| aC2 | C2 | boolean | Attribute on the second subclass of class C. |
| aC21 | C21 | boolean | Attribute of class C21 on the 6th inheritance level. |
| aC22 | C22 | boolean | Attribute of class C22. |
| aC3 | C3 | boolean | Attribute of class C3. |
| aC31 | C31 | boolean | Attribute of class C31. |
| aC311 | C311 | boolean | Attribute of class C311, on the 7th inheritance level. |
| aC32 | C32 | boolean | Attribute of class C32. |
| aC321 | C321 | boolean | Attribute of class C321. |
| aC322 | C322 | boolean | Attribute of class C322. |
| aC33 | C33 | boolean | Attribute of class C33. |
| aC331 | C331 | boolean | Attribute of class C331. |
| aC332 | C332 | boolean | Attribute of class C332. |
| aC333 | C333 | boolean | Attribute of class C333. |
| aChar | ProductData | char | CORBA IDL basic typedef char. |
| aDouble | ProductData | double | CORBA IDL basic typedef double. |
| aFloat | ProductData | float | CORBA IDL basic typedef float. |
| aLong | ProductData | long | CORBA IDL basic typedef long. |
| aShort | ProductData | short | CORBA IDL basic typedef short. |
| aString | ProductData | string<255> | CORBA IDL basic typedef String. |
| anEnum | ProductData | short | CORBA IDL basic typedef enum. |
| anULong | ProductData | unsigned long | CORBA IDL basic typedef unsigned long. |

Table 28: Attributes of the benchmark domain model.

| Name | ClassName | Type | Definition |
|-------------|-----------------------|----------------|---|
| anUShort | ProductData | unsigned short | CORBA IDL basic typedef unsigned short. |
| approved_by | QualitySecured Object | string<8> | Login name in database for employee who has approved this information for manufacturing (or purchase). |
| checked_by | QualitySecured Object | string<8> | Login name in database for designer who has checked the information recorded in this database object. |
| created_by | DatabaseObject | string<8> | Login name in database for the designer who originally created this object. The date for this is specified by the attribute dtAdded. |
| highId | DatabaseObject | long | Higher 32-bit part of a 64-bit globally unique object identifier. Used for uniquely identifying a certain database. |
| lowId | DatabaseObject | long | Lower 32-bit part of a 64-bit globally unique object identifier. Used for uniquely identifying objects within the same database. |
| modified_by | DatabaseObject | string<8> | Login name in database for designer who made the latest modification to this information. The date for this is indicated by the attribute dtModified. |
| name | Designation Object | string <40> | A descriptive name for the object. |
| quantity | Article | string<8> | An integer that specifies the quantity of this article that are parts of its owner. |
| quantity | IncludedArticle | string<8> | An integer that specifies the quantity of one article that is included in another article. |

Table 28: Attributes of the benchmark domain model.

| Name | ClassName | Type | Definition |
|------------------------|--------------------------|-----------|---|
| ref_id | Designation Object | string<8> | Fast human readable reference identifier. Could serve as candidate key in a relational database. |
| release | QualitySecured Object | string<4> | Release identification. Identifies a release of the information in this databaseobject. Release 0 is the working release. |
| scheduled_ duration | DesignActivity | long | Scheduled duration of the activity. |
| status | QualitySecured Object | string<3> | Status of the information given, according to a company-specific standard. |

Appendix B Benchmark Application Task Descriptions

This appendix just sketches how the benchmark task descriptions can be defined. The detailed work of developing a working benchmark must be made by a standardisation organization or software consortium.

```

ProcessModel benchmark;
  Role ExecutiveDirector;
  Role ChiefDesigner;
  Role ProjectManager;
  Role Designer;
...
Task ArticleChecking
  performed by ChiefDesigner;

  Transactions
    Begin CompletenessCheck
      InfoSet
        Path Company.departments(1)->          Department.products(1)->
          Product.mainArticle(1)->
            { Article.owns(*)-> }*
          Article
        Read Company.name(1)      1.0
        Read Department.ref_id(1) 1.0
        Read Product.ref_id(1)   1.0
        Read Product.name(1)     1.0
        { Read Article.ref_id(*)  1.0
          Read Article.name(*)    1.0
          Read,Create,Update
            Article.checked_by(*) 1.0
        }*
      EndInfoSet
    End
  ...
EndTransactions
EndTask

```

The “Transactions” statement includes a set of transactions that each specifies a certain InfoSet. The infoSet includes one or several Paths which specify how the contents of the infoSet are located. The path name has the format [**<Class>.<rXX>->]*<Class>**. The default name of an **aect** role is **<Class>.<attributeName>(<rolename>) <importance_weight>**. The role * indicates that the role name is the number of the hierarchical nesting level.

Appendix C User Interface Object Characteristics

Chapter 18 gave an introduction to the different types of user interface objects that are used to implement the state of the user interface (Figure 41 on page 150). Chapter 19 described the display-, selection-, clipboard- and transaction manager, which handle some important dynamic interaction mechanisms between different user interface objects.

This appendix presents a suggested conceptual framework and terminology that enable different user interface configurations to be evaluated and compared using theoretical models. The framework has been implemented for the benchmark application.

An informal definition of a user interface configuration is: A set of window types where each window type has predefined what kind of information it can display and provide access to, and how. The user interface configuration also states which window types may be used by which user roles, and during which tasks.

To be able to make predictions of how a certain user interface configuration will behave in various situations, we must know the operations available for the kinds of user interface objects they may contain, and have some statistics about average completion times for these operations.

Section C.1 "Information recorded in a log record" describes a suggested set of information to log for operations available on user interface objects. Section C.2 describes special operations which the transaction manager handles.

The rest of the sections describe operations and measurable statistics for window types, object editor types, attribute editors, relationship1-, and relationshipN editors. Most of them are average timings. The timing parameters can be used for calculating average completion times for tasks performed by a MHP (Model Human Processor) actor given a certain user interface configuration.

C.1 Information recorded in a log record

After completing an operation issued by the user, a user interface object sends a message to the transaction manager to log the operation. Table 29 shows the information recorded in a log record for the user interface operations presented in Table 33 - Table 45.

Table 29: Recorded log record information for a user interface operation.

| attribute name | datatype | Description |
|----------------------|----------|---|
| <i>domainModel</i> | string | Name of the domain model for the application from which the log record was generated. |
| <i>application</i> | string | The name of the application generating this domain model. |
| <i>userId</i> | string | Login identifier for user who made the operation. |
| <i>sessionId</i> | long | Unique identifier for the session which the log record was generated from. |
| <i>transactionId</i> | long | Unique identifier for the transaction within which the log record was generated. |
| <i>timestamp</i> | long | Integer timestamp when the log record was generated. |
| <i>duration</i> | long | Duration in milliseconds for completion of the logged operation. |
| <i>windowId</i> | string | Unique textual identifier within the session for the window in which the user made the logged operation. |
| <i>uiObjectId</i> | string | Unique textual identifier within the window, for the <i>uio</i> which processed the users command for the operation. The identifier includes the type and an instance identifier, such that log-entries from the same <i>uio</i> can be traced. <i>Uios</i> may be of any type presented in Appendix C. |
| <i>operation</i> | string | Name of the logged operation. |
| <i>newValue</i> | string | A new value that is specific for the <i>uiObject</i> and operation. See column 1 in the tables in Appendix C. |
| <i>oldValue</i> | string | A previous value that is specific for the <i>uiObject</i> and operation. See column 1 in the tables in Appendix C. |
| <i>iqVisible</i> | long | Information quantity made visible by this operation. |
| <i>iqRead</i> | long | Information quantity read during the operation. |
| <i>iqCreated</i> | long | Information quantity created by this operation. |
| <i>iqDeleted</i> | long | Information quantity deleted by this operation. |

Log records from different applications and different domain models may be stored in the same table in a relational database for various analyses.

The *application* attribute should include a version identifier, to be able to distinguish logs recorded from different prototype versions.

The transaction manager is responsible for providing *userId*, *sessionId*, *transactionId* and *timestamp* to the log record. It is also recommended to channel all accesses to the global clock for calculating *durations* through the transaction manager, where the clock can be guarded by a monitor and thus the calls for timing information made strictly serialized.

iqVisible, *iqRead*, *iqCreated*, *iqDeleted* are only recorded for the operations that are marked with (iqv),(iqr),(iqc),(iqd) in Table 33 - Table 45.

C.2 Logging begin and end of user transactions

The begin of a user transaction is defined to be when an e-message presented in a window is modified through an operation, and not yet stored in the KB. This occurs for instance when the user modifies a value in an attribute editor on a form based window.

Definition : A window is said to be *valid* when the e-messages it presents have a direct 1-1 correspondence with the e-constellations in the KB.

A new transactionId is generated and a log entry that marks the begin of the transaction is inserted automatically by the transaction manager the first time it logs a modifying nonatomic operation from a user interface object in a *valid* window.

Table 30: Special operations generated by the transaction manager.

| operation kbo for uio newValue oldValue | U T | Operation description |
|--|--------|--|
| beginTran oid - - | ? B | Inserted into the log by the transaction manager when it records the first ?B operation from an <i>uio</i> with a <i>kbo</i> that is not currently invoked in a user transaction. Generates a new transactionId and registers it for the oid of the <i>kbo</i> represented by the <i>uio</i> . |
| commitTran oid - - | ? C | Inserted by the transaction manager when it records a ?C operation from an <i>uiobject</i> that represents a <i>kbo</i> where the oid has a pending transaction. |
| rollbackTran oid - - | ? R | Inserted by the transaction manager when it records a ?R operation from an <i>uiobject</i> that represents a <i>kbo</i> where the oid has a pending transaction. |

Such modifying operations are marked by the flag ‘?B’ in the tables of operations presented in Table 33 - Table 45. Log entries that are automatically inserted by the transaction manager for ?B, ?C and ?R operations are listed in Table 30.

The end of a user transaction is defined to be when the user gives a command that invokes a ?C or ?R marked operation. The invocation may be explicit by pressing a button, or implicit when the user has changed some values on a form browser and tabs to an attribute editor in another object editor.

The kbCreate-command on a 1-N relationship editor is an example of an atomic operation marked B-C. (see Table 42 on page 208). It needs to commit its changes immediately to the KB in order to refresh the relationship editor correctly.

In an advanced graphical user interface, different e-messages representing the same e-constellation can be visible on many parallel windows. If the user has pending changes of the representation for the same e-constellation in several windows, he/she must consciously be forced to decide which update should be committed, and which one should be rolled back.

The situation often occurs when the user interleaves a started modification activity with some browsing, and then forgets about the previous started update.

In order to detect conflicting updates, the transaction manager keeps track of the uio and kbo for all pending transactions. When a modifying (?B-marked) uioperation is logged, it checks if there is another uio with a pending transaction that represents the same kbo. If there is, a conflict has been detected. There are several ways to resolve the conflict.

- 1) Just to rollback the oldest initiated transaction, using a ?R operation, such as uiRefresh on the *uio* containing the data.
- 2) Bring the window with the conflicting old transaction to the front and inform the user that the changes in that window will be rolled back.
- 3) Present both windows to the user, and ask which pending transaction to rollback.

Conflicts must be resolved immediately when they are detected by using a method like the ones presented above. Otherwise there is risk that the user will overwrite later committed data with a store operation in a window containing modified data based on an old copy of data from the database.

C.3 Window types

Windows are a basic medium for display and interaction with information on a workstation. Average task completion times depend severely on how

different types of application related information are allocated to, and handled by different types of windows. How the windows are used by the user during the completion of a task may reveal significant information on how a window type can be redesigned in order to improve task completion time, and reduce the cognitive load on the user.

Figure 42 on page 152 shows the object domain model for the state of the user interface. A window type is a predefined configuration that specifies the state which contains the layout of *uios* within the window when it is opened. A window type may also be a view, where its state, represented by object editors and links, depends on the structures within the knowledge base.

Table 31 show a set of attributes for a window type. Table 33 shows a set of basic operations to log for windows.

Table 31: *Window type* attributes.

| Name | Description |
|--------------------------|---|
| <i>name</i> | Class name for the window. |
| <i>windowTypePrefix</i> | A 4-character prefix that uniquely identifies a certain window class within an application. |
| <i>avgTimeOpen</i> | Average time to open the window in milliseconds. |
| <i>avgTimeClose</i> | Average time to close the window in milliseconds. |
| <i>avgTimeRefresh</i> | Average time to refresh the window in milliseconds. |
| <i>avgTimeVisualize</i> | Average time in milliseconds to perform an operation that redraws the window. The operations included in this average time are listed in Table 31. |
| <i>ratio</i> | Defines the maximum error in the statistical estimates above, $\text{ratio} = dx/x$, where dx is the maximum error in x . |
| <i>targetSystemRange</i> | An integer that specifies an interval in which other target system implementations may have their average timings. The interval is between $[x/\text{tsr}, x*\text{tsr}]$, where $\text{tsr} = \text{targetSystemRange}$. |

Table 32: *Window* visualization operations.

| |
|--|
| Window operations in Table 33 that count as visualisation operations. |
| <i>uiOpen</i> , <i>uiMaximize</i> , <i>uiRestore</i> , <i>uiSize</i> , <i>uiBringToFront</i> , <i>uiRedraw</i> , <i>uiRefresh</i> , <i>uiSet</i> |

Table 33: Basic operations to log for windows.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|---|
| uiOpen (iqv) oid windowRect - | Form 2D | - | Create the internals of the window, open it and refresh the information. |
| uiMinimize oid iconRect windowRect | Form 2D | - | Change the window representation to an icon. |
| uiMaximize oid windowRect prevWindowRect | Form 2D | - | Enlarge the window to its maximum size. |
| uiRestore (iqv) oid windowRect iconRect | Form 2D | - | Change the window representation from icon to open. Refresh the information. |
| uiMove oid newWindowRect oldWindowRect | Form 2D | - | Move the window interactively, and redraw it. |
| uiSize (iqv,iqr) oid newWindowRect oldWindowRect | Form 2D | - | Change the window size interactively and refresh the window. |
| uiBringToFront oid windowRect - | Form 2D | - | Move the window to the front of the window stack and, redraw it if necessary. |
| uiSendToBack oid - windowRect | Form 2D | - | Send the window to the bottom of the window stack. Redraw other windows as necessary. |
| uiRedraw (iqv) oid windowRect windowRect | Form 2D | - | Redraw the window. |

Table 33: Basic operations to log for windows.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|---------|--|
| uiRefresh (iqv,iqr) oid newWindowRect oldWindowRect | Form 2D | ?R | Update the window with any changes in the database. |
| kbStore (iqc) oid windowRect windowRect | Form | ?C | Store any changes on data presented in the window to the knowledge base. |
| uiSet (iqv,iqr) newOid newOid oldOid | Form 2D | - | Set the window to reflect the object held in the selection, if possible. |
| uiSetFrom kbo newUiObjectId oldUiObjectId | Form 2D | - | Set the window to reflect any object selected in the <i>uio</i> that owns the current selection. |
| uiSetLock kbo | Form | - | Lock the window on its current kbo. Inhibits the kbEdit operation from using the editor for another kbo. |
| uiSetFree kbo | Form | - | Free the window from its current kbo. The window is now a free resource for use during browsing. |
| uiCancel oid - windowRect | Form | ?R | Close the window and ignore any unstored changes to data. |
| uiClose oid - windowRect | Form 2D | Q ?C | Ask the user to store any unstored changes, then close the window and destroy its representations permanently. |

C.4 Object editor types

An object editor type is a predefined configuration of owned attribute- and relationship editors that define what subset of **aect**'s and **rect**'s within a domain model that the *oe* will provide access to. The object editor type also specifies the graphical layout of the attribute and relationship editors.

Object editors may be chained through relationship editors in such a way that they reflect a browsing path from *kbo*s of one class via a relationship to *kbo*s of another class, and so on.

In dynamic views, *oe*'s may be created dynamically, reflecting how related *kbo*'s in the knowledge base are structured.

Table 34: *Object Editor type* attributes.

| Name | Description |
|-----------------------------------|---|
| <i>name</i> | Class name for the object editor. |
| <i>keyName</i> | Unique name for the object editor configuration within a certain window class. |
| <i>avgTimeRefresh</i> | Average time to refresh the object editor in milliseconds. |
| <i>avgTimeVisualize</i> | Average time in milliseconds to redraw the object editor. |
| <i>isDynamic</i> | Flag set to true if the number of instances of this object editor type, within a window can vary dynamically during run time. |
| <i>probabilityZeroCardinality</i> | Probability that there are no instances of this object editor type within the window. |
| <i>avgCardinality</i> | Average number of dynamic object editors of this type within the window during runtime. |
| <i>stdCardinality</i> | Standard deviation of the above measure. |
| <i>ratio</i> | Defines the maximum error in the statistical estimates above, $ratio = dx/x$, where dx is the maximum error in x . |
| <i>targetSystemRange</i> | An integer that specifies an interval in which other target system implementations may have their average timings. The interval is between $[x/tsr, x*tsr]$, where $tsr = targetSystemRange$. |

Table 35: Basic operations to log for object editors.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|---|
| kbRead (iqv,iqr) oid - - | Form 2D | - | Some object editors may only indicate the existence of an object. In those cases an explicit read command may be issued in order to quickly display object information on a temporary popup window. |
| kbCreate (iqv,iqc) newOid newOid oldOid | Form | B C | Create a new object instance and load it to the editor. |
| kbEdit oid editOid - | Form 2D | - | Open a default object-specific editor on the <i>kbo</i> represented by the object editor. |
| kbStore (iqc,iqd) oid - - | Form | ? C | Store any pending changes made to data in owned attribute editors to the database. |
| kbDelete (iqd) oid - deletedOid | Form 2D | B C | Delete the kbo represented by the object editor. In the 2D case, remove the object editor from the window. |
| uiRefresh (iqv,iqr) kbo - - | Form 2D | ? R | Update all owned editors with fresh data read from the KB. |
| uiClear - - oldOid | Form | - | Clear the editor from its currently loaded kbo. Clears the attribute fields, so a new uiLoad pattern can be entered. |
| uiLoad (iqv,iqr) loadedOid loadedOid oldOid | Form | - | Load an object from the database that matches the pattern data entered in the attribute editors. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| uiChange (iqc,iqd) oid newRepr. oldRepr. | 2D | ? C | Change the graphical representation of the object editor. This changes the layout in the window, and maybe some data stored in the object that are represented in the layout. "Repr" is a textual representation for the layout. |
| uiSet (iqv,iqr) newOid newOid oldOid | Form 2D | - | Set the object editor to reflect the object held in the selection, if possible. |
| uiRemove oid - oldRepr. | 2D | - | Remove object editor from the window. |
| setSelectionOwner oid - - | Form 2D | - | Clear any previous selections, and make the object editor the single owner of the selection. |
| addSelected oid - - | Form 2D | - | Add the object to the selection held by the selection manager. |
| removeSelected oid - - | Form 2D | - | Remove the object from the selection held by the selection manager. |
| clearSelections oid - - | Form 2D | - | Clear all selections held by the selection manager. |
| kbCopy (iqr) oid copyOid - | Form 2D | B C | Copy the kbo in the object editor to the clipboard. |
| kbPaste(iqv,iqc,iqd) oid pasteOid copyOid | Form 2D | B C | Paste the contents of the clipboard to the kbo represented by the object editor, if possible. |

C.5 Attribute editor

An attribute editor has a one-to-one mapping to an attribute e-message type (*aemt*) in the domain model. Most of the statistical timing estimates for an attribute editor are dependent on statistics that describe the attribute value. Thus the statistical parameters for attribute editors are implemented as methods that take the average value size in a convenient measure, depending on the type of the value, as input parameter. For a string value editor, the average value size is the number of characters.

Table 36: *Attribute Editor* attributes.

| Name | Description |
|------------------------------|---|
| <i>className</i> | Class name for the attribute editor. |
| <i>uiName</i> | Unique name for the attribute editor within an object editor configuration. Used for logging purposes. |
| <i>avgInformationDensity</i> | A measure of information quantity or e-messages per 100 square pixels. |
| <i>ratio</i> | Defines the maximum error in the statistical estimates returned by the attribute editor methods in Table 37. $ratio = dx/x$, where dx is the maximum error in x . |
| <i>targetSystemRange</i> | An integer that specifies an interval in which other target system implementations may have their average timings. The interval is between $[x/tsr, x*tsr]$, where $tsr = targetSystemRange$. |

Table 37: *Attribute Editor* methods.

| Name | Description |
|--------------------------------------|---|
| <code>avgTimeRead(avgSize)</code> | Calculate an average time in milliseconds for a user to read and interpret a value given the average value size <code>avgSize</code> . |
| <code>avgTimeCreate(avgSize)</code> | Calculate the average time in milliseconds for a user to create a value in this attribute editor, given the average value size <code>avgSize</code> . |
| <code>avgTimeUpdate(avgSize)</code> | Calculate the average time in milliseconds for a user to update a value in this attribute editor, given the average size <code>avgSize</code> . |
| <code>avgTimeDelete(avgSize)</code> | Calculate the average time to delete a value in this attribute editor, given the average value size <code>avgSize</code> . |
| <code>avgTimeRefresh(avgSize)</code> | Average time to refresh the attribute editor in milliseconds given the average value size <code>avgSize</code> . |

avgTimeRead(avgSize) needs a comment. The model human processor assumes 230 milliseconds for a saccadic eye movement. An explicit e-messages that can be received in one eye fixation and requires no further inference would probably take no more than 1/2 a second to read and interpret. Some coding representation techniques may require extra mental reasoning. Text containing more than 15-20 characters may need several eye fixations to read. For attribute editors with complex value representations, data has to be collected experimentally.

The average time to modify a value using a certain attribute editor is probably most convenient to measure with experiments. Log functions can be built into the attribute editors and a set of test data distributed over the value size range generated. After a simple practical experiment, a function returning a reasonable estimate can be found and implemented.

avgTimeRefresh(valueSize): System average widget refresh time in milliseconds. This value is also easy to find out experimentally. The value can for instance be measured by creating a form with 100 attribute editors of the type holding data of valueSize, and time the refresh method.

Table 38: Basic operations to log for attribute editors.

| operation kbo for attribute newValue oldValue | UI type | U T | Operation description |
|--|----------------|-----------------------|---|
| vaRead (iqv,iqr) oid value - | Form 2D | - | Some attribute editors may only indicate the existence of a value. The value may be read by issuing an explicit read command. |
| vaVisit oid value - | Form 2D | - | Issued when the user temporarily enters the attribute editor, leaving the original value intact when exiting. |
| vaCreate (iqv,iqc) oid newValue - | Form 2D | ? B B- C | Issued when the previous value carried no information. It may have had the value “undefined” or a “dummy” default value. |
| vaUpdate(iqv,iqc,iqd) oid newValue oldValue | Form 2D | ? B B- C | Update the value in the attribute editor. The 2D-case commits immediately. |

| operation kbo for attribute newValue oldValue | UI type | U T | Operation description |
|--|----------------|-------------------|---|
| vaDelete (iqd) oid - oldValue | Form 2D | ? B B- C | Set the value to “undefined” or to a non-information carrying default value. |
| kbStore (iqc) oid newValue oldValue | Form | ? C | Explicitly store any changes on the value presented in the attribute editor to the knowledge base object. |
| uiRefresh (iqv,iqr) oid newValue oldValue | Form 2D | ? R | Explicitly update the attribute editor with fresh data read from the database. |
| vaCopy (iqr) oid value - | Form, 2D | B C | Copy the value of the attribute editor to the clipboard. |
| vaPaste(iqv,iqc,iqd) oid pastedValue oldValue | Form, 2D | B C | Paste the contents of the clipboard to the attribute editor, if it contains a legal value. |

C.6 Relationship1Editor

A relationship1 editor is used for accessing a 1-1 relationship from both sides, and a 1-N relationship from the class2 side. Figure 45 shows 4 examples of relationship1 editors labelled Product, Deliverable, Drawing and Owner. These are all on the 2-side of their relationships, in the way they are defined in the benchmark domain model. The Product relationship1 editor is the only relationship1 editor that represents a 1-1 relationship. See the benchmark domain model in Figure 44 on page 183.

The screenshot shows a window titled "Article" with a menu bar (Object, Edit, Info, Window, Help). The window is divided into several sections:

- Object Editors:** HighId (41), LowId (105978951), DtAdded (04/18/1996 14:35:51.000), DtModified (09/08/1996 19:00:40.000), Created_by (nobody), Modified_by (nobody), Checked_by, Approved_by, Release, Status (dropdown), Ref_id (a), Name (Article1), Quantity (1).
- Relationship Editors:** Product (p1 Product1), Deliverable (<none>), Drawing (<none>), Owner (<none>). Each has a "select" button and an "Edit" button.
- RelationshipN Editors:** "Owns" section with a list of items (a1 Article a1, a2 Article a2, a3 Article a3) and buttons for Create, Add, Edit, Remove, and Delete.
- Relationship1 Editors:** "IncludedIn" and "Include" sections, each with a list box and buttons for Create, Add, Edit, Remove, and Delete.

At the bottom, there are buttons for "Select", "Store", and "Cancel".

FIGURE 45. Window owning one object editor, 13 attribute editors, 4 relationship1 editors and 3 relationshipN editors.

Table 39 shows some statistics for a relationship editor implementation class and Table 40 the basic operations to log for a relationship1 editor.

Table 39: *Relationship Editor* implementation class attributes.

| Name | Description |
|------------------|---|
| <i>className</i> | Class name for the attribute editor. |
| <i>uiName</i> | Unique name for the attribute editor within a object editor configuration. Used for logging purposes. |

Table 39: *Relationship Editor* implementation class attributes.

| Name | Description |
|------------------------------|---|
| <i>avgTimeRead</i> | Average read time in milliseconds. If no candidate key attribute value is presented in the relationship editor, the user may have to issue a command to bring up a popup-menu or subform that displays an identifiable value that can be read. It may sometimes be useful to trade read access time for screen space. |
| <i>avgTimeCreate</i> | Average time to issue a create command plus average system response and refresh time in milliseconds. |
| <i>avgTimeAdd</i> | Average time to issue add command plus the average system response and refresh time in milliseconds. |
| <i>avgTimeRemove</i> | Average time to issue remove command plus average system response and refresh time in milliseconds. |
| <i>avgTimeDelete</i> | Average time to issue delete command plus average system response and refresh time in milliseconds. |
| <i>avgTimeRefresh</i> | Average system refresh time after a change in the represented relational e-constellation or the underlying object within the product model. |
| <i>avgInformationDensity</i> | A measure of information quantity or e-messages per 100 square pixels. |
| <i>ratio</i> | Defines the maximum error in the statistical estimates above, $ratio = dx/x$, where dx is the maximum error in x . |
| <i>targetSystemRange</i> | An integer that specifies an interval in which other target system implementations may have their average timings. The interval is between $[x/tsr, x*tsr]$, where $tsr = targetSystemRange$. |

Table 40: Basic operations to log for relationship1 editors.

| operation kbo in relationship newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|---|
| kbRead (iqv,iqr) oid readOid - | Form 2D | - | Logged during read operations. Generated when the user actively issues a read command that displays further information about the related object. |

| operation kbo in relationship newValue oldValue | UI type | U T | Operation description |
|--|------------|---------|---|
| kbEdit oid editOid - | Form 2D | - | Provide a default editor for the related object. |
| kbCreate (iqv,iqc) oid createdOid - | Form 2D | B- C | Create a new related object and add it to the relationship, if allowed by the cardinality constraints. |
| kbAdd (iqv,iqc) oid addedOid - | Form 2D | B- C | Add the object in the current selection to the relationship, if possible. |
| kbRemove (iqd) oid - removedOid | Form 2D | B- C | Remove an object from the relationship, if there is one. |
| kbDelete (iqd) oid - deletedOid | Form 2D | B- C | Remove the related object from the relationship and delete it, if there is one. |
| uiRefresh (iqv,iqr) oid refreshOid - | Form 2D | - | Explicitly update the relationship editor with fresh data read from the database. |
| setSelectionOwner oid selectionOwnerOid - | Form 2D | - | Clear any previous selections, and make the object editor the single owner of the selection. |
| addSelected oid addedOid - | Form 2D | - | Add the object on the other side of the relationship to the selection held by the selection manager, if there is one. |
| removeSelected oid - removedOid | Form 2D | - | Remove the object from the selection held by the selection manager. |

| operation kbo in relationship newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|---|
| clearSelections oid - - | Form 2D | - | Clear all selections held by the selection manager. |
| kbCopy (iqr) oid copyOid originalOid | Form 2D | B C | Copy the object referred by the relationship editor to the clipboard. |
| kbPaste (iqv,iqc) oid newOid copyOid | Form 2D | B C | Add the copy in the clipboard to the relationship, if possible. |

C.7 RelationshipNEditor

A relationshipN editor is used for accessing the 1 side of 1-N Relationships and both sides of M-N Relationships. A relationshipN editor has a one-to-one mapping to a set of relational e-constellations represented by the same e-message type. A relationshipN editor implementation class has the same attributes as a relationship1 editor implementation class (see Table 39).

Table 41 shows a method for estimating average search time.

Table 41: *RelationshipN Editor* implementation class methods.

| Name | Description |
|---|---|
| <i>avgTimeSearch</i> (avgCardinality) | Average time to search for a special e-message in milliseconds, given that the user will recognize a supplied reference message when it is visually located, and the average number of e-messages displayed are avgCardinality. |
| <i>avgVisibleRatio</i> (avgCardinality) | Average percentage of e-constellations in the relationship that are visible within the relationship editor, given the average cardinality of N. |

Table 42: Basic operations to log for relationshipN editors.

| operation kbo in relationship newValue oldValue | UI type | U T | Operation description |
|--|------------|---------|--|
| kbRead (iqv,iqr) oid - - | Form 2D | - | Logged during read operations. Generated when the user scrolls a list box, or actively issues a read command that displays further information about the related object. |
| kbEdit oid - - | Form 2D | - | Provide an editor for the selected related object. |
| kbCreate (iqv,iqc) oid createdOid - | Form 2D | B- C | Create a new related object and add it to the relationship. |
| kbAdd (iqv,iqc) oid addedOid - | Form 2D | B- C | Add the object in the current selection to the relationship, if possible. |
| kbRemove (iqd) oid - removedOid | Form 2D | B- C | Remove a selected object from the relationship. |
| kbDelete (iqd) oid - deletedOid | Form 2D | B- C | Remove the selected object from the relationship and delete it. |
| uiRefresh (iqv,iqr) oid - - | Form 2D | - | Explicitly update the relationship editor with fresh data read from the database. |
| setSelectionOwner oid selectionOwnerOid - | Form 2D | - | Clear any previous selections, and make the object editor the single owner of the selection. |
| addSelected oid addedOid - | Form 2D | - | Add the object to the selection held by the selection manager. |

| operation kbo in relationship newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|--|
| removeSelected oid - removedOid | Form 2D | - | Remove the object from the selection held by the selection manager. |
| clearSelections oid - - | Form 2D | - | Clear all selections held by the selection manager. |
| kbCopy (iqr) oid copyOid/oidSet originalOid/oidSet | Form 2D | B C | Copy selected objects in the relationship editor to the clipboard. |
| kbPaste (iqv,iqc,iqd) oid pasteOid/oidSet copyOid/oidSet | Form 2D | B C | Add the copies in the clipboard to the relationship if possible. |
| uiExpandN (iqv,iqr) oid - - | 2D | - | Expand the contents of the relationship into the 2D view by adding object-editors for all its related objects. |
| uiShrinkN(iqv,iqr) oid - - | 2D | - | Shrink the contents of the relationship from the 2D-view by removing all object-editors for its related objects. |
| uiExpand1 (iqv,iqr) oid addedOid - | 2D | - | Expand the contents of the relationship into the 2D view by adding an object-editor for the object with oid. |
| uiShrink1 (iqv,iqr) oid - removedOid | 2D | - | Shrink the contents of the relationship from the 2D-view by removing the object-editor for oid. |

C.8 Relationship and object link editors

Link editors were introduced in section 18.3.1 on page 156. A link editor is drawn as a graphical link between to object-editors. There are two types of link editors. Relationship link editors represents a **rec** $\langle\langle o_1, o_2 \rangle, r \rangle$. Object link editors represent an information rich relationship between two objects

which is stored in its own *kbo*.

In some cases it is useful to have the positions of object editors and link editors explicitly stored in the knowledge base. This is, for instance, the case when storing structural views of product models, such as the one presented in Figure 47 on page 224. In this case, a *kbo* that stores the graphical information for a link editor are called link *kbo*.

Table 43 shows the basic operations to log for relationship link editors. Note that the selection mechanism behaves a little differently from other *uio*'s which have a direct one-to-one mapping to one *kbo*. A relationship link editor for a **rec** always represents both its objects, and the selection manager returns both these objects when the *le* is in the current selection.

A link editor is created with commands in its view.

Table 43: Basic operations to log for relationship link editors.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| uiChange(iqv,iqc,iqd) oid newRepresentation oldRepresentation | 2D | B C | Change the graphical representation of the link editor. This changes the layout of the view, and perhaps some data stored in the link- <i>kbo</i> . |
| kbRemove (iqv,iqd) oid1 - oid2 | 2D | B C | Remove the relationship between its two objects <oid1,oid2> from the database and its link editor from the user interface, and any associated link- <i>kbo</i> . |
| uiRefresh (iqv,iqr) oid1 - oid2 | 2D | ? R | Update label and layout of the link editor with fresh data read from the database. |
| uiRemove oid1 - oid2 | 2D | - | Remove a link editor from the window. |
| setSelectionOwner oid1 oid2 - | 2D | - | Clear any previous selections, and make the link editor the single owner of the selection. |
| addSelected oid1 oid2 - | 2D | - | Add the link object to the selection held by the selection manager. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| removeSelected oid1 - oid2 | 2D | - | Remove the link object from the selection held by the selection manager. |
| clearSelections oid1 - oid2 | 2D | - | Clear all selections held by the selection manager. |

Table 44: Basic operations to log for object link editors.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|--|------------|--------|--|
| kbRead (iqv,iqr) oid - - | 2D | - | Some link editors may only indicate the existence of a link object. In those cases an explicit read can display information in a temporary popup window. |
| kbCreate (iqv,iqc) newOid - - | 2D | B C | Create a new link kbo, and add a link editor for it to the window. |
| uiChange(iqv,iqc,iqd) oid newRepresentation oldRepresentation | 2D | B C | Change the graphical representation of the link editor. This changes the layout of the view, and maybe some data stored in the link- <i>kbo</i> . |
| kbEdit oid - - | 2D | - | Provide a default editor for the represented <i>kbo</i> . |
| kbDelete (iqv,iqd) oid - - | 2D | B C | Delete the <i>kbo</i> from the database and its link editor from the user interface. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|--|
| uiRefresh (iqv,iqr) oid - - | 2D | ? R | Update label and layout of the link editor with fresh data read from the database. |
| uiAdd (iqv,iqr) addedOid - - | 2D | - | Add a new link editor to the window for a selected link object. |
| uiSet (iqv,iqr) newOid newOid oldOid | 2D | - | Set the link editor to reflect the link object with oid, if possible. |
| uiRemove oid - - | 2D | - | Remove a link editor from the window. |
| setSelectionOwner oid - - | 2D | - | Clear any previous selections, and make the link editor the single owner of the selection. |
| addSelected oid - - | 2D | - | Add the link object to the selection held by the selection manager. |
| removeSelected oid - - | 2D | - | Remove the link object from the selection held by the selection manager. |
| clearSelections oid - - | 2D | - | Clear all selections held by the selection manager. |
| kbCopy (iqr) originalOid copyOid - | 2D | B C | Copy the link object represented by the link editor to the clipboard. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|---|
| kbPaste (iqv,iqc) oid pasteOid copyOid | 2D | B C | Paste the contents of the clipboard to the link object represented by the object editor, if possible. |

C.9 Views

Views were introduced in section 18.3.3 on page 156. They are a special kind of window that allow object editors and link editors to be created dynamically.

Table 45 shows special operations to log for views.

Table 45: Basic operations to log for views.

| operation kbo newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|--|
| uiZoom (iqv,iqr) oid newViewRect oldViewRect | 2D | - | Zoom, pan or scroll within the view. |
| kbCreate (iqv,iqc) newOid newUioEditorRect - | 2D | B C | Create a new <i>kbo</i> , or <i>rec</i> , based on objects in the current selection, and add an object- or link editor for it to the view. |
| uiAdd (iqv,iqr) addedOid newObjectEditorRect - | 2D | B C | Add a new object or link editor to the view for the kbo or kbo's in the selection. |
| clearSelections oid - - | 2D | - | Clear all selections held by the selection manager. |
| kbCopy (iqr) originalOid copyOid - | 2D | B C | Copy all selected objects within the view to the clipboard. |

| operation kbo newValue oldValue | UI type | U T | Operation description |
|---|------------|--------|---|
| kbPaste (iqv,iqc) oid pasteOid copyOid | 2D | B C | Paste the contents of the clipboard to the view, if possible. |

Appendix D Information Processing Model of a User

In [Card et al. 83] the authors introduce an information processing model of a human user⁶⁸. The model is a comprehensive synthesis of human characteristics derived from a large set of experimentally verified data. This model can be used for theoretical evaluation of different user interface designs. This appendix will give a short description of its essence. The interested reader is encouraged to read the second chapter of [Card et al. 83].

The Model Human Processor (MHP) consists of three processors: the perceptual, the cognitive and the motor processor. They interact with each other through the visual image store, auditory image store and working memory (WM). WM consists of activated chunks in long term memory (LTM). Sensory information flows into WM through the perceptual processor. The basic principle of operation is the recognize-act cycle of the cognitive processor (Principle P0 described below). The motor processor is set in motion through activation of chunks in WM. Each of the processors has cycle times τ of about 100 msec. The visual image store has a half-life of about 200 milliseconds and can store an image of a size of about 15 letters. Working memory can hold about seven chunks, and has a half-life depending on the amount of chunks currently activated. Experiments show half-lives of 70 seconds for a single chunk and about 7 seconds when three chunks are activated simultaneously.

The following ten principles describe the operation of the Model Human Processor (from [Card et al. 83] p 27):

- P0. Recognize-Act Cycle of the Cognitive Processor.** On each cycle of the Cognitive Processor, the contents of Working Memory initiate actions associatively linked to them in Long-Term Memory. These actions in turn modify the contents of Working Memory.
- P1. Variable Perceptual Processor Rate Principle.** The perceptual processor cycle time τ_p varies inversely with stimulus intensity.

⁶⁸. The model is a rather technical one and is intended to provide a handle on the “intangible human” for mathematical and other scientific analysis of user interfaces. Care should be taken when generalizing results that are derived from these kinds of models. Since humans are so immensely complicated naturally only limited sets of human characteristics can be taken into account using such a model.

- P2. Encoding Specificity Principle.** Specific encoding operations performed on what is perceived determine what is stored, and what is stored determines what retrieval cues are effective in providing access to what is stored.
- P3. Discrimination Principle.** The difficulty of memory retrieval is determined by the candidates that exist in memory, relative to the retrieval clues.
- P4. Variable Cognitive Processor Rate Principle.** The Cognitive Processor cycle time τ_c is shorter when greater effort is induced by increased task demands or information loads. It also diminishes with practice.
- P5. Fitt's Law.** The time T_{pos} to move the hand to a target of size S which lies a distance D away is given by:

$$(EQ\ 39) \quad T_{pos} = I_M \log_2(D/S + 0,5)$$

where $I_M = 100[70\sim 120]$ msec/bit.

- P6. Power Law of Practice.** The time T_n to perform a task on the n th trial follows a power law:

$$(EQ\ 40) \quad T_n = T_1 n^{-a}$$

where $a = 0.4 [0.2\sim 0.6]$.

- P7. Uncertainty Principle.** Decision time T increases with uncertainty about the judgement or decision to be made:

$$(EQ\ 41) \quad T = I_c H$$

where H is the information-theoretic entropy of the decision and $I_c = 150 [0\sim 157]$ msec/bit. For n equally probable alternatives (called Hick's Law),

$$(EQ\ 42) \quad H = \log_2(n + 1)$$

For n alternatives with different probabilities, p_i , of occurrence,

$$(EQ\ 43) \quad H = \sum_i p_i \log_2(1/p_i)$$

P8. Rationality Principle. A person acts so as to attain his goals through rational action, given the structure of the task and his inputs of information and bounded by limitations on his knowledge and processing ability:

$$\text{(EQ 44) } \text{Goals} + \text{Task} + \text{Operators} + \text{Inputs} + \text{Knowledge} + \text{Process-limits} \\ \Rightarrow \text{Behavior}$$

P9. Problem Space Principle. The rational activity in which people engage to solve a problem can be described in terms of (1) a set of states of knowledge, (2) operators for changing one state into another, (3) constraints on applying operators, and (4) control knowledge for deciding which operator to apply next.

Appendix E Model of Human Mental Object Models

This appendix briefly describes some ideas inspired from cognitive psychology that have guided the design of the user interface software architecture. Much of their origin and inspiration come from [Lindsay&Norman 77],[Minsky 75],[Schank 82] and of course [Card et al. 83]. The text below has to be viewed in an “approximate” way. The purpose is to give the reader an idea of a “soft” hypothesis of how human memory works. As mentioned before, the number of features under consideration when modelling human memory structures has to be restricted. It is important to remember that the purpose of this model is to have some guidance for the design of efficient user interfaces for knowledge engineers and experts when they want to build and interact with large knowledge-bases for artificial real systems⁶⁹. Care should be taken not to make any over-generalizations.

E.1 Chunks

The basic primitive of human knowledge structures seems to be the chunk. A chunk is a mental representation of something, and at the same time an organizer for other chunks. A chunk is stored in long term memory (LTM), and is recalled to working memory⁷⁰ (WM) when parts of its contents happen to appear in WM. A chunk can group together about 5 to 9 other chunks in a particular context⁷¹. The need for hierarchical organization is a consequence of human memory being organized in chunks. The organization of knowledge into rules, objects and procedures (or scripts) is also a consequence of the human chunking mechanism.

A good knowledge representation does not violate the 5-9 limit of the number of related chunks in a particular context. Therefore, a good rule should not contain more than 5-9 premises. A good object should not have more than 5-9 attributes, or groups of related attributes. A good procedure should not contain more than 5-9 statements, unless they can be divided into groups where each group of statements can be grasped as one single

69. The concept of real system was defined in 5.1.1 on page 32. By artificial real systems, we mean systems that are entirely designed by humans. Natural real systems from fuzzy domains such as medicine may be very hard to model formally.

70. Often referred to as short term memory (STM) in the literature.

71. This is a coarse simplification, but it will do for describing the general idea.

chunk.

The reason for chunking seems to be the limitations of human working memory. WM can only contain about 5-9 chunks at any one time⁷². If a new chunk is activated, another becomes deactivated, to make “room” for it. The deactivation is not discrete. We talk about a half-life of about 7 seconds [Card et al. 83], p38. If a chunk is not actively put into mental focus again, the chance of it reappearing decreases exponentially with time.

The chunking mechanism operates without the human being aware of it. Observable psychological phenomena with limited WM capacity can be explained by the chunk phenomena. For instance, the reason why humans have problems understanding a long sentence is explained by the fact that it does not fit into the chunk capacity of working memory. The way to overcome this is to try to understand the parts of the sentence by making chunks of them, and then understanding the total sentence by seeing the connections between the parts, and thus make an organizing chunk. The concept of elementary constellations defined in section 12.4 on page 100 is closely related to what can be stored in a single chunk.

E.2 Learning, or storing chunks in long term memory

Learning, or storing chunks in LTM, occurs when different chunks are simultaneously activated in WM. During the activation an association is enforced between the chunks. This association allows other chunks to be recalled (i.e. reactivated) when one of its associates becomes activated later. Activation time, uniqueness and strength of emotional state seem to have a strong impact on the “permanency” of a chunk in LTM. Interest, for instance, is an emotion that helps the storing process.

E.3 Recalling of chunks

Sensory stimuli have a strong influence on the recall of chunks. Hearing a word or a voice, or seeing a picture or a written symbol immediately allows us to recall our “chunked” experience associated with the sensory stimuli. This is the famous “symbolic” mechanism that is one of basic mechanisms contributing to human intelligence.

The process of reading is an example of a continuous stream of visual stimuli that recalls (activates) previously stored chunks. The order that words are written in influences the pattern of chunk-activation. The chunks interfere with each other, producing an integrated mental picture, which becomes the readers interpretation of the text. Active thinking is another way of recalling chunks. The current mental state, i.e. the currently activated chunks in WM try to activate other chunks in LTM which they have previously been associated with.

72. Psychological experiments show that the actual limit of activated chunks in working memory is about three [Card et al. 83] p 39. However there is a floating border between activated chunks and chunks stored in long term memory, so in practice it becomes 5-9.

The process of active thinking and recall requires less effort if it is supported by visual stimuli. Take, for instance, an engineer working on an electronic circuit design. Fetching a piece of information (i.e. receiving and interpreting an e-message⁷³) about a particular circuit design into WM is achieved faster by looking at a drawing than by trying to recall its corresponding chunk from LTM, even if it is there. Which technique is used depends on which one is faster in the current situation. Say for example that all information of how this circuit is connected is represented on a drawing that has been prepared by an engineer himself. If this drawing is in the next room at the particular moment when he needs the particular information, recalling the corresponding chunk from LTM is probably faster. If the drawing was on his desk, a visual fetching of the e-message from the drawing on the desktop would probably be faster.

E.4 The Soft Semantic Network Hypothesis

Human knowledge (of the type that is useful for engineering) is organized as a “soft” semantic network (SSN). A SSN is a specialization of a mental model. The nodes in the SSN are distinct chunks, each representing some concept, and the links are associations between the chunks. In WM, there can only be a limited number of chunks or “nodes” activated at the same time. The reactivation of a chunk (or node in the SSN) is only possible if there are links to it from currently activated chunks. Visual stimuli, occurring during the reception of an e-message through the senses will put sensory activation into VM. This activation pattern enforces a reactivation of its associated chunks in LTM, i.e. provokes a re-presentation.

When there are no external stimuli available that can induce a certain activation pattern into WM associated with a particular chunk, this chunk can only be retrieved from LTM by following a path in the SSN. This is how the process of active remembering works.

Visual or other sensory stimuli can immediately activate any of the nodes in the SSN given that it has been associated with a certain visual key. Words, icons and symbols are, for instance, visual keys. By providing the right (visual) stimuli, any node can be brought into WM instantaneously. The association between a visual key and a node requires a learning effort i.e. learning the graphical syntax for certain semantical concepts.

E.5 Connection between KB, UI and SSN

A knowledge-base is supposed to represent and simulate the knowledge of a human expert. This means being some kind of picture of one (or several) experts’ SSN. The problem of knowledge acquisition is to elicit and express the SSN in some external knowledge representation i.e. a conceptual model. This model is then analysed and a model of the

73. The e-message could be for instance <IC71,<pin7, GND>> which means that pin number 7 of the integrated circuit 71 is to be connected to the ground power supply.

conceptual model (meta-model) is formulated which defines the format of knowledge to be entered into the knowledge-base. Knowledge visualization during the knowledge acquisition process often gives the expert new insights that make him/her reorganize his/her SSN. This is why knowledge acquisition has to be performed in an iterative way.

The development of a meta-model of a domain is a way of giving the expert a language for expressing the contents of his SSN. The language itself will both influence his SSN and be the core for the view others get of the knowledge. As mentioned, experience in knowledge acquisition shows that once a model of the conceptual model is formulated and a user interface built for entering the knowledge into the knowledge-base, the expert preferentially uses it on his own to enter and model the larger bodies of detailed knowledge.

Entering and refining the knowledge of a knowledge-base is a creative iterative process. It can be accelerated if the system places no restrictions on the possibility of visualizing the knowledge and allows the expert to experiment with it. The expert's line of thought can traverse his entire SSN. At any time, he may want to have visual stimuli support to recall the details of any information or knowledge previously entered into the knowledge-base. The user interface should provide him with this support.

E.6 Mapping the SSN to the KB

The user interface should provide the expert with visual keys which he associates with the contents of his own SSN and through which he can reach the information and knowledge entered into the knowledge-base. This means that the graphical syntax for semantic concepts should be developed in close cooperation with the expert.

The organization of knowledge-base views into object- and link editors will give the expert user a visual map over the contents of the knowledge-base, which is a model of his/her own SSN. This visualization and feedback allows him to reason about his own knowledge. The knowledge-base views enable him to acquire an efficient access structure over his own expertise. This access structures can later be internalized and used during mental reasoning.

E.7 Reservation against the concept of SSN

The model of the expert's knowledge structures as an SSN is a metaphor for explaining ideas. Chunks are not implemented in the form of some kind of distinct nodes, but rather as a spatio-temporal activation pattern in the neurons of the brain. A chunk has no explicit links to other chunks, but its activation pattern puts a kind of statistical gravitation on other chunks' activation patterns, depending on how the chunks have been activated in WM before. If different chunks are activated in WM at the same time, each of their statistical gravitation fields will interact and enforce or repel the

activation of other chunks. If a set of chunks stay in WM long enough, they will form a new chunk which inherits its constituents' gravitational fields.

E.8 Conclusions

The conclusion of the hypothesis of human memory described above is that the user's mental activities should be supported by the graphical user interface. The objects and relations in the knowledge-base and mental models should be visualized. The pictorial structure of the views are worthwhile remembering in themselves, since they will become organizing chunks for their constituents. Such "picture" chunks can, as mentioned, be incorporated into the user's mental model of the knowledge-base and used in his mental reasoning processes.

Access to information and knowledge in the knowledge-base should be fast because of the half-life of WM.

Appendix F User Interaction Example

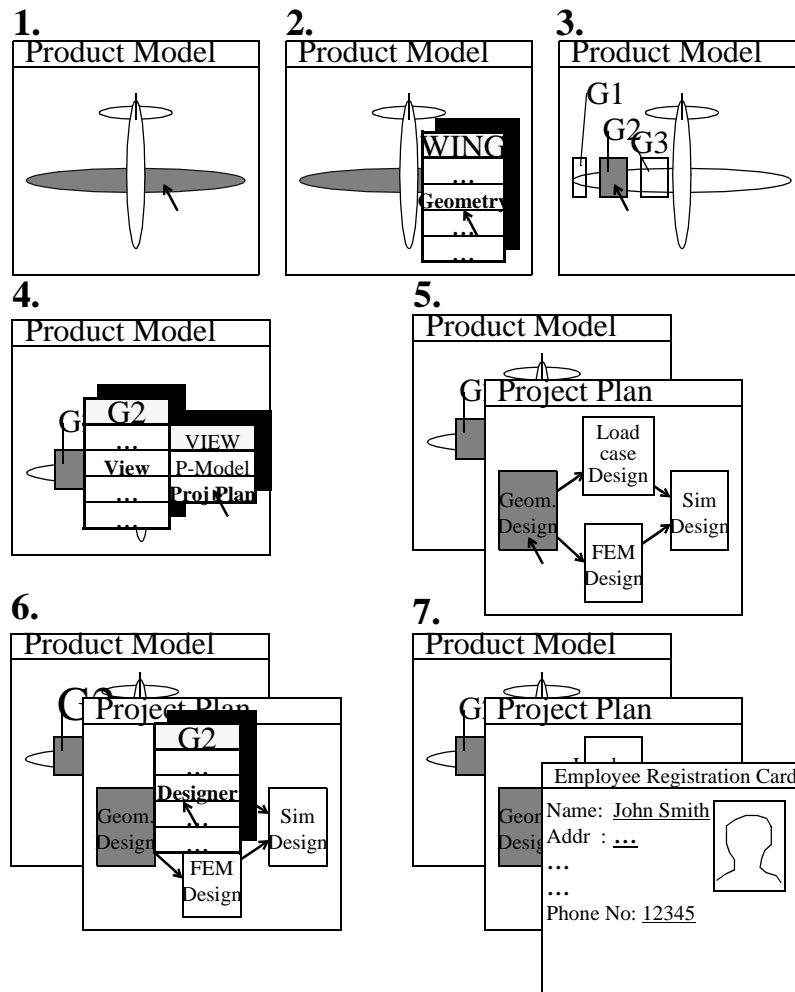


FIGURE 46. Example that illustrates walking an associative access path over relational e-constellations to find information in an object-oriented knowledge-base. The user wants to find out who the designer of the geometry of a certain aircraft wing part is i.e. find the e-constellation $\langle G2, \langle \text{designer}, \text{"John Smith"} \rangle \rangle$. The initial view v_1 on the screen displays parts of a structural model (product model) of the aircraft. The access path passes three view instances v of different window types; the product model view, the project plan view and finally the employee registration card view.

Appendix G Sample OO-model of a Car Braking System

This appendix illustrates some concepts introduced in chapter 18.3 on page 156 by showing a small instance of an infological model that represents a structural model of a car braking system. The model is used to illustrate different types of views on the object-oriented model and to give an idea of how to quantify the information content numerically as the number of elementary constellations. (See chapter 12 on page 97 for definitions of the theoretical concepts).

G.1 A structural view (71 e-messages)

Figure 47 shows a stylistic view of the structural model of a car braking system. It utilizes the user's background knowledge about the relative spatial layout of a car for interpretation. The car braking system contains two isolated brake circuits, that each influence three of the car's wheels.

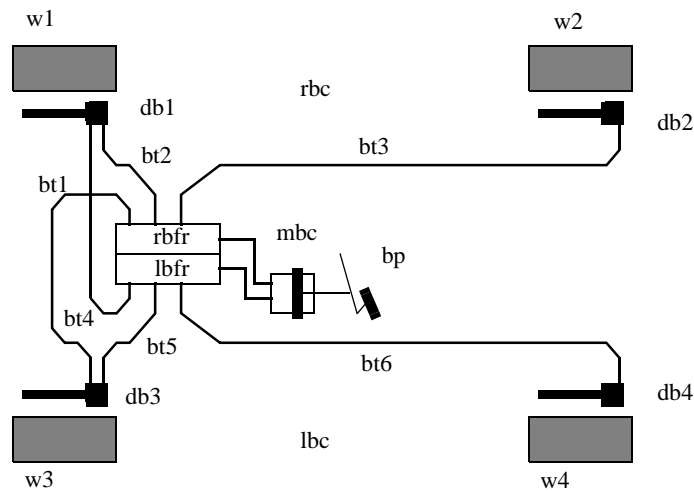


FIGURE 47. A view of a structural model of a car braking system.

The four shaded rectangles (w1..w4) represent the wheels. The disc brakes are named db1 to db4. The brake tubes (bt1-bt6) propagate the pressure from the right- and left brake fluid reservoirs (rbfr, lbfr) to the disc brakes. The two brake circuit reservoirs are fluidly disconnected, but maintained at

a constant pressure by a connection to the main brake cylinder (mbc). mbc is physically connected to the brake pedal (bp). A left and a right brake circuit (rbc,lbc) can be inferred from the structural connections, e.g. rbc consists of db1,db2,db3,bt1,bt2,bt3 and rbfr. If the components of the two brake circuits together with the names rbc and lbc were painted in different colours, for example rbc in red and lbc in green, the knowledge view could be split up into 71 e-constellations, as shown in Fig G.7 on page 228.

The utilization of the users “invisible” background knowledge makes views such as the one in Figure 47 very efficient (i.e. they have a high e-message density on the screen and are easy for the user to remember and recall). A structural view of this kind can obviously not be automatically generated by a graph drawing algorithm. The spatial layout has to be stored explicitly in the model. It is possible to use the user interface software architecture for this purpose.

The drawing algorithms in the view v have to have an explicit mapping function for the object identifiers of instances i to geometric coordinates for their corresponding object editor instances oe . Though the main purpose of the software architecture is not to draw these kinds of views. The structural view in Figure 47 was introduced to give a comprehensive view of the contents of the sample object-oriented knowledge-base.

G.2 A classical knowledge-base browser view

A classical KB-class browser view of the object-oriented knowledge-base could look like the one depicted in Figure 48 on page 226. This view provides fast access to all classes and instances in the KB. The bold links denote subclass relationships and the dotted ones denote instance-of relationships. There are 5 subclass links and 18 instance-of links which are e-messages for relational e-constellations. The view also contains e-messages for the name attribute of each class and instance. There are 6 classes and 18 instances. The view expresses 5 subclass and 18 instance-of relationships, 6 class name attributes and 18 instance name attributes which makes a total of 47 e-messages. Observe the spatial sorting order provided in this view. Class < subclass < instance is the topological ordering in the left to right direction. The object editors on the same level, having the same ancestor, are sorted alphabetically by name. This is an example of sorting criteria that have to be specified in the view-class V that implements the view. If the knowledge engineer has a KB that contains, for example, 200 classes and 1000 instances, the sorting orders in this view allow him to quickly visually look up an object editor for a certain instance.

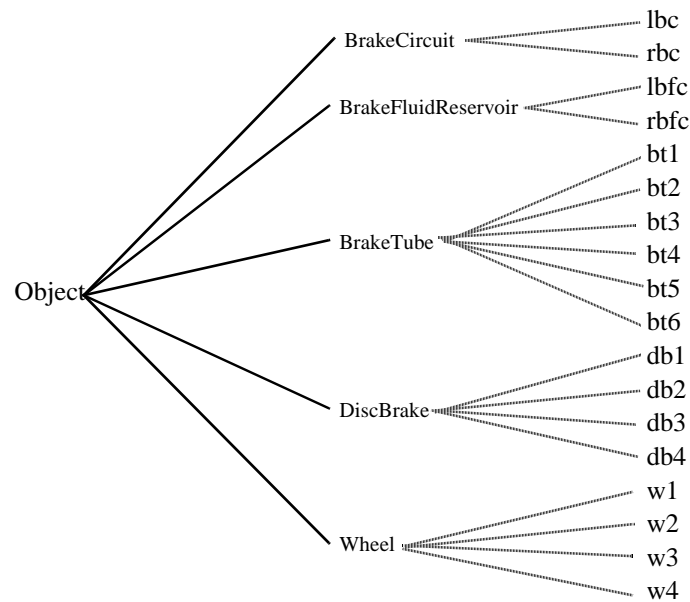


FIGURE 48. A classical KB-browser view that is commonly provided by frame-based expert system shells.

G.3 Topological index views

Topological indexes can be used for coarse visual search. Figure 49 shows an example of a topological index of the KB-view in Figure 48. The benefit of a topological index is the low amount of screen space it uses. Object editors for this kind of view only express the existence of an instance. Further information can quickly be obtained by issuing a *kbRead*-operation on the object editor (See Table 35 on page 199). A simple implementation of this type of topological index view is provided, for example, by KEE [Intellicorp 87] for fast visual navigation through large knowledge-bases.

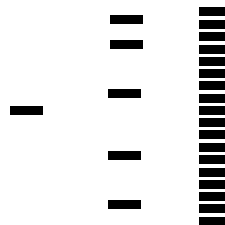


FIGURE 49. A topological index of the view in Figure 48.

G.4 A part-of view (30 e-messages)

Figure 50 shows the part-of relationship for the components in the car braking system. This view is very useful for modeling purposes, but may not be provided by expert system shells. Note that the object editors do not have to be rendered as the names of the different parts. They could be in the form of icons or boxes containing fields with attribute information other than the name. If the topological sorting is strictly defined, the spatial location represents the identifying attribute for each object.



FIGURE 50. A view that shows the part-of relationships in the two brake circuits. Note the alphabetical sorting order on the different levels.

G.5 A simple table view (30 e-messages)

Figure 51 contains a simple table view. These are typically sorted in alphabetical order, which allows a fast visual “binary search” in the vertical direction, to locate the object editor of interest. This table expresses the same 30 elementary constellations as Figure 50.

| Part | Sub part |
|------|----------|
| lbc | bt1 |
| lbc | bt2 |
| lbc | bt3 |
| lbc | db1 |
| lbc | db2 |
| lbc | db3 |
| lbc | rbfr |
| rbc | bt4 |
| rbc | bt5 |
| rbc | bt6 |
| rbc | db1 |
| rbc | db3 |
| rbc | db4 |
| rbc | lbfr |

FIGURE 51. A simple table view that expresses the same part-of relationships information as Figure 50.

G.6 A PERT-diagram view.

PERT stands for program evaluation and review technique [Schaffer et al. 65]. PERT-diagrams are commonly used for visualizing the time requirements and dependencies among tasks during the planning of a project. The PERT-diagram in Figure 52 expresses time on the horizontal

axis, and parallel design activities in the vertical dimension. Note that there is no scaled sorting order in the vertical dimension. The object editors which represent milestones are placed by the algorithm (in a view-class V) to avoid crossing links [Battista et al. 89]. The link editors (e.g. le_1) which denote design activities are annotated with the duration that the activities are expected to take. Possible user interactions on the link editors can pop up a menu and provide more information about the activity. A click on the object editors (e.g. oe_1) might display the exact point in time for that milestone. The elementary message analysis is left for the reader.

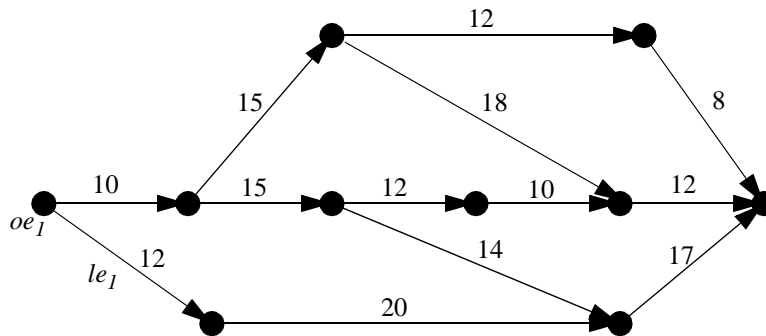


FIGURE 52. A PERT-diagram view.

G.7 An elementary constellation view (71 e-messages)

Below, we have a list of all the elementary constellations contained in the sample structural model in Figure 47 on page 224. There is no particular well-defined sorting order.

Attribute e-constellations:

- 1) $\langle w1, \langle \text{name}, "w1" \rangle \rangle$
- 2) $\langle w2, \langle \text{name}, "w2" \rangle \rangle$
- 3) $\langle w3, \langle \text{name}, "w2" \rangle \rangle$
- 4) $\langle w4, \langle \text{name}, "w4" \rangle \rangle$
- 5) $\langle db1, \langle \text{name}, "db1" \rangle \rangle$
- 6) $\langle db2, \langle \text{name}, "db2" \rangle \rangle$
- 7) $\langle db3, \langle \text{name}, "db3" \rangle \rangle$
- 8) $\langle db4, \langle \text{name}, "db4" \rangle \rangle$
- 9) $\langle bt1, \langle \text{name}, "bt1" \rangle \rangle$
- 10) $\langle bt2, \langle \text{name}, "bt2" \rangle \rangle$
- 11) $\langle bt3, \langle \text{name}, "bt3" \rangle \rangle$
- 12) $\langle bt4, \langle \text{name}, "bt4" \rangle \rangle$
- 13) $\langle bt5, \langle \text{name}, "bt5" \rangle \rangle$
- 14) $\langle bt6, \langle \text{name}, "bt6" \rangle \rangle$
- 15) $\langle rbfc, \langle \text{name}, "rbfc" \rangle \rangle$
- 16) $\langle lbfc, \langle \text{name}, "lbfc" \rangle \rangle$

- 17) <mbc,<name,"mbc">>
- 18) <bp,<name,"bp">>
- 19) <lbc,<name,"lbc">>
- 20) <rbc,<name,"rbc">>

Relational e-constellations :

- 21) <<Wheel,w1>,instance-of>
- 22) <<Wheel,w2>,instance-of>
- 23) <<Wheel,w3>,instance-of>
- 24) <<Wheel,w4>,instance-of>
- 25) <<DiskBrake,db1>,instance-of>
- 26) <<DiskBrake,db2>,instance-of>
- 27) <<DiskBrake,db3>,instance-of>
- 28) <<DiskBrake,db4>,instance-of>
- 29) <<BrakeTube,bt1>,instance-of>
- 30) <<BrakeTube,bt2>,instance-of>
- 31) <<BrakeTube,bt3>,instance-of>
- 32) <<BrakeTube,bt4>,instance-of>
- 33) <<BrakeTube,bt5>,instance-of>
- 34) <<BrakeTube,bt6>,instance-of>
- 35) <<BrakeFluidContainer,rbfc>,instance-of>
- 36) <<BrakeFluidContainer,lbfc>,instance-of>
- 37) <<BrakeCylinder,mbc>,instance-of>
- 38) <<BrakePedal,bp>,instance-of>
- 39) <<rbc,rbfc>,part-of>
- 40) <<rbc,bt1>,part-of>
- 41) <<rbc,bt2>,part-of>
- 42) <<rbc,bt3>,part-of>
- 43) <<rbc,db1>,part-of>
- 44) <<rbc,db2>,part-of>
- 45) <<rbc,db3>,part-of>
- 46) <<lbc,lbfc>,part-of>
- 47) <<lbc,bt4>,part-of>
- 48) <<lbc,bt5>,part-of>
- 49) <<lbc,bt6>,part-of>
- 50) <<lbc,db1>,part-of>
- 51) <<lbc,db3>,part-of>
- 52) <<lbc,db4>,part-of>
- 53) <<rbfc,bt1>,fluid-connected>
- 54) <<rbfc,bt2>,fluid-connected>
- 55) <<rbfc,bt3>,fluid-connected>
- 56) <<bt1,db3>,fluid-connected>
- 57) <<bt2,db1>,fluid-connected>
- 58) <<bt3,db2>,fluid-connected>
- 59) <<lbfc,bt4>,fluid-connected>
- 60) <<lbfc,bt5>,fluid-connected>
- 61) <<lbfc,bt6>,fluid-connected>
- 62) <<bt4,db1>,fluid-connected>
- 63) <<bt5,db3>,fluid-connected>
- 64) <<bt6,db4>,fluid-connected>

```
65) <<bp,mbc>,brake-influence>
66) <<mbc,rbfc>,brake-influence>
67) <<mbc,lbfc>,brake-influence>
68) <<db1,w1>,brake-influence>
69) <<db2,w2>,brake-influence>
70) <<db3,w3>,brake-influence>
71) <<db4,w4>,brake-influence>
```

FIGURE 53. A textual representation of the 71 e-constellations that model the car braking system in Figure 47 on page 224.

G.8 Conclusion

In section 18.2 on page 151 we introduced the user interface software architecture and a notation for different concepts. The notation is intended to allow a mathematical representation for comparing different user interface configurations. Automatic calculations of the number of elementary message built into the *uio*'s might be useful. The number of e-messages expressed by object editors *oe* and link editors *le* of particular implementation classes OE and LE can easily be calculated initially, and stored in methods in OE and LE. A view-instance *v* can then easily count the number of e-messages expressed by summing the e-messages for each of its visible *uio*'s. If we proceed further with such calculations, we could estimate how accessible objects in the knowledge-base are to the knowledge-engineer. Mathematical analysis techniques similar to the ones that are used for estimating the average performance of application programs that work on a data-base with a certain index configuration could be used for tuning the graphical user interface for large knowledge-bases. If the abstract *uio*-implementation classes are augmented with methods for collecting statistics from the user interaction, experimental data can be collected automatically. Such data enables performance evaluation of different user interface designs based on actual measurements.

The possibilities for applying the software architecture for interaction with modern commercial object-oriented databases seem very good [Cattell et al. 96].

The proposed software architecture demonstrates a way to implement user interfaces that allows reduced granularity and increased referability of stored data. By using different knowledge views that work both as graphical knowledge representations and as natural domain related indexes to the contents of the knowledge-base, we get access paths of varied types. The need to invent unique mnemonic names for objects decreases, since they can be identified by their relative locations and referred to by pointing. These properties are consistent with those proposed as future developments in artificial intelligence by Sandewall in [Sandewall 88].

A

access cost 132
 Access time 131
 active expert approach 15
 AECTRole 136
 aggregate 53
 ambiguous references 103
 artificial systems 30
 atomic object 101
 Attribute 117, 151
 attribute 78, 102, 117
 attribute e-constellation 151
 attribute e-constellation type 102
 attribute editor 201
 attribute editor implementation class 151
 attribute editor instance 151
 attribute e-message type 104, 154
 AttributeGroup 78
 AutoLISP 89
 average bandwidth 149
 avgCardinality 114
 avgCardinality1to2 120

B

block 52

C

CAD/DB interface procedures 89
 CAD/UI interface 89
 CAD-DBI interface 67
 CAE-system 37
 car braking system 224
 cascadeDelete2 77
 chunk 215, 218
 Class 77, 114, 151
 class1 77, 119
 class2 77, 119
 classical knowledge-base browser 225
 Client applications 51
 clipboard manager 151, 161
 Cognitive processor 215
 combined cycle power plant 49
 complete elementary message 103
 component 53
 compound object 101

conceptual aspects 18
 conceptual model 30, 33, 99
 conceptual spaces 157
 configurable access control system 66
 constraint-checking 25
 cover-and-differentiate 21

D

data 98
 data acquisition tools 30
 data logical model 32, 99
 database interface 67
 DataDictionary 77
 DBOobject 76
 Declaration 79
 Deep knowledge 18
 deep knowledge components 26
 default information quantity 118
 definition 114
 delivery limit 53
 delivery system 34
 delivery system platform 35
 derived concepts 101
 design instantiation 25
 designation system 57
 design-choices 24
 design-constraints 24
 design-extension 25
 design-problem-solver 24
 design-space 24
 design-state 25
 desirability-function 24
 development environment 34
 display manager 151, 159
 Domain 79
 domain model 19, 33
 DomainModel 77
 drawing algorithms 225
 drawing environment 57
 drawing frame 57

E

e-concept 104
 e-constellation 101
 elementary concept 104
 elementary constellation 97, 101

elementary constellation type 102
 elementary constellation view 228
 elementary message 97
 elementary message type 104
 e-message type 104
 entropy 106, 216
 execution environment 35
 expert systems 12
 explicit reference 103

F

facts 31
 field class 151
 field editor 154
 field editor implementation class 155
 field instance 151
 fix 25
 fixed finite alphabet 105
 frame of reference 98
 fundamental concepts 101

G

generally identifying attribute 102
 Generated source code measurements 90
 generating object relation 103
 generating property 103
 generic task 20
 genSqlFlag 77
 GOMS-Analysis 130
 graph drawing algorithms 157
 GUI-DBI interface 68

H

human visual system 167

I

identifying property 101
 implementation, operation and maintenance 32
 implicit reference 103
 importance weight 133
 increased referability 230
 inference calculus 18, 26
 infological model 99
 information 98, 99, 105
 information processing model of a

human user 215
 information quantity 97, 116, 118, 122, 125
 information sphere 101
 information theory 105
 InfoSet 135, 137
 instrument 53
 intelligent front end 37

K

KEE 42
 Knowledge 11
 knowledge acquisition tools 30
 knowledge entities 30
 Knowledge Level Hypothesis 10
 Knowledge roles 21
 Knowledge-base object 151
 knowledge-based system 34

L

layers 57
 legal design-areas 24
 Link editor 209
 link editor 156
 link editor implementation class 151, 156
 link editor instance 151
 link-instance 156
 local information quantity 116
 local names 103
 long term memory 215

M

meaningful e-concept 104
 meaningful time version of the e-concept 104
 mechanical limit 53
 mental model 30, 32
 message 103
 meta-data model 27
 meta-database 69, 82
 meta-database domain model 76, 82
 meta-model 30, 33
 Microsoft Windows 40
 M-N relationship 124
 model 99
 Model Human Processor 215

-
- model of the conceptual model 30, 33
 - Model-View-Controller 166
 - mustExist1 120
 - mustExist2 120
 - N**
 - name 114
 - name1to2 77, 119, 120
 - name2to1 77, 119, 120
 - notation 151
 - O**
 - Object 76
 - object 61, 101
 - object component 103
 - object editor 153
 - object editor implementation class 151, 154
 - object editor instance 151
 - object editor type 151, 153, 198
 - object group 102
 - Object identifier 151
 - object identifier 101
 - object relation 101
 - object system part 101
 - Object types 102
 - object-oriented domain model 61, 62
 - Object-Oriented Domain Model Example 72
 - OOCASE 69
 - operability 134
 - operation cost 134
 - operation importance weight 134
 - owns1to2 119, 120
 - P**
 - P&ID application 55
 - part-of relationship 119
 - part-of view 227
 - P-basis 101
 - PC prototype 40
 - perceptual processor 215
 - performance system 34
 - PERT-diagram 227
 - P-generator 101
 - Pipe 53
 - plan-based design 26
 - plant 52
 - plant browser 53
 - PMDB triggers 89
 - potential x- reference 103
 - pragmatic aspects 18
 - predicate component 103
 - prefix 77, 114
 - Principle of rationality 11
 - probabilityClear1to2 120
 - problem solving method 21
 - problem-solving component 17
 - ProCAD 49
 - ProCAD domain model 51
 - Process and Instrumentation Diagrams 51
 - product modeling system 1, 61
 - project specific attribute 54
 - project specific field 54
 - Properties are generative 101
 - Property 78
 - property 101
 - property type 101
 - propose-and- revise 21
 - propose-and-revise 26
 - prototype knowledge-base 33
 - Q**
 - quasi-names 103
 - R**
 - R-basis 101
 - real system 32
 - real world system 30
 - Reality 99
 - recognize-act cycle 215
 - RECTRole 136
 - reference expression 103
 - reference r 103
 - reference relationship 119
 - relational e- message type 104, 154
 - Relational e-constellation 151
 - relational e-constellation type 102
 - relational type 101
 - Relationship 77, 119, 151
 - relationship 119

- Relationship operations 121
- relationship1 editor 203
- relationship1 editor implementation class 151
- relationship1 editor instance 151
- relationshipN editor 207
- relationshipN editor implementation class 151
- relationshipN editor instance 151
- remt 154
- representational component 17
- R-generator 101

S

- Second Generation Expert Systems 17
- section 52
- selection manager 151, 160
- semantic network 220
- simple table view 227
- Smalltalk PMDB 91
- Smalltalk User Interface 91
- Source Code Generation Example 74
- space to be searched 24
- stdCardinality 114
- stdCardinality1to2 120
- stored procedures 89
- surface knowledge 18
- system 52
- system architecture of ProCAD 50

T

- target 103
- target domain 30
- target system 34
- Task 136
- task 135
- ternary relationship 126
- test cases 41
- time 101
- topological index 226
- Topological index views 226
- transaction manager 151, 161
- transformable meta-model 41
- traveling path 133
- TypeDef 79

U

- unit of information 106
- universal names 103
- update access rights 54
- user access rights 51
- user interface configuration 131
- user interface object 151, 153
- user interface software architecture 3
- user role 135
- UserRole 135, 136
- UserTransaction 136

V

- valid e-constellation type 102
- valid infological context 103
- valid time version of the e-constellation type 102
- values 102
- View 213
- view 156
- view implementation class 151
- view instance 151
- view-class 157
- visual key 220

W

- window 194
- window implementation class 151
- Window instance 151
- Window type 151
- window type 195
- working memory 215

Numbers

4GL - Fourth Generation Language. An integrated application development environment including database, (macro) programming language, form painter and report generator.(p. 50).

A

aect - Attribute e-constellation type.(p. 102).

aemt - Attribute e-message type.(p. 104)., (p. 154).

C

CAD - Computer Aided Design. (p. 1).

CAE - Computer Aided Engineering. (p. 1).

CAM - Computer Aided Manufacturing. (p. 64).

CASE - Computer Aided Software Engineering. (p. 2).

CAX - Common abbreviation for CAD, CAE or CAM.(p. 65).

CORBA - Common Object Request Broker Architecture. Standard for distributed object-oriented application communication developed by Object Management Group (OMG) (p. 109).

D

DBI - Database Interface. 3GL based library of functions for client communication with a database. (p. 67).

DBMS - Database management system. (p. 50).

E

EER - Extended Entity Relationship-model. An Entity-Relationship meta-model language extended with constructs for describing, for instance, inheritance. (p. 3).

G

GUI - Graphical User Interface. (p. 68).

I

IDL - Interface Definition Language, used to describe the interfaces that client objects call and object implementations provide. (p. 79).

IFE - Intelligent Front End. A software that provides an engineer with a nice graphical front end to, for instance, a command language based CAE-program. (p. 37).

K

KBS - Knowledge Based System.(p. 9).

KKS - Kraftwerks Kennzeichen System(p. 57).

L

LTM - Long Term Memory.(p. 215).

M

MHP - Model Human Processor.(p. 215).

O

oid - Object identifier. (p. 101).

OODM - Object-oriented domain model. (p. 61).

OPR - Object Property Relationship. (p. 3).

P

P&ID - Process and instrumentation diagram. (p. 51).

PERT - Program Evaluation and Review Technique. (p. 227).

PFBC - Pressurised Fluid Bed Combustion. (p. 49).

PMDB - Product Model Database. (p. 58).

PMS - Product Modeling System. (p. 1).

R

rec - Relational e-constellation. (p. 209).

rect - Relational e-constellation type. (p. 102).

remt - Relational e-message type. (p. 104).

S

SSN - Soft Semantic Network. A special type of mental model. (p. 220).

U

UISA - User Interface Software Architecture. (p. 3).

W

WM - Working Memory. (p. 215).

| | | |
|-----------|---|-----|
| Table 1: | Generated stored procedures for object handling. | 74 |
| Table 2: | Generated stored procedures for relationship handling. | 74 |
| Table 3: | Domain model measurements. | 88 |
| Table 4: | Source code generator measurements. | 89 |
| Table 5: | Generated source code measurements Benchmark (ProCAD). | 90 |
| Table 6: | Generated source code size / source code generator size. | 90 |
| Table 7: | Class attributes. | 114 |
| Table 8: | Class operations. | 114 |
| Table 9: | Class instance operations. | 115 |
| Table 10: | <i>Attribute</i> attributes. | 117 |
| Table 11: | <i>Attribute</i> operations on class C. | 118 |
| Table 12: | <i>Attribute</i> operations on objects of class C. | 118 |
| Table 13: | <i>Relationship</i> attributes. | 120 |
| Table 14: | 1-1 <i>Relationship</i> operations in both directions. | 121 |
| Table 15: | 1-N <i>Relationship</i> operations in both directions. | 124 |
| Table 16: | <i>UserRole</i> attributes. | 136 |
| Table 17: | <i>Task</i> attributes. | 136 |
| Table 18: | <i>UserTransaction</i> attributes. | 136 |
| Table 19: | <i>InfoSet</i> attributes. | 137 |
| Table 20: | <i>ECTRole</i> attributes. | 137 |
| Table 21: | <i>AECTRole</i> attributes. | 138 |
| Table 22: | <i>RECTRole</i> attributes. | 139 |
| Table 23: | Basic operations on <i>uios</i> that involve the selection manager. | 160 |
| Table 24: | Some useful <i>Selection Manager</i> methods. | 161 |
| Table 25: | List of documentation of the benchmark domain model. | 182 |
| Table 26: | Classes from the benchmark domain model. | 184 |
| Table 27: | Relationships from the benchmark domain model. | 185 |
| Table 28: | Attributes of the benchmark domain model. | 186 |
| Table 29: | Recorded log record information for a user interface operation. | 192 |
| Table 30: | Special operations generated by the transaction manager. | 193 |
| Table 31: | <i>Window type</i> attributes. | 195 |
| Table 32: | <i>Window</i> visualization operations. | 195 |
| Table 33: | Basic operations to log for windows. | 196 |
| Table 34: | <i>Object Editor type</i> attributes. | 198 |
| Table 35: | Basic operations to log for object editors. | 199 |
| Table 36: | <i>Attribute Editor</i> attributes. | 201 |
| Table 37: | <i>Attribute Editor</i> methods. | 201 |
| Table 38: | Basic operations to log for attribute editors. | 202 |
| Table 39: | <i>Relationship Editor</i> implementation class attributes. | 204 |
| Table 40: | Basic operations to log for relationshipl editors. | 205 |
| Table 41: | <i>RelationshipN Editor</i> implementation class methods. | 207 |
| Table 42: | Basic operations to log for relationshipN editors. | 208 |
| Table 43: | Basic operations to log for relationship link editors. | 210 |
| Table 44: | Basic operations to log for object link editors. | 211 |
| Table 45: | Basic operations to log for views. | 213 |

| | | |
|-----|------------|--|
| 2 | FIGURE 1. | Software development approach for product modeling systems. |
| 31 | FIGURE 2. | The process of developing a knowledge-based system. |
| 43 | FIGURE 3. | A user-interface view of a knowledge-base instance of the class <code>CRACK_GROWTH_GEOMETRY_MODEL</code> . |
| 49 | FIGURE 4. | Photo of a steam turbine power plant from ABB STAL. |
| 51 | FIGURE 5. | Architecture of the ProCAD product modeling system for power plant system design. |
| 52 | FIGURE 6. | Outline of the domain model for ProCAD. |
| 54 | FIGURE 7. | The plant browser. |
| 55 | FIGURE 8. | P&ID CAD-application editing a turbine system drawing. |
| 56 | FIGURE 9. | A zoom-in of a P&ID drawing. |
| 56 | FIGURE 10. | A form for editing data for an aggregate in the product model database. |
| 57 | FIGURE 11. | Menu providing 20 of the more than 300 symbols in ProCAD. |
| 58 | FIGURE 12. | Graphical database reports directly on the drawing. |
| 62 | FIGURE 13. | Approach to overcome the three obstacles to PMS development. |
| 65 | FIGURE 14. | Sample product modeling system. |
| 67 | FIGURE 15. | Software layers in the product model database. |
| 68 | FIGURE 16. | Software layers in the CAD-client. |
| 69 | FIGURE 17. | Software layers in a browser application. |
| 70 | FIGURE 18. | Architecture of the PMS development system. |
| 73 | FIGURE 19. | Simple PMS domain model example to describe the graphical syntax of our object model diagrams. |
| 75 | FIGURE 20. | Example of an automatically generated stored procedure. |
| 82 | FIGURE 21. | An object model diagram for the meta-database. |
| 84 | FIGURE 22. | A subset of the EER-model for the meta-database |
| 85 | FIGURE 23. | Example of simple generated source code from the benchmark domain model presented in chapter 13. |
| 85 | FIGURE 24. | Source code generator for the code in Figure 23. |
| 86 | FIGURE 25. | Principle of declarative 1-step source code generation. |
| 86 | FIGURE 26. | Source code generated by a 2-step source code generator. |
| 87 | FIGURE 27. | Principle of declarative 2-step source code generation. |
| 87 | FIGURE 28. | Small portion of the generated SQL-code generator. |
| 88 | FIGURE 29. | Source code generator for the code in Figure 26. |
| 98 | FIGURE 30. | Transformation of data into information. |
| 100 | FIGURE 31. | The homomorphic mapping of a slice of reality into a database model. |
| 108 | FIGURE 32. | The benchmark domain model. |
| 110 | FIGURE 33. | Example of an object editor window for instances of the class <code>Company</code> within the benchmark domain model. |
| 110 | FIGURE 34. | Example of a form-based browser window for the path <code>Company</code> down to <code>Article</code> in the benchmark domain model. |
| 111 | FIGURE 35. | Example of a form-based browser window for the |

-
- hierarchical structure Article->owner_owns.
- 111 FIGURE 36. Example of a 2D browser for hierarchical 1-N relationships.
- 112 FIGURE 37. Interactive calculation of the information quantity owned by an Article object. In this view 40 e-messages are visible that represent e-constellations owned by "a3 Article a3".
- 113 FIGURE 38. View of the meta-database domain model, showing domain model meta-data used for performance calculations.
- 135 FIGURE 39. Object model diagram of the task description language.
- 143 FIGURE 40. Responsibility areas for the different user roles in the benchmark domain model.
- 150 FIGURE 41. A systems and subsystems model of the user-user interface - knowledge-base interaction.
- 152 FIGURE 42. Object model diagram showing relationships between different user interface objects *uio* in the domain model of the user interface state.
- 155 FIGURE 43. Information entities in a form-based object editor.
 $\underline{aemt} = \langle r(\text{Company}), r(\text{Company.name}) \rangle$
 $\underline{remt} = \langle \langle r(\text{Company}), r(\text{Drawing}) \rangle, r(\text{company_drawings}) \rangle$
- 183 FIGURE 44. The benchmark domain model.
- 204 FIGURE 45. Window owning one object editor, 13 attribute editors, 4 relationship1 editors and 3 relationshipN editors.
- 223 FIGURE 46. Example that illustrates walking an associative access path over relational e-constellations to find information in an object-oriented knowledge-base.
- 224 FIGURE 47. A view of a structural model of a car braking system.
- 226 FIGURE 48. A classical KB-browser view that is commonly provided by frame-based expert system shells.
- 226 FIGURE 49. A topological index of the view in Figure 48.
- 227 FIGURE 50. A view that shows the part-of relationships in the two brake circuits. Note the alphabetical sorting order on the different levels.
- 227 FIGURE 51. A simple table view that expresses the same part-of relationships information as Figure 50.
- 228 FIGURE 52. A PERT-diagram view.
- 230 FIGURE 53. A textual representation of the 71 e-constellations that model the car braking system in Figure 47 on page 224.