

OOCASE User Manual

Version 4.0.4 2018-12-11

Abstract

Welcome to the OOCase User Manual!

The purpose of this manual is to deliver a core conceptual understanding of some of the fundamentals in efficient practical information management of technical information whose life times span decades.

These core concepts are high-level and hard to acquire without concrete examples. Thus OOCase which is a fully commercially competitive product within its scope of applicability, is used as a teaching tool to transfer this higher-level of understanding.

Intended readers are programming experienced life-time-students of any age that have severe challenges to master, and need the best education possible to implement what is necessary to gain the long-lasting quality-of-life producing effects into foundational physical infrastructure that we all need.

Preface.....	8
Chapter 1 The Purpose of OOCase.....	9
Chapter 2 Data and Information	17
Chapter 3 The Role of OOCase in the Software Development Process.....	19
Chapter 4 The Meta Model of OOCase	22
Chapter 5 Domain Model Development.....	29
Chapter 6 Source Code Generation.....	33
Chapter 7 Prototype Iteration.....	36
Chapter 8 Configuration Files	37
Chapter 9 Import of Domain Models	39
Chapter 10 Export of Domain Models	52
Chapter 11 Quality Assurance, Version and Release Management.....	55
Chapter 12 Quality Assurance Functionality	58
Chapter 13 Version and Release Management.....	64
Chapter 14 Using a Relational Database for Model Sharing and Distribution.....	70
Chapter 15 Reuse of Models with Copy and Paste	72
Chapter 16 Information Quantity Measurement.....	76
Chapter 17 Profile Extensions of the MetaModel	77
Chapter 18 Functionality	81
Chapter 19 Summary and Conclusions.....	82

A. References 83

B. References for Exceptional Students..... 85

C. DomainModel of DocumentDictionary 86

D. Glossary 87

Copyright © 2016-2018, ROJTEC, Olof Johansson

All rights reserved.

You may download and use a personal copy of this document. You may not distribute copies of this document to 3rd parties without a written permission.

Table of Contents

Preface.....	8
Chapter 1 The Purpose of OOCASE.....	9
1.1 Calculation of information quantity in a model.	9
1.2 Support for automated checks and documented quality assurance.	10
1.3 Globally unique 128 bit object identifiers.....	11
1.4 Open Export / Import in many neutral formats	11
1.5 Explicit well documented quality assured and release managed meta model.....	11
1.6 Automated source code generation for a number of well established software platforms.....	11
1.7 Purpose of the Infological Approach	12
1.8 Towards an Efficient Shared Information Highway Network for Implementing the UN 2030 Agenda for Sustainable Development.....	13
Chapter 2 Data and Information	17
2.1 Definition of Data	17
2.2 Definition of Information	18
Chapter 3 The Role of OOCASE in the Software Development Process.....	19
3.1 Brief history of OOCASE	19
3.2 Product Modeling Systems.....	20
Chapter 4 The Meta Model of OOCASE	22
4.1 DBObject.....	22
4.2 Element	23
4.3 ModelElement	23
4.4 NameSpace.....	23
4.5 Object	23
4.6 Package	23
4.7 Model	24
4.8 DataDictionary	24
4.9 DomainModel	24
4.10 Module	24
4.11 Class	24
4.12 Relationship.....	25
4.13 Attribute	25
4.14 AttributeGroup	25
4.15 Property / DataElementType.....	25
4.16 TypeDef.....	26
4.17 ValueDomain	26
4.18 Graphical Syntax used in Object Model Diagrams	27
Chapter 5 Domain Model Development.....	29
5.1 Single user application development.....	29
5.2 Team based application development	31
Chapter 6 Source Code Generation.....	33
6.1 Overall workflow	33

6.2	OOCASE built-in Source Code Generators	33
6.3	Source Code Generation Services	34
6.4	MetaModelDatabase SQL based Source Code Generators	34
6.5	Organizing Source Code Generator Build Systems	34
6.6	Using GIT for Source Code Generator Maintenance	35
6.7	Quality Assuring Source Code Generators with the Benchmark Domain Model	35
6.8	Test Suites for Source Code Generators	35
Chapter 7	Prototype Iteration.....	36
Chapter 8	Configuration Files	37
8.1	Directory Structure.....	37
8.2	Text format.....	37
8.3	Macro expansion \$(<parameter name>)	37
Chapter 9	Import of Domain Models	39
9.1	Import ODBC.....	39
9.2	Import TEXT.....	43
9.3	XML.....	46
9.4	Create XML Import Definition	49
Chapter 10	Export of Domain Models	52
10.1	Binary Storage Formats.....	52
10.2	TEXT	53
10.3	SQL	53
10.4	XML.....	53
10.5	XML DTD.....	54
10.6	XMI.....	54
Chapter 11	Quality Assurance, Version and Release Management.....	55
11.1	Purpose of Quality Assurance	55
11.2	Purpose of Version Management	56
11.3	Semantic Versioning 2.0.0	57
Chapter 12	Quality Assurance Functionality	58
12.1	Checking and Automated Checks	58
12.2	Check Level Structure and Checkpoints	59
12.3	Recording a passed check	62
12.4	Approving an object.....	63
Chapter 13	Version and Release Management.....	64
13.1	Editions, Versions and Releases.....	64
13.2	Version History	66
13.3	The VersionOwnerPath.....	67
Chapter 14	Using a Relational Database for Model Sharing and Distribution.....	70
14.1	Requirements for Providing a Relational Database Service	70
Chapter 15	Reuse of Models with Copy and Paste	72
15.1	Dominance Ranking of Classes in a DomainModel	73
15.2	The Copy/Paste Metaphore and it's complication in the real world.....	73

15.3	Examples for Functionality Coverage and Performance Analysis.....	74
15.4	Managing Traceability with Object Identifiers during Copy Paste.....	75
Chapter 16	Information Quantity Measurement.....	76
16.1	Theory	76
16.2	Practise	76
Chapter 17	Profile Extensions of the MetaModel	77
17.1	Short Introduction to Profiles	77
17.2	The DomainModel of Profiles.....	78
17.3	Comparison with UML Profiles	80
Chapter 18	Functionality.....	81
18.1	Egoless Business	81
18.2	Building Efficient Interfaces Between Huge Refactorable Knowledge Domains - Efficient Knowledge Economy	81
Chapter 19	Summary and Conclusions.....	82
A.	References.....	83
B.	References for Exceptional Students.....	85
C.	DomainModel of DocumentDictionary	86
D.	Glossary	87

Preface

To Software Engineers who want to get something done without complicating it more than necessary

OOCASE is an Object Oriented Computer Aided Software Engineering application that significantly enhances the productivity of a software engineer or collaborating software development team.

Every tool has its *scope of applicability*. Outside this scope there is an exponentially rising "cost/benefit" barrier that requires a different architectural and methodological approach to break through. This barrier has to do with what we humans are and the limitations of how much knowledge and information a person or small team of efficiently collaborating engineers can be masters of.

OOCASE works well for model driven software development for applications whose information models contain up to 250 classes and a similar amount of relationships. Such models may produce generated source code sizes of perhaps ½ a million lines of source code, for ONE implementation that can be maintained by a small team.

Even if OOCASE works well with object oriented information models up to 5000 classes and has been used for integration planning between software systems with more than 10000 classes and hundreds of thousands of attributes, such kinds of developments require larger teams of people that are organized in a healthy knowledge ecology by a company with healthy corporate governance and well maintained long-term win-win relationships with companies in its supply chain and its customers.

The purpose of this User Manual is to provide the theoretical framework for becoming proficient in using OOCASE, and with that conceptual understanding have acquired the knowledge potential to develop and deliver the next generation of tools we software engineers and our collaborating environment peers need, to with implementations that work in practice, cost-efficiently master the known challenges that decade spanning information systems management impose. Including providing users with high-performance.

Chapter 1 The Purpose of OOCASE

When the purpose of something is MUCH bigger than itself, it receives a divine force that can make it overcome
[The Poet]

The purpose of OOCASE is efficient use and reuse of Information Models¹.

An Information Model is a declarative design specification, or can be seen as a contract of what information a computer system software should be able to capture, store, process and present.

The information model is expressed in a language that after a short introductory training can be understood by non computer specialists, and serves as a tool for communication between 3 different groups of people. a) domain specialists that provide the requirements of information representation needed by their knowledge domain, b) computer specialists that implement a software solution, and c) end users of the computer system software who use the information model as documentation.

OOCASE is an object oriented computer aided software engineering tool with a number of unique features.

- 1) Calculation of information quantity in a model
- 2) Support for automated checks and documented quality assurance.
- 3) Globally unique 128 bit object identifiers for distributed development
- 4) Open Export / Import in many neutral formats
- 5) Explicit well documented quality assured and release managed meta model.
- 6) Automated source code generation for a number of well established software platforms.

Now it's perfectly alright and recommended to skip to Chapter 2 on page 17, since the following is only interesting for large scale project managers who build IT-systems that need to be operational for decades in whatever shape new technology allows safe information storage and exchange resources to be implemented upon.

1.1 Calculation of information quantity in a model.

Information quantity is measured in a unit called EC for elementary constellation. An elementary constellation or e-constellation is the smallest possible unit that still carries meaningful information that can be stored or transmitted as a message from a sender to a receiver.

¹ An information model defines Structure AND from the Structure directly inferable Behaviour. Inferable behavior that implements from the structure inferable application programming interfaces (API's) that follow easy to remember and use naming and parameter setting conventions and can be implemented by automated source code generators specialized for a highly optimized implementation in a particular target source code language. Other conformant languages such as UML define both structure and behavior. Large amounts of basic kinds of behavior can be derived from the structure and implemented automatically with model driven source code generators or model driven interpreters. For new application development it is cost efficient not to waste too much time on hand written software behavior before the core information structure (DomainModel) is well understood with examples that are entered into generated prototype applications populated with realistic production environment information.

The purpose of this measure is to provide decision support for which model to choose if there are several alternative ways of modeling a certain physical product or artifact, perhaps using different software systems.

A model can be anything that has been formalized with the language constructs in the DomainModel language to a level of detail that allows software to be implemented and the model's information stored in a database representation. If two models provide equivalent functionality but differ severely in complexity or computational performance on present technological infrastructure available for the purpose of the model's end user application, someone has to decide which model to choose.

Fact based measurements are decisive (or important decision support in political environments) when the accumulated cost of several decades of IT-system maintenance for serving a fleet of complex engineered products that delivers a fundamental service for the sustenance or protection of a customer's society, requires IT-support for cost-efficient management and maintenance.

In a modern larger engineering company that promotes its staff by merit and leaves plenty of opportunities for choices of a future career open and inviting, the time at the IT department while learning the information structures of the core business is a knowledge development platform for the staff supply that can take on challenges of the more advanced jobs². Jobs that require an understanding of how to implement production capacity for new business opportunities. This in modern engineering companies generous staff meritocracy requires that the supply can be held up by efficient education of new staff to replace the vacuum for skill enabled career advancement that exposure to this core business knowledge in a concentrated format produces. The choice of model can actually impact the choice of paradigm for career advancement in a company, and with that the whole future for its business³.

The output of a modern larger engineering company in the form of life quality sustaining societal infrastructure products are frequently taken for granted by consumers. It is only the experts of these products who have the REAL power to make them deliver their output at the cost possible given the current state of the art knowledge in all those fields of expertise who combined make such products producible at an affordable cost for the end users of the product or its services.

Information quantity based decision support is especially important when two separate communities fight over a standard and are unexperienced⁴ in each others technological domains or software implementation support for these technological domains. Unnecessary complex models are harder to teach and maintain, and divert resources from other important areas of development. Having some measurable facts may resolve disputes and get the "fighting communities" focused again on delivering added value, standing on a fact based ground, and if corporate governance is excellent, get amplified by mutual education.

1.2 Support for automated checks and documented quality assurance.

The Quality Assurance techniques applied here were adopted from the mechanical engineering industry where the high cost of failures due to errors in design specifications (drawings etc) drove this industry to develop survival skills that add some spending in the earlier stages of development. Where this added spending serves as insurance against unpleasant expensive surprises later. OOCASE supports a number of levels of automated quality checks, that aid various "check-points" in a cost efficient iterative development cycle where many people (frequently with too little time) are involved.

² Where the best examples of Engineering Companies are plain clever self sustaining career production machines for talented people whose skills are necessary to fix our real problems.

³ There is always potential competitive advantage layers above a current well-known established business. Those above layers requires creative people who know the core business AND something else that none of the established market players have thought of before or delivered the investments to make it become available for a solvent enough customer basis who appreciates its new products and buys them for a GOOD reason. If the core business is obscured by an unnecessary complex model, the supply of people who understand it well enough to develop the potentially business income generating layers above it will be throttled. This is stuff that matters over decades, when "great asset" people move for reasons outside the control of the company.

⁴ This is a real fact due to the enormous size of various industrial information models, and the amount of studying time and practice it takes to become familiar enough with their details to make fact based decisions.

1.3 Globally unique 128 bit object identifiers

Ensuring global unique identifiers for objects is a re-occurring problem throughout the whole globalizing IT industry. It has with performance in applications to do, scalability and ability to produce large amounts of uniquely identified objects in parallel by people and teams that are unaware of each others existence. Where the unique object identifier issuing mechanism must provide the ability to combine results of independent uncoordinated work in a database that requires unique object identification AND traceability for quality assurance reasons, and efficient on-demand-access to linked external resources outside the local database.

The method chosen for OOCASE and plenty of production systems produced, is providing each creator of new object identifiers with a unique 64 bit identifier (HighId). Each such creator has a self managed 64 bit incrementing counter (LowId) for lifespan unique identifiers from that source.

There are plenty of other ways to solve this problem, but this approach is simple, efficient and it works in practice.

1.4 Open Export / Import in many neutral formats

OOCASE provides many ways of exporting and importing information models. Thus your models are never locked-in within this tool. If a better tool comes along (creative destruction) you can proceed with that.

1.5 Explicit well documented quality assured and release managed meta model.

The meta-model of OOCASE is licensed to all paying customers for their own implementation needs in the most empowering format. OOCASE is modeled in OOCASE. Thus if you develop source code generators for a new software platform that completely outperforms the one OOCASE is using, you are free to implement your own OOCASE tool on that new platform and migrate to that platform with all your information model assets intact⁵.

1.6 Automated source code generation for a number of well established software platforms.

Source code generators are available for a number of SQL92 compliant relational databases, Smalltalk, C++ and a few other programming languages.

The rest of this introductory chapter is there for readers who wan't to understand the benefit of a more efficient standardized "asphalt laying machine" for putting "tarmac on the emerging INFORMATION gravel roads of all diverse shapes and sizes, so they without numerous severely errorprone and costly reloading can carry the truckloads of global information we need to ship to the computational centers that can convert it to reliable decision support for our industrial leaders and national governments.

⁵ This may seem stupid with regards to the self-sustainment principle of the company making a living on OOCASE. However in the perspective of global warming whose solution is more important than the self-sustainment of a particular company whose employees can find a living somewhere else, it's non-productive with regards to over history gathered experience to prevent "creative destruction" to happen if the new alternative over time and by facts and evidence delivers a much better output performance with regards to achievement of the goals setup in the UN 2030 Agenda.

1.7 Purpose of the Infological Approach

To understand the purpose of any kind of software one has to analyze its role in the larger whole.

The following is a quote from the preface of [Sundgren 73] which describes the foundational infological theory on which OOCASE builds.

"An infological approach to data bases" reports parts of the data base research and development work which has been carried out over a number of years at the National Central Bureau of Statistics, Sweden. Professor Börje Langefors, University of Stockholm, Department of Administrative Information Processing, has been the scientific supervisor of the reported project. Very briefly the objective of the project has been to develop an integrated theoretical framework for design of large-scale data bases. The framework should

- (a) enable people who are not data processing professionals to co-operate actively and constructively in data base design projects
- (b) make it possible to transform systematically the problems, desires, and requirements of those who are affected by the projected data base into problems which can be tackled by data processing specialists
- (c) enable data processing specialists to analyze the computer-oriented data base problems systematically and with sufficient precision
- (d) make it possible to design data bases with which decision-makers, planners, and researchers within different specialized fields could interact constructively, even if the information needs of the interactors are complex, and even if they lack knowledge about computers and computing

There are definitely different opinions among authorities in the computing world as to whether it is feasible to cover all the aspects (a)-(d) within one and the same framework. This report supplies evidence in support of the hypothesis that an integrated approach is both feasible and necessary for the success of large-scale data base undertakings."

The above quote from year 1973 is still valid, however the situation has improved. (a) has been improved with graphical representations of information models that are used in interactive development seminars where a mix of domain specific expertise participate and all understand what they are talking about so efficient communication can take place. (b) and (c) have for the purpose of implementation of basic information handling software functionality for delivering fully functional prototype software implementations been fully automated for certain target platforms. (d) has been significantly improved with automated model driven declarative implementation of software prototypes from information models, that enable domain experts to express their expertise with large scale examples, that reveal the "problems" in the details, where the *Information Model* does not adequately represent reality. Some stuff in seminar or prototype evaluation situations are "gutt feelings" of participating experts, and it requires certain "emotional language literacy" and social skills by a software engineer/seminar leader to get that information out. According to a non-disclosed source there are 39 different emotional expressions that experienced people use while communicating interactively. Human skills in understanding and acting efficiently on the cues of non-spoken emotional language AND knowing the domain of the expert to a level where the facts can be brought out by asking the right questions requires a special brand of people, of which YOU are a candidate.

Or you can focus on the technical implementation parts of translating declaratively specified *Information Models* to efficient software implementations on the latest superior hardware and software platforms.

1.8 Towards an Efficient Shared Information Highway Network for Implementing the UN 2030 Agenda for Sustainable Development

Some problems can not be solved in traditional ways since there is no-one who owns the problem. Or there exists no single entity with enough resources or authority to solve the problem in practice. Or the entity producing the problem is not within the authority of the entities subjected to its effects.

In order to solve a problem it must first be understood. The UN 2030 agenda for sustainable development has set out a goal. To reach that goal, we need a plan for how to get there.

A typical approach could look like:

- 1) Build a reasonably adequate map of the current situation
- 2) Identify spots where investments would provide largest return with regards to goal achievement
- 3) Allocate resources to fix those problem spots
- 4) Implement the fixes and restart at 1)

Besides ignoring the natural law of self-sustainment⁶, the above approach hit's it's exponential boarder of applicability rather quickly, due to a problem we encountered in Software Industry a long time ago:

The Language Problem

We still have not solved it satisfactorily however software industry rolls on, in its complex supplier value chains, each actor in it's own little language islands, at the "speed and load capacity of a horse/8-bit CPU", where we could use a "modern truck/64-bit multi-core" instead.

But that is unfortunately not possible, since there are too few "roads/standards" that can "carry such a truck/make use of available information exchange eco-system" (even if there are instances of such "roads" and "trucks" in certain niches that have an enormous turn-around).

Even if it is possible to implement a "truck" for, for instance "Environmental Data", there would not be a large enough market for it. The infrastructure ranging from CPU, OS, DataBase, Network Communication Protocols, User Interfaces, Local User Language Adaptations, needs to be implemented with instances for which there is an educated work force who can install and maintain them.

And finally the end users, which collectively are the most expensive and valuable actors in this chain, needs to be educated in order to know what decisions to take based on the delivered "Environmental Data".

But there is a solution to this. A distributed one. One that will require an agreement or arrangement of peace and non-hampering interference by external actors that have their own agendas (or internal problems) and don't care about "the truck's" success since it will not be under their control.

⁶ Law of self sustainment) An active entity e.g. @) biological cell, a) plant, b) animal, c) person, d) organization, e) company or f) nation;

that can not find a way to sustain itself with; @) nutrition and energy (provided by its organism), a) nutrition and sunlight, b) food, c) food and housing, d) work force, e) work force and income to pay taxes and work force, f) tax income (to pay for education, defence and law enforcement), educated workforce, peace and stability, law enforcement (that allows its educated citizens to set up companies that are not robbed of their resources (material, money, working time), and employ an educated workforce that can produce the added value necessary to generate tax and the companie's self sustainment);

will starve and eventually perish. The basis for action is energy, without energy action is not possible.

The solution is based on some lessons from the evolution within software industry and other domains, and delivers shortcuts that can shorten the time table from decades to years.

To explain this the "generalize from examples" approach is taken

1.8.1 The Software Expansion

The personal computer was founded on the invention of the CPU on a chip. Success stories of chips like the Intel 8080 lead to low cost competitors like the Zilog Z80, Motorola 6800 and MOS 6502. The possibility to mass produce personal computers at a low cost lead to an extraordinary expansion during the late 1970's. The availability of PC's in all kinds of different industrial, academic and personal environments lead to the creation of software industry that in turn had an extraordinary expansion in the 1980's.

Due to the large distributed presence of personal computers and software tools for software development, the same scale-up problems when software grew larger were detected in vast amounts of different places. Plenty found their own solutions and put them into their own software and software development methods.

In the 1980's a very diversified ecosystem of programming languages, software development methods, tools and software products had emerged.

In academic and other meeting arenas for software developers, people realized that they had common problems with translating data from one program to a suitable format that could be used in another. The same thing appeared in larger software development projects where models and methods from different interacting sub suppliers had to be interfaced or integrated. There was a growing need for a common language.

There seems to be three approaches for finding a common language. These have been tested in plenty of computer science historical peak events and delivered their output in the form of publications, standards, organizations and companies maintaining the standards. The three identified alternatives are:

- 1) Some clever people with indepth knowledge of the problem, design something that takes the best out of everything they know and design something new that solves the problem.
- 2) A large group of representatives with different backgrounds, requirements and visions come together and with fact based arguments and efficient negotiation techniques come up with a compromise that works and deliver an output that is useful to all participants.
- 3) Some actor goes ahead and markets its solution to a scale that it in practice becomes the de-facto standard that gets the most users and thus everyone, in one way or another, has to adapt to. A lock-in that, if the technology is inferior compared to others, may put a suffocating blanket on development seen from a larger context and the potential available in its large user base.

OK, so what are we supposed to say about this?

A Swedish famous quote is: "Ja, det är för jävligt" and that ends the story with a statement that everyone can agree with. A big sigh and no change. However that comment is a violation against since long gathered wisdom. The ones that know something better need to pursue their acts, knowing and learning more, and facts if proven usually makes a difference. You might know where the facts are, however you must find them and present them in a way that serves the purpose of change towards the better.

And remember, it usually does not matter what a GOOD leader who cares about her/his followers does, as long as it put's unity within the group which ends internal fighting and frees up working resources for working towards a goal that leads to an improvement.

So lets focus on combatting global warming, and the facts about what is needed for that will fix the rest.

1.8.1.1 1) Engineered Standard - Relational Database Language (SQL) Example

Proos:

+

Cons:

-

1.8.1.2 2) Negotiated Standard - Unified Modeling Language (UML) Example

Proos:

+

Cons:

-

1.8.1.3 3) DeFacto Standard - (somewhat sensitive to select the example) Example

UNIX, Windows, Linux, C, C++ etc.

Proos:

+

Cons:

-

1.8.2 The Railway Expansion

With the invention of the steam engine, and the landmark locomotive Rocket , it suddenly was possible to move heavy goods and passengers over larger distances that outperformed "horse and carriage". Railways were built in all kinds of places in Europe by entrepreneurs with a transport business idea by different contractors.

As the different tracks came to meeting ends, it eventually became clear that the whole transportation system would become much more efficient if there was a standardized track size. The benefit with a common standard was that the goods would not have to be moved between trains running on different track sizes at the meeting points, and that a manufacturers of locomotives and waggons would get a much larger market if all rail-tracks had the same dimensions, thus enabling mass production of the same designs, with a better profit margin as result, and competition which lowered the prices for "rolling transport infrastructure".

1.8.3 The Shipping Industry Expansion

A break through in the cost efficiency of shipping was the invention of the container. A standardized package for all kinds of goods that could be transferred between ships, railway carriges and trucks reduced time and cost at goods transit points. Just the lowered transport cost made it possible to trade larger volumes of goods at a lower profit margin, thus increasing trade.

The common lesson from all these examples is that the EXACT formulation of the contents of a standard is not that important, however it MUST work in practice, it MUST be efficient compared to the state of the art, and it MUST have a self-sustaining system structure where all participant roles in that structure have win-win relationships to each other. Who is taking a particular role in the system structure is not that important if that actor maintains healthy win-win relationships with its partners in a supply chain AND maintains healthy co-opetition with its competitors, where they collaborate on developing efficient standards for the higher layers that are not yet mature enough or large enough to provide the volume benefits of a by GOOD standard enabled mass market. There is always a higher layer for the actors that are thrifty, follow the natural laws of healthy business ecosystem dynamics and build their value adding products on the best available standards.

Chapter 2 Data and Information

Some things are immaterial, but must be transmitted with the aid of the material
[The Poet]

Most people have an intuitive understanding of the terms data, information and knowledge. Like for most words this understanding is a product of their life experience of how these words are used. Many see them as different labels for the same thing, and use them interchangeably. People talk about databases, information technology and knowledge based systems, without really making much difference between them unless having some deeper interest or experience in developing such things or exploring them.

There is however a clear benefit of having a more precise definition and understanding of the concepts of data and information in the quest of developing efficient information system.

2.1 Definition of Data

The following definition is from [Sundgren 73], the thesis that my GREAT professor put in my hands after entering his office and explaining my problem of making measurable science out of data modeling.

Definition of Data: [Sundgren 73], page 20

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.

This definition covers all kinds of data: digital data, analogous representations, spoken and written language, etc.

...

Note the wording " ... intentially arranges ... to represent".

It is not sufficient that two phenomena are related to each other, incidentally or by design. There should have been a human intention of representing , and not only correlating, one thing with another.

In most cases data represents primarily a person's knowledge about reality and only secondarily the piece of reality itself.

People have been using data as an aid in their daily activities for a very long time. Before the number systems were invented or taught, shepherds had a bag with small stones, one for each sheep, which they used to account for them.

2.2 Definition of Information

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.

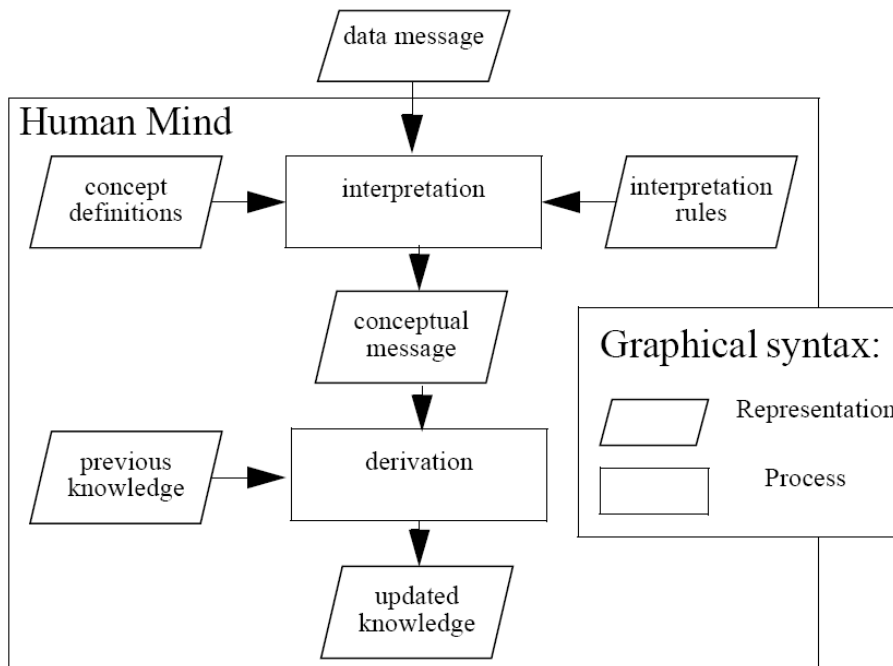


Figure 1. Transformation of data into information. From [Sundgren 73] page 24.

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:

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

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 theory was later refined in [Sundgren 73], and adapted to the 1990's main stream scientific and commercial software supported theory of object-orientation in [Johansson 96]. That thesis presented a framework for measuring information quantity in a unit (e-constellations) that is close to the underlying representation in which humans store and index information in their brains [Appendix B].

Chapter 3 The Role of OOCASE in the Software Development Process

A skilled actor can play many roles. The extent to whether the performance is perceived as an excellent experience, depends on how well the actor knows the role and it's purpose

[The Poet]

The features of OOCASE, such as the ability to work with very large object-oriented information models has its roots in the tool's history. The special scale up and productivity requirements forced it to focus on a practically working core theory. Whereas many other CASE tools originated from the need to document existing large software systems in a graphical way, OOCASE had the privilege of being the origin of the design specification of whole systems.

OOCASE is not restricted to large scale applications. The productivity benefits of model driven source code generation start to pay off already when developing small databases with around 10 tables/classes. The benefit of building a new application on other reusable core model modules that provide specific functionality such as persistence, timestamping, quality assurance, version management, that are already supported by a range of tested well working source code generators, saves plenty of "reinventing the wheel" basic strait forward programming efforts that are already automated by source code generators.

Thus more of a developer teams' time and effort can be allocated to "high value" activities with regards to functionality and productivity of end users.

3.1 Brief history of OOCASE

OOCASE has its roots in the late 1980's and early 1990's. In those times the "patterns of thinking" and understanding of information systems development met with leading reasearch in software development and formed new ways of thinking that set foundations for great leaps in software development productivity.

Concepts that had been discovered and rediscovered almost everywhere where larger software systems were developed now became widely published. Some agreement on common vocabulary and terms started to form in the literature and commercial development tools.

Those were the times of 4'th generation programming languages⁷, and vendors and users of Computer Aided Software Engineering (CASE) tools were developing new markets.

There was however a gap in the CASE markets in the area where main stream commercial databases met with computer aided design (CAD) tools. The software tools at that time had been developed by groups with different end user requirements. The database groups were focusing on masses of textual business data, whereas the CAD groups were focusing on graphical drawings of mechanical and electrical designs.

⁷ Programming Language Generations: 1st) = numeric machine code, 2nd) assembly language, 3rd) languages like Pascal, C, ADA, etc.

Large electrical and machinery engineering firms had a need to integrate large amounts of business and CAD data in their information flows, and that job was done on a case by case project basis by a new wave of software application consultancy businesses.

The idea of an integrated product model (PM) that described all aspects of these large engineering firms' products and how to build such systems with the CASE tool approach began to form.

3.2 Product Modeling Systems

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

Figure 2 shows the approach taken to manage the software engineering of product modeling systems. The idea is to maintain a high-level PM-system design specification in the form of an object-oriented CASE model in a meta-database; in the early 1990's commonly called data dictionary.

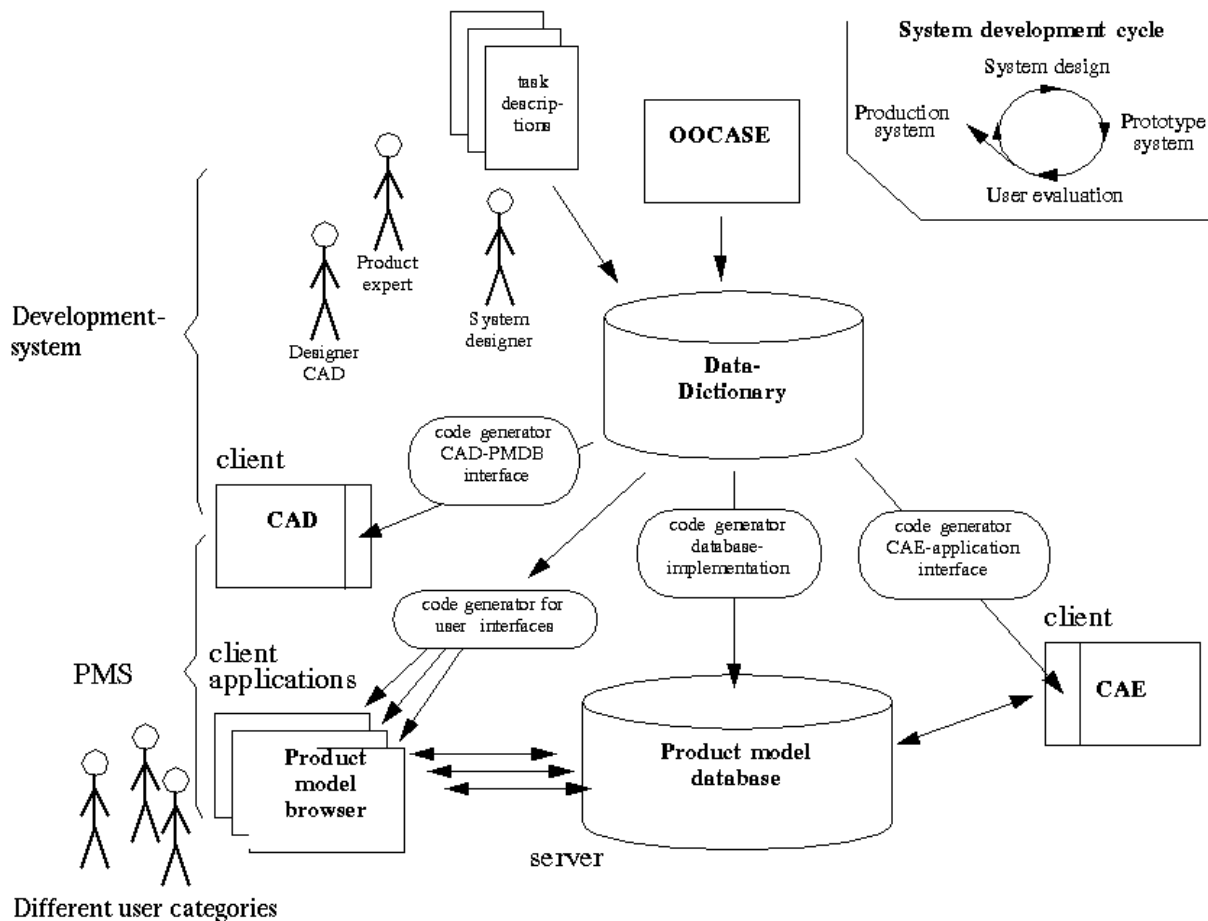


Figure 2. Software development approach for product modeling systems

The OOCASE *DomainModel* of the objects to be managed in the product model database is developed in cooperation with product-, CAD-, and CAE-application experts.

The software development approach for product modeling systems works as follows.

Most of the source code for the PM system implementation is generated automatically, using SQL-based source code generators. The 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.

The System development cycle depicted in the upper right corner of Figure 2 consists of:

- 1) System design that is modeled in OOCASE and maintained with the help of the DataDictionary.
- 2) Creation of a prototype system where the code generators deliver the source code that directly depend on the design of the DomainModel for the PM system, which may change in each iteration.
- 3) User evaluation, where the users model their products with the functionality provided by the PM-system prototype, give feedback on functionality and performance, get inspired and deliver new lists of wishes and requirements, which are collected and taken into the next System design iteration.
- 4) If the users are satisfied with the functionality of the PM-system, it is taken into production on a system platform that can handle the load of a full user base and lives up to the requirements of a production system. Which are maintenance requirements with regards to regular backup procedures, bug-reports, user help-desk support, basic introductory education of new users, etc .

Chapter 4 The Meta Model of OOCASE

A life shapening peak experience for a software developer is the first time experience of implementing a system that can implement itself.
[The Poet]

Manual author's comment: From FORTH to Heaven. Heaven from which you can see SPACE and with your previous experience get a clear hunch on how to explore it.

This chapter provides understanding that is important for your productivity in OOCASE. If you are familiar with UML class diagrams, please have a look at Figure 3 on page 28, and read section 4.18 Graphical Syntax used in Object Model Diagrams.

The information of an application domain can be described with an information model named DomainModel. A meta-database serves as a data dictionary database for domain models.

In the same way as a PM-database is an engineering database described by a PM-system's domain model, the meta-database is a software engineering database, described by a meta-database's domain model. The domain model for the meta-database is actually stored in the meta-database itself, and has been used for generating most of OOCASE's object manipulation software, such as internal classes and object oriented methods, software components in its browser user interface, and the table structures of the meta-database itself.

Figure 3 on page 28 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.

You can also open the DomainModel named "mmdb404e.odm" or later versions in OOCASE, and browse the classes and their definitions interactively.

4.1 DBObject

All metamodeldatabase 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 metamodel database, this kind of functionality can easily be implemented with triggers.

DBObject also defines attributes for quality assurance (checkedBy, dtChecked, approvedBy, dtApproved), version and release identifiers (version, release) and full traceability to the original object that an object was copied from (releaseBasedOn-attributes). This functionality is described in Chapter 11 - Chapter 13

4.2 Element

OOCASE is conformant⁸ with the UML standard, and has adopted it's basic superclass structure for common model elements. An Element is a superclass of all meta classes.

4.3 ModelElement

A ModelElement is an abstract class (one without instances) that has a name and a definition attribute.

A well choosen *name* for a ModelElement that is comprehensible for all collaborating people in the development process as well as expected educated end users is an investment that has an enourmous payback in saved working hours for communication among collaborating people over the life cycle of the modelled software.

The *definition* attribute is used for defining the purpose of the model element. It's model element specific value is used for generating documentation, help files, comments in generated source code etc. Thus spending effort on a well formulated concentrated definition that is understandable for all involved people that will work with the model element or instances in generated software applications will pay back over decades to come.

4.4 NameSpace

A NameSpace inherits ModelElement, and is an abstract container class that owns other ModelElements that are identified by unique names within the NameSpace. Subclasses of NameSpace model program construct in generated source code.

4.5 Object

Object inherits NameSpace. Meta-database objects which are created by the developer such as classes and attributes inherit from Object. When working at an international company such as ABB, the PM system domain models have to be coordinated between different sites in different countries. Therefore it is useful to besides *name* that is inherited from ModelElement, 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. The English names are used for source code generation. Due to name size restrictions in a wide range of possible target software platforms such as relational databases, GUI languages, CAD system programming languages etc, the English names should have a size less than 27 characters. Sometimes, however, older target languages do not allow such long identifier names and then a shortName can be used instead. The size limitation of names in the meta model database is 127 characters.

The label of an Object is frequently left blank, however in some generated applications it is used as a compact short hand to designate a unique position withing a hierarchical part-of structure in the same way as "4.5" is a short hand to identify this particular section in the OOCASE User Manual.

4.6 Package

This abstract class serves as a container for higher level model elements such as a Model or a Module. It is conformant with an UML package.

⁸ Conformant means that model elements/meta-objects in OOCASE have a mapping to UML model elements and carry the same information, however the names of those model elements may differ a little and the structure of the information carrying objects may differ somewhat.

4.7 Model

A model is an abstract class for a self contained model that is handled as one unit. Within a generated application, a Model can for example be saved in a separate file and the abstract Model class serves as place holder for implementing such reusable behavioral functionality.

4.8 DataDictionary

The owned contents of a DataDictionary instance is adapted to a particular range of applications that are defined in DomainModels that share and reuse that DataDictionary. Some DataDictionary instances stored in the meta-database may contain the full range of imported industrial standards. These however are typically too large for practical application DomainModel development, and are used as reference libraries for search, copy/paste⁹ to a development datadictionary which only contains the datadictionary specific model elements that are used and reused in the DomainModels under development in its particular range of applications.

4.9 DomainModel

For each application or product Modeling system implementation, there is a corresponding domain model. Instances of DomainModel represent the “root”-object of such a domain model.

4.10 Module

The Module enables a large domain model to be structured in Modules and submodules. Only Classes and relationships can be assigned to a module. A module typically has a 1-1 correspondence with an IDL¹⁰ module that declares the application programming interface (API) which the module provides. The API is manually implemented in a reusable software library for a particular object oriented target language. The Module is an information model that documents this library. In OOCASE a module does not declare imports like a package but should be entirely self contained to simplify source code generators.

4.11 Class

Instances of Class hold class-related information. When generating source code for a non-inheritance capable relational DBMS, the inheritance hierarchy is flattened, so that leaf-classes have all their attributes stored in the same table. In such (widely industrially used) databases, only leaf classes are relevant for generation of tables. To distinguish them from abstract inherited ones such as DatabaseObject in Figure 1, leaf classes have their genSqlFlag set to TRUE. The prefix holds a two-to-six character prefix that uniquely identifies the class within a domain model. This is useful since many named source 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 [Johansson 1996] 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 160713: OOCASE can currently only have one DataDictionary open at the same time, thus open the library DataDictionary, select the DataElementTypes you need, copy them to the clipboard, then load the development datadictionary, and paste them there.

¹⁰ IDL is the CORBA Interface Definition Language standardized by OMG.

4.12 Relationship

Relationships in DomainModels 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. The M-N case is available if the target platform has performance beneficial special support for it.

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.

If you need a ternary relationship or higher, you create a class to represent that relationship and set mustExist1 and mustExist2 accordingly on the 3 or more binary relationships to the three or more related classes to ensure referential integrity in the generated code. This well-known transformation trick saves at least 3 times the implementation effort in source code generators for a particular target language, and much more, if you like any serious software developer, implement full design space and function point covering testcases for a benchmark domain model.

4.13 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/DataElementType, it will receive its data type declaration via the path

Attribute.dataElementType->Property/DataElementType.DETValueDomain->DETValueDomain.valueDomain(*implementationConcept*=NULL|int64)->ValueDomain.typeDef->TypeDef.declarations(*language*=SQL92|IDL|C++|Smalltalk|...)->Declaration.declaration, where *implementationConcept* selects a 32 or 64 bit platform and *language* is a parameter that selects a Declaration for the programming language.

4.14 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 DomainModel and for spacial grouping of related attributes in generated user interface objects.

4.15 Property / DataElementType

A property is a feature that belongs to an object. 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 PM system 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. The name of the `Property` meta class was changed to `DataElementType` in 1998 to adhere to the industrial standard IEC 61360, which contains a standardized library of thousands of properties for electrical engineering. Later IEC 61360 was adopted by `ecl@ss` which is an e-business standard for product data exchange that contains more than 17 000 standardized properties.

4.16 TypeDef

All variable and attribute implementations are based on some data type. A product Modeling system 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/Microsoft SQLServer's Transact-SQL. When generating code, especially for interface libraries between CAD-systems and databases, this kind of data-type mapping information is necessary. The specific textual declaration of a `TypeDef` for a particular programming language is stored in a `Declaration` object owned by the `TypeDef`. In the OOCASE meta-database, there is 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 [CORBA 1991], and [CORBA 2012].

4.17 ValueDomain

An example of a `ValueDomain` 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 PM system 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 `ValueDomain week_number`. Through the relationship `dataElementType_attributes` the attributes of the `DesignActivity` and `Manufacturing Activity` are connected to their corresponding `Properties/DataElementTypes 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 `valueDomain_DETValueDomains->DETValueDomain(implementationConcept)->dataElementType_DETValueDomains->Property/DataElementType` and `dataElementType_attributes`.

By changing the definitions in the `ValueDomain`, the change can be automatically propagated through the `Properties/DataElementTypes` to attributes in all domain models. Using the code generators, the product Modeling systems can be re-implemented with the new data type.

This section gave a short description of some central classes and relationships in the domain model of the OOCASE metamodel database. More details are available in MetaModelDatabase_4.[0-9].[0-9].[a-z].odm and [Johansson 1996].

4.18 Graphical Syntax used in Object Model Diagrams

The graphical notation for DomainModels are called ObjectModelDiagrams. These are an efficient compact and practical working representation for describing classes, inheritance, attributes, and relationships between classes. They are easy to use in seminar or team discussions and serve as "mind maps" for accessing the details of the DomainModel. The notation is easy to teach team members of all kinds of knowledge domains, including people without programming experience.

In the DomainModel of OOCASE in Figure 3 on page 28, classes have their class name in bold font in their first box. The second box contains the superclass's name preceded by a "->". All attributes and relationships specified for the superclass are inherited by the class. The third box within a class contains a list of attribute names.

The naming convention for relationships is "<name2to1>_<name1to2>".

Relationships between classes have cardinality constraints telling how many instances that must participate in the relationship on each side, specified as intervals "<min>..<max>". A '*' as <max> denotes infinity. A black diamond on the owner class side of the relationship denotes that instances on the other side belong to that class. An unfilled diamond denotes that the instances on the other side are aggregated by a class, but are "physically" owned by another relationship.

Descriptions of the notation of UML class diagrams can be found through [OMG UML]. See [Fowler 2003] for an introduction to UML.

The design choice of using a name reference like "->Superclass" in the graphical notation of a class in an ObjectModelDiagram to identify the superclass of a class, is based on plain practical working experience with large and complex object model diagrams.

In a standard UML Class Diagram an inheritance link is represented by a graphical wire with a unfilled pyramid symbol that connects the superclass with its subclasses.

The amount of manual working time lost with graphical re-routing of inheritance links while incrementally evolving a DomainModel with 20 or more classes, is the rationality behind the design choice of this graphical syntax.

Chapter 5 Domain Model Development

DomainModeling is the Art of tapping a limited group of bright minds of their great immaterial assets, and turn their collective immaterial value product into something that can be transmitted with material means. Means that empower and emancipate each and everyone in the group .

[The Poet]

Manual author's comment: This requires a significant understanding of human beings and human nature, and the culture that has shaped the groups' patterns of thinking. It also requires some teaching skills to make some members see and understand the value they personally receive from the shared effort. The better you know your team, the better you can make it perform. However you need to believe in what you are doing. Reading the business plan of the highest ranking responsible manager is the kick you need to get started.

In order to get access to bright minds, you need to give them something they want. Being well prepared, knowing your tools and showing by real examples that your audience are interested in, is one good way of earning trust. Trust that opens doors as long as you can deliver.

First you need to become an expert in using your tools. One very good way to become a master of a tool, is to use it to develop something you are severely interested in yourself. Section 5.1 gives you a framework for developing your first application. It is very important that this application is something that delivers value to yourself, and thus gives you a reason to use it yourself. You need feedback from using your own application to develop the skills necessary to improve it with a rational way-of-working. Skills you can reuse when taking on larger projects.

With the practical experience of the craft internalised, you are ready to take on a realistic team.

Section 5.2 gives you the basic framework for team based DomainModel development. This is fun, exciting and demanding. It is very helpful to have run some pure interest based team projects where all members have a common goal that they are all excited about, and where the outcome can only add value if it succeeds, and if it fails, it's purpose was plain teaching and practical experience that can be reused later. The team scenario delivers useful practice with the mechanisms necessary for efficient rational feedback loops on complex design structures.

5.1 Single user application development

This scenario is for developing an application for the developer's own needs. In this case the developer fulfils all 3 roles a) domain specialist that provide the requirements of information representation needed by the knowledge domain, b) computer specialist that implement the software solution, and c) end user.

Having gone through the whole development life cycle for at least one application, prepares a developer for taking on the role of b) computer specialist in collaborative teams whose products target a particular customer or end user group.

5.1.1 Reuse of modules and frameworks for new application development

Reuse is fundamental in all productive value adding businesses. Designs that work in practice and have proven themselves in numerous tests with reality carry immaterial value that form a foundation that delivers leverage from the start. Becoming proficient in reusing proven modules is a basis to be able to deliver to your team.

Reuse is beyond the functionality delivered by a module. It is also about reusing the preknowledge of the people that come in contact with it.

Before building your first own application you need to acquaint yourself with an example of an application that you are familiar with and use in your own work. Which modules does the example application contain? Which of those contain functionality that you can reuse and stand on to build your own application?

Study the DomainModel of the example application, both on the meta level, which is the DomainModel and the application level, which is the application you use.

The more modules you acquaint yourself with, the more ground you find to stand on, given that your customers need that functionality for their business in the situation they are now. Careful preparation to move your customers on to the next level when they are ready, is something that is worthwhile if you see more potential in their business than they do themselves.

5.1.2 Sketching an outline of the application structure

After initial thorough studies of modules and example applications, you need to rest. Do something you enjoy. Physical exercise that delivers adrenaline kicks and afterwards endorfine kicks is one example. Enjoy being a human being for a while. After a good nights sleep, rested with clear thinking, just outline your design the way it comes out.

Initially paper and pencil may be faster than working with a tool. Write down keywords in boxes, sketch relationships, work your way down your design. Anything goes for the moment, the purpose is to deliver a physical sketch that you can use to collect your thoughts around. Don't be self critical, just record what your mind has come up with.

Once the ideas and creativity starts to faden, go through your design and use your tool to describe it. During that detailing process, new thoughts will come to mind and you might have to adjust your sketches.

5.1.2.1 Application Root Object

An application must have a root object to maintain the integrity of itself. It should be something that you naturally think of as the "master of all the details". Let the most desired functionality of the application decide what is a useful root-object. You can always at a later stage create a new version of the application with a root-object on some higher level.

Example 1) When OOCASE was initially developed, the DomainModel was the root object. Later when the basic functionality started to deliver its value, the DataDictionary root object was added, while preserving the integrity of the DomainModel with a relationship interface between Property/DataElementType and Attribute.

Example 2) DocumentDictionary started its life as a simple Excel spreadsheet containing the table of contents of a course book and some columns calculating the page-count of each chapter, a column estimating the TimeToRead and a column to enter the recorded ReadTime. The root-object in this case was the excel file itself, which corresponds to a DocumentRecord in DocumentDictionary.

5.1.2.2 Names for all classes, attributes and relationships

Given the root-object, define a class for it, with a name and a definition that states its purpose. Then continue with the "details" that are parts of the root-object. Create classes for the details and define the relationships they have with the root-object and amongst each other. Continue on this abstraction level until you feel that you have got the names right.

5.1.2.3 Definitions

Definitions are initially as much a tool for thinking as a way of documenting the design. When formulating the definitions you will discover things that might change the way you are thinking about the design. Having thought through the Names before starting to write definitions on a level suitable as documentation for other people, saves plenty of rewriting work.

5.1.3 Feedback from automated checks and the team

Once a comprehensive draft of the outline of the application is ready, it is time to receive feedback on it. The fastest way to identify well-known errors is to run automated checks. Automated checks identify common mistakes, such as name collisions in shared name spaces or deep class inheritance hierarchies or lattices, names violating naming conventions, forgotten definitions, too long or too short definitions, etc.

Once the errors and warnings from automated checks are corrected, the design can be shared with a team member for feedback. A well prepared draft is much more time efficient and respectful than a sketch that requires you to be present in real-time to explain what parts are just a preliminary non-thought-thought dump of initial thoughts, and what parts you really want your team member¹¹ to comment and give feedback on.

Once you have got feedback from whatever source and made adjustments to a level where you feel satisfied, it's time to prepare the design for implementation.

5.1.4 Preparing for application code generation

Different target platforms have different declarative information requirements on the DomainModel, that depend on the programming language of the target platform.

To be able to generate source code for a particular target platform, the DomainModel must satisfy a number of modeling rules, in order for the source code generators to produce source code that can be compiled and executed and while being executed will deliver the correct application programming interface (API) behaviour.

Modeling rules are implemented with a set of automated checks on the contents of the DomainModel. There are several ways to implement Modeling rules:

- 1) OOCASE built-in checks, which contain the accumulated experience of common modeling errors for a range of target platform projects and can be configured and run from the Quality Assurance, check functionality.
- 2) User developed SQL-query based checks, that with an SQL SELECT statement detects a known error condition on the rows that belong to the particular domainmodel instance that is checked, and can be directly run from the Reports->Queries interface.

Once all errors and warnings detected by the checks have been corrected, the DomainModel is ready for source code generation.

5.2 Team based application development

This scenario applies when the roles of a) domain specialists, b) computer specialists and c) end users are held by different people. In an efficient well practiced approach, the b) computer specialist serves as seminar leader and teacher of the graphical notation of the ObjectModelDiagrams.

¹¹ If you are an experienced student and want this first learning step to get done while having fun with no strings attached, you just recall what a number of former teachers and team members would say about your design, and include that if it was constructive and good. Put yourself in a context that helps your imagination spin, you might be one in a million watching the final result. Perhaps not very helpful, so focus on the goal that must be broken down into a number of steps to be achievable, and do what is most important given the information you have access to. Knowing or at least having a reasonable approximation of the S of your team member will tell you if you are just bothering a person you respect, or if your asking for feedback is a win-win relationship. Fast as a thought.

After an initial face-to-face meeting with the team members, where the team discuss the goal with the application, their requirements and wishes, the work collaboration can be done from their favorite working places via skype or some other remote meeting platform.

5.2.1 Teaching the ObjectModelDiagram notation to domain specialists and endusers

The DocumentDictionary application can be used as teaching example for the ObjectModelDiagram notation and how the DomainModel represents the structure of a working application program. The DomainModel for DocumentDictionary, the DocumentDictionary application and teaching material can be downloaded from the web-site. It is also provided in the open source team server whose virtual machine or installation image can be downloaded from the web site.

The domain specialists can then use the DocumentDictionary application to prepare a reading agenda for the other team members on books, manuals etc that provide a shared knowledge pool with terms and concepts that all team members must understand in order to communicate efficiently during the project.

The TimeToRead estimates provided by DocumentDictionary are helpful for giving each team member enough time to read, and prepare before the next DomainModeling seminar is scheduled.

5.2.2 DomainModeling seminars and seminar web reports

On a seminar, the seminar leader walks through the domain model from the root object and down. Explain and discuss the classes, their attributes and relationships. Take notes on the comments and discussions the team in the audience provides. The seminar leader or her/his secretary will use the notes to update the DomainModel, change names, structures, write in more detailed and explaining definitions of the model elements etc.

As soon as possible after a seminar, go through the notes, clarify and think about them, and then create a new version of the DomainModel, where the feedback from the seminar is entered.

Quality assure the DomainModel by running the automated checks on check level 3D Documentation.

Use OOCASE web report, to publish the DomainModel in a browsable HTML format.

Publish it on the web-server in `var/www/oocase/<domainModel>/<SemanticVersionNumber>/` and the team can find it from the web-servers home page. The team members study the new DomainModel version, take notes on mistakes, further suggestions for enhancements and questions they have for the next DomainModeling seminar.

The seminars go on until the team feels satisfied enough with the DomainModel to want to invest in a prototype implementation and evaluating it's performance by building real application specific models within the prototype.

5.2.3 Concurrent Work on the same DomainModel

If all team members are comfortable with working with OOCASE, the current edition of the DomainModel can be shared in a common datadictionary from which all members can load it, save a copy to a file, edit the copy and synchronize their latest edits with the shared edition of the DomainModel. To be practical and efficient, this concurrent engineering approach requires a division of responsibility in time, where only one team member at a time takes the author responsibility for a particular Module in the model.

Project information, such as schedules, responsibilities, different types of feedback etc can be customized in a Profile which is stored within the DomainModel itself.

Chapter 6 Source Code Generation

Delegating a laboursome limited task to the most trustworthy and efficient servants available, is a skill that develops with experience
[The Poet]

A source code generator converts a DomainModel into source code the way a Compiler converts a source code into executable machine code.

The input to this step is a DomainModel that has been checked to fulfill the ModelingRules for the particular target language(s) for the source code generation process. Support for automated checking is described in 12.1 Checking and Automated Checks.

Chapter 9 and 10 on page 61-76, 80-81 in [Johansson 1996] give an introduction to the high-level concepts and benefits of source code generation.

This chapter focuses on the more practical aspects.

6.1 Overall workflow

- 1) Check the DomainModel for well known errors. Automated checks are available for that.
- 2) Once the DomainModel is error free, save a copy of the DomainModel with a unique name, so you know the origin of the generated source code.
- 3) Generate the source code, or have it generated through a source code generation service.
- 4) Load the generated source code into your development environment, compile and test it.

6.2 OOCASE built-in Source Code Generators

The DomainModel Window in OOCASE provides a Tools->Generate menu. Here a target language can be selected, and then a generation definition chosen that contains specific parameters for the built-in source code generator.

One way to investigate these is to load a simple well known DomainModel, for example the DocumentDictionary domain model and just test to generate code, and inspect the generated code.

The generation definition files (suffixed by .def) are stored in the Generate/<language>/dm subdirectory in the OOCASE installation directory. Their structure is according to Chapter 8.

The menu Tools->Administration->Edit Generate Definition opens a generate definition file in a text editor.

6.3 Source Code Generation Services

This kind of service is typically handled by an implementation software specialist. The information necessary for ordering generated source code is the information contents of a DataDictionary (.opl file), a DomainModel (.odm file) and an identifier of the type of source code ordered.

6.4 MetaModelDatabase SQL based Source Code Generators

The principles for these are described in Chapter 9 and 10 in [Johansson 1996].

The data dictionary query program **ddq** takes an SQL-batch as standard input, and prints the generated source code to standard output: Example

```
ddq < generate/$(SOURCE_CODE_GENERATOR_FILENAME) > $(DIRECTORY)/$(SOURCE_CODE_FILENAME)
```

Before **ddq** is called, some environment variables have to be set, that tell **ddq** what ODBC data source to connect to, for executing the SQL-commands in the SQL-batch. The ODBC data source is the same DataDictionary database that you synchronize your DomainModel with. The environment variables are set in an init-script which is called before any source code is generated.

Typically the repeatable source code generation process of a complete application, or piece of application functionality, is automated in a batch file that calls **make** with different declared targets in the order they should be striped¹² to the output source code files.

The makefile may reside in the source code generation directory, or be available in a path to a make library directory that is set up in the init-script.

6.5 Organizing Source Code Generator Build Systems

The directory structure of a source code generation system for a particular application is organized into a source code generation directory (SCG-directory). The description below contains variables represented by \$(<variable name>), in line with standard unix shell scripts.

Examples of variable values:

TARGETLANGUAGE = (any of the below explained values, or a custom designations)

hlp : Help system source documentation files

lsp : AutoCAD AutoLISP source code

psql : PostgreSQL source code

sql : Microsoft SQLServer SQL source code

st : Smalltalk source code for the DomainModel object manipulation implementation

st_app : Smalltalk source code for the Application user interface classes and resources

```
make$( TARGETLANGUAGE ) /
```

¹² The concept of striping comes from parallel processing where a parameter-tuned task is allocated to individual processors that do the same task, with their specific parameters. In source code generation, typically each Class in the DomainModel that needs to have a class adapted implementation of a standard API, whose source code can be generated with the "algorithm" implemented in the source code generator, adds their band or stripe of compileable source code functionality to the source code file that after compilation by an optimizing compiler implements that class's specific behavior in machine code on the target hardware platform. The order is important due to dependencies that the target source code compiler may have. Some compilers fail if the stream of source code has not supplied them with all definitions they need to interpret new source code coming in from the sequential stream originating from the generated source code file.

```

init.bat # A batch file that initializes environment variables for the
        # DomainModel ODBC data source read by ddq and PATH to
        # a make library directory
init     # The Unix/Linux variant of the initialization shell script
generate/ # An optional directory for application specific customized
        # source code generators
build/   # A temporary directory for generated intermediate build scripts

# TARGET SOURCE CODE DIRECTORIES
# The following is for TARGETLANGUAGE = st
autocode/ # A target directory for generated loadable Smalltalk source code

# The following is for TARGETLANGUAGE = sql or psql
storproc/ # Generated SQL stored procedures
triggers/ # Generated SQL triggers
user_storproc/ # Manually written stored procedures, where some serve as
               # library or framework
# You may designate target directories for your own favorite languages

make_${TARGET_FEATURE}.bat # A batch files that calls make in a sequence
or                          # that builds the TARGET_FEATURE
build_${TARGET_FEATURE}.bat #

$(TARGET_FEATURE) batch files may be stored in a make library directory, made
accessible by a "set PATH=..." statement in the init.bat batch file.

```

The probably most productive way to get acquainted with the SQL-based source code generation method is to study a `$(TARGET_FEATURE)` batch file for a familiar `TARGET_FEATURE`, and by tracing the code, follow what it does and how it does it.

6.6 Using GIT for Source Code Generator Maintenance

GIT is a powerful directory structured source code version management system. Together with a suitable front-end such as TortoiseGit, it makes geographically distributed collaborative development and maintenance of source code generation libraries much easier.

6.7 Quality Assuring Source Code Generators with the Benchmark Domain Model

The benchmark DomainModel was designed to cover the design space of the primitives used for developing large scale product modeling systems. It is described in more detail in Chapter 13 of [Johansson 1996]. It is available as an `.odm` file in the OOCASE program distribution. If you develop source code generators, it can be used for testing all operations described in chapter 14 "Primitives of Domain Models" in [Johansson 1996].

6.8 Test Suites for Source Code Generators

When developing source code generators for a new language, it is convenient to use the benchmark DomainModel as test-application. The test suite for the new language should cover the full range of primitive operations on one selected instance of class, relationship type and attribute type of each unique variant that these may occur in the output designspace of the intended target language. The testsuites, once familiar with its implementation in one language, are then rather easy to migrate to another new target language.

Chapter 7 Prototype Iteration

Being a listened-to and influential participant in creating your shared future with others, is the best protection for its sustainability, since the genius of a learning and evolving team frequently excels the genius of an individual or a for granted taken cultural, traditional or bureaucratic past.

[The Poet]

A software application development project is an iterative process. The DomainModel is the Information model that specifies what information structures the application shall be able to serve the users with.

Since applications that deliver any added value compared to what already is available by standard applications without the need for a development project, such information models tend to be rather complex. It is very difficult for human beings to in their imagination in detail foresee what they like the application to do and serve them with. Thus developing a prototype system, that users can experiment with, and with the aid of the prototype formulate their requirements and wishes was observed to be a much more efficient way of delivering a product system the users wanted and adopted into their work procedures.

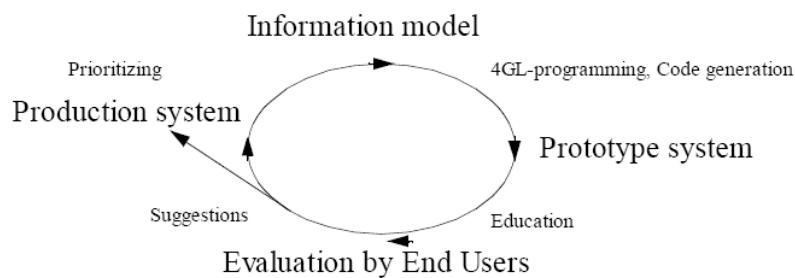


Figure 7.1 Iterative application development process

The purpose of building a prototype system is to receive feedback from end users. In most cases a new prototype must be delivered with an education for the end users who will evaluate it. Once they know how to operate the user interface of the application, they can start building their information structures and fill these with real business information. During that process they will discover if some information creation capabilities are missing in the DomainModel, if they like the user interface, and perhaps if they need convenience functionality, such as copying whole structures or alternative ways of having the information presented to them while pursuing their tasks etc.

The more feedback, the better the next prototype system can become. This off-course requires users to actively work with the prototype and record or explain their feedback so their suggestions can be collected and implemented in the next version of the DomainModel and user interface of the next prototype system.

If the amount of feedback is large, it might be helpful to organize it and present it to the whole group of end users and other project stakeholders, in order to under discussions identify what is most important in a priority order, and if there are conflicts of opinion.

If conflicts of opinion are NOT IDENTIFIED, discussed and resolved at a solution that all stakeholders can accept, it may lead to unnecessary iteration loops, where functionality is altered forth and back between the wishes of camps of conflicting opinions. Putting conflicting issues at the bottom of the priority list, allows resources to be put on delivering "most value first" when the next prototype system version is implemented.

Chapter 8 Configuration Files

*On some minor things, that allows growing value to be built upon it, someone has to decide.
Once the decision is made, everyone accepts it, and nobody cares since the problem is solved.
[The Poet]*

OOCASE uses configuration files for import, export, automated checks, code generation, interactive sql queries and user role configuration.

8.1 Directory Structure

The directory structure for configuration files follow this pattern:

<action>/<format>/<model class prefix>/

For example the export definitions for CSV text for DomainModels are stored in:
Export/Txt/dm/

8.2 Text format

The general format is close to windows .ini files with the following syntax:

```
[<section 1 name>]
<entry 1 name> = <entry 1 value>
<entry 2 name> = <entry 2 value>
...
[<section 2 name>]
...
```

The first line of a configuration file contains a heading that is displayed in the user interface as guidance for selecting a particular configuration.

When issuing an action in OOCASE, for example File->Export->Text in the DomainModel Window, all configuration files in the directory export/txt/dm are read, and the first line in each file is displayed in a dialog box for selection.

8.3 Macro expansion \$(<parameter name>)

To ease the maintenance of configuration files, a very simple form of macro expansion is available.

It follows the make file standard where a macro in the text is referenced with `$(<macro name>)` and expanded with the string contents of the macro.

8.3.1 System Macros

The following macros are assigned automatically when reading a configuration file:

8.3.1.1 `$(firstLine)`

This macro is replaced by the first text line in the configuration file. That is convenient for setting the title of interactive reports.

8.3.1.2 `$(firstLineBeforeColon)`

Is replaced by the text on the first line in the file up to the first appearing colon. This is used for identifiers, for instance the login identifier of an automated checker, to stamp as quality assurance mark on an object (if the checker had no complaints).

8.3.1.3 `$(filename)`

Is replaced by the name of the configuration file. Useful in copy-edit-test cycles with several alternative configuration files.

8.3.1.4 `$(filenameWithoutExtension)`

Is replaced by the name of the configuration file, except its extension which may be `.def` or `.mcd`. This can be used for example in:

[CONFIG]

REPORT_FILE = check\$(filenameWithoutExtension).txt

8.3.2 Selection Macros

Some configuration supported operations such as the Report -> Query -> Run SQL Query..., take objects that are interactively selected in the user interface as parameter input.

Attribute values of selected objects can be accessed in the configuration file with:

`$<attributeName><selectionIndex>`

for example:

`$HighId1`

`$LowId1`

where `<selectionIndex>` represent the position in the current selection. The first object has selection receives index 1, the second 2 etc. Thus preconfigured interactive SQL queries to the meta model database can be parameterised with attribute values from the current selection.

Chapter 9 Import of Domain Models

*If you carry information in a language I don't speak,
and I have within translation, your value is mine to keep.
[The Poet]*

Authors comment: Well you annoying genius, you have to study the copyright laws before your all-knowing wisdom hit's the ground on this planet and destroys our economy.

*True Life is given solely, to the ones who live in trust.
If life requires honesty , so be it, since it must.
[The Poet]*

Hrm, the following sections describe various methods and formats to import DomainModels into OOCASE. These have primarily been chosen by needs within projects that funded the development of OOCASE. Thus the import functionality shall be regarded as a preview, where real paying customer project needs has and will fund the development of the generic functionality that is present and will be added into the product for all tool customers to use.

The way distributed parallel software development life all over the planet works today, is that frequently the knowledge barrier and budget of a development team limits the options for what standards and platform product supplier dialects they can support.

The teachings of the advocates of "customer lock-in" economy's "historically momentary" monopoly economic benefits, off course has delivered a plethora of diversification of syntax and mechanisms for communicating the same semantic information in a, as big plethora of variations as the singing-variants of birds and wing patterns of butterflies. This off-course is no help to the united human manufactured organism to fix global warming.

However the laws of life apparently governs uncoordinated parallel processes, so we need to ensure we massproduce knowledgeable solvent customers who can pay for the efforts of with science outlining and then with science hammer out the shape and implementation of this high-level efficient self-sustaining organism whose goal is to keep our planet at a medium temperature that serves us as human beings and ensure our quality of life, the way the best knowledge on this planet can guide us, if we give it efficient communication capabilities, and ensure there are enough people who can make good use of that information in actions delivering implementation.

9.1 Import ODBC

The Microsoft Open Database Connectivity (ODBC) interface is a C programming language interface that makes it possible for applications to access data from a variety of database management systems (DBMSs).

ODBC is a low-level, high-performance interface that is designed specifically for relational data stores. OOCASE uses these API calls to extract database schema information from an ODBC data source on the same computer.

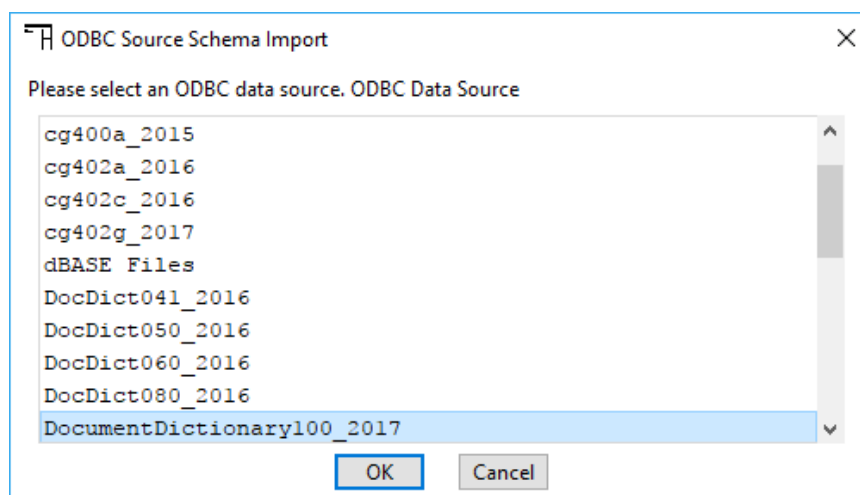
In a Windows operating system you can administrate ODBC data sources with the program `odbcad32.exe`.

9.1.1 Import ODBC Source Schema

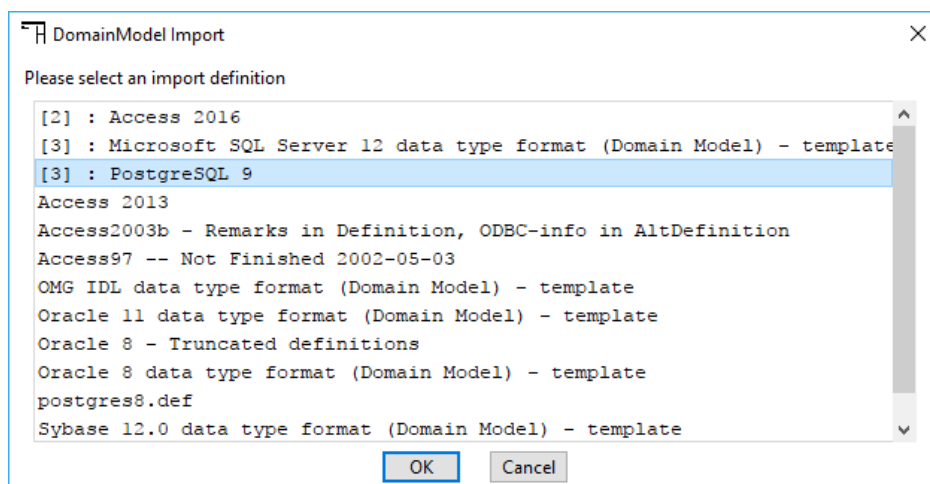
In the DomainModel Browser issue File->Import->ODBC->ODBC Source Schema.

A dialog box appears with a list of available ODBC data sources in the logged in users operating system environment, as reported by standard ODBC API calls.

Select the ODBC source specified on you local computer that you want to import.



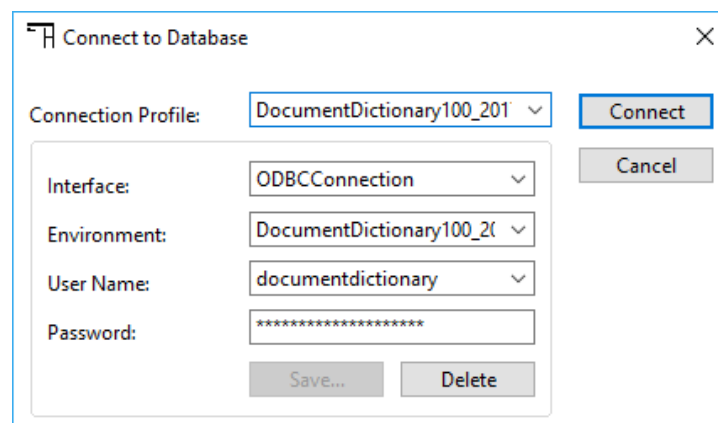
After the odbc data source is selected, with the current "improveable " implementation, you have to select the import definition that adapts the ODBC channelled SQL queries to the language dialects and capabilities of the specific data source.



The choices are ordered according to the level of quality assurance that has been applied on the configurations before release of the running OOCASE implementation, where [1] means that it has

been tested without delivering any errors or warnings on at least one data source, [2] it has been tested on several different datasources, [3] it has been tested on DocumentDictionary, OOCASE and the Benchmark application, [4] It has been tested against a defined and documented designspace, [5] it has been tested and documented on a level that a very experienced and resourceful manager has approved it and will take responsibility and action if something goes wrong. The status of import definitions without a [n] quality assurance marker have served their purpose in earlier releases of OOCASE, but their quality state has not been determined on the current release.

After the odbc data source is selected, a login dialog box appears where you have to provide the name of the data source in the Environment field (again) and enter a User Name and Password.



After that the built-in import procedures take over and do their standard ODBC-calls and from the through the configuration gathered information produce a DomainModel with classes and attributes that represent the tables and columns within the database. The import can serve as an initial working template for a well documented DomainModel, that may be reimplemented on a new platform, or just serve as documentation or decision support for the customer.

9.1.2 Add ODBC Source Statistics

This command will submit SQL-queries to the selected ODBC source to add measurements of the statistics from an ODBC imported DomainModel as described in Chapter 4 of [Johansson 1996].

Classes: *avgCardinality*

Attributes: *probabilityClear*, *avgSize*, *stdSize*

Relationships: *probabilityClear1to2*, *avgCardinality1to2*, *stdCardinality1to2*

Due to the plentiful dialectic variants of SQL only a few are fully supported where some are restricted to only measure *avgCardinality* of each class/table in the ODBC source, which is the number of rows of a single measured ODBC source. The implementation currently follows the "just-do-it" implementation method that frequently is the most efficient to solve a particular urgent payed for need.

The measurement results can be visualized in the user interface of the DomainModelWindow using the menu command View->Classes->Class Statistics, and View->Relationship->Statistics, View->Attributes->Attribute Statistics. When these view modes are set, the information appears on DomainModelDiagrams, that can be useful for brief decision support while going through the contents of databases.

A scan of more detailed statistics from the ODBC source, that may improve the understanding of the data which a new DomainModel should accommodate can be made by studying the measurement's recorded in the Profile¹³ "ODBCSourceStatistics".

This data is visible in the GenericObjectEditor's that are enabled by View->Set Default Object Editor Class->GenericModelElement.

The Profile attributes are distinguishable in the AttributeValues listbox of this ObjectEditor, with lowercase first letters. Look for profile attributes prefixed with max, min, avg and std.

A sample set of the 30 most frequently occurring values of an attribute can be seen in the Attribute.attributeValues listbox by selecting the lowercase Profile Relationship 'attributeValues' in the Relationships listbox in Figure 4.

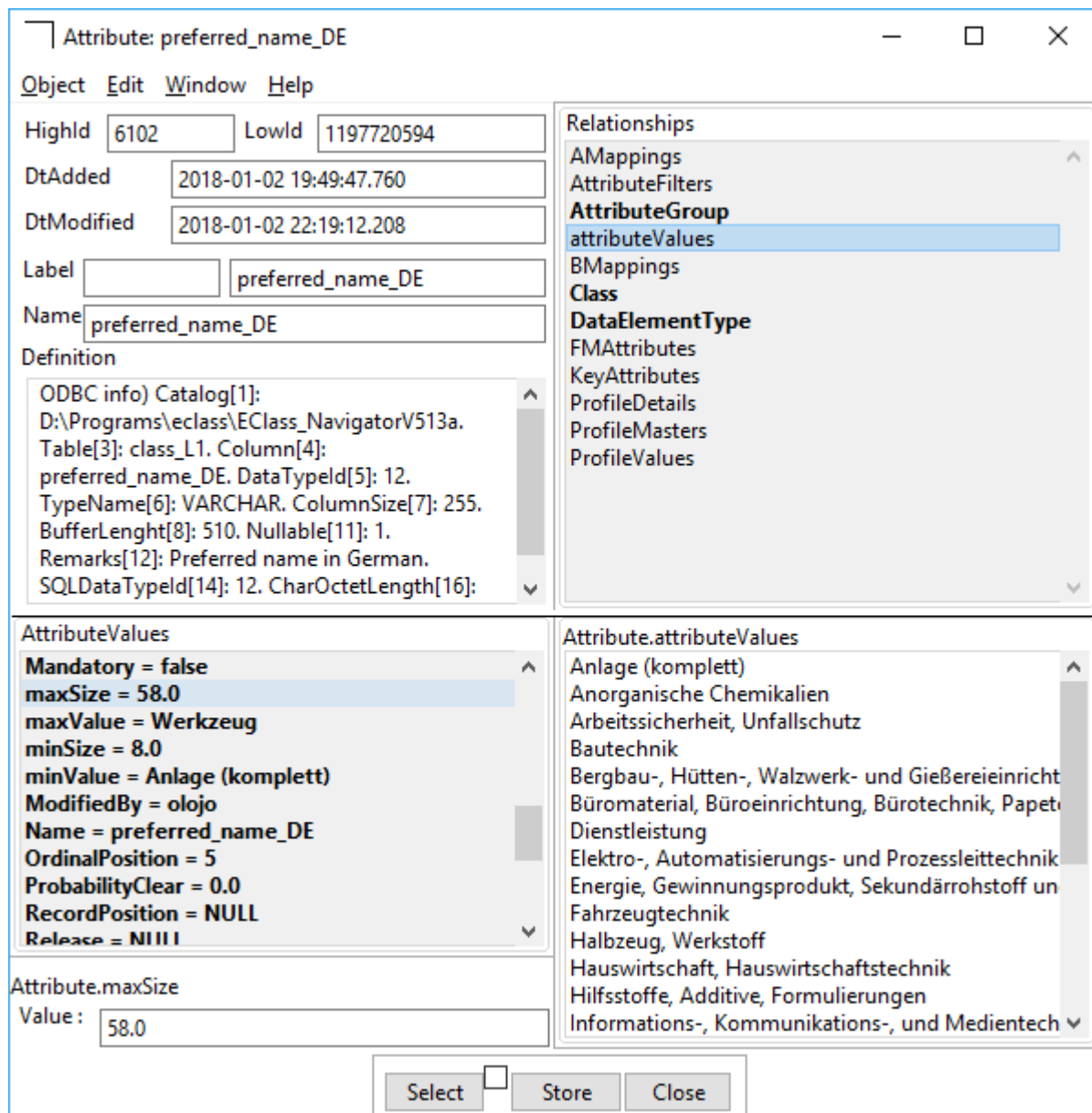


Figure 4. GenericObjectEditor displaying statistics and samples from an ODBC data source.

¹³ Profiles are described in Chapter 17 "Profile Extensions of the MetaModel" starting on page 77.

For more thorough statistical analysis of larger databases, table specific stored procedures were generated with declarative SQL based source code generators which were compiled in to the database server itself and delivered significant better execution performance that could be scheduled for weekend batch runs, or executed on separate analysis servers that were feed with automated backups from the production system.

Interactive reports from such analyses can be distributed in compiled aggregated OLAP (On Line Analytical Processing) tools and databases, where Microsoft Access was a useful distribution format decades ago and still is for a reason.

9.2 Import TEXT

File->Import->Text, enables import of text data arranged in a standard comma, or tab separated table text format, with one text file for each class of objects in OOCASE, e.g. domainmodel.txt, class.txt, attribute.txt etc.

For each such file, e.g. class.txt, each row in this text file corresponds to one instance of a class in OOCASE. Each value in a column in the table corresponds to a particular attribute's value for the class instantiated on the particular row. The imported text files may have column headings, that determine a mapping of values' positions to attributes.

A text import is configured with an import definition, following the syntax of Chapter 8. In those files, each class in a DomainModel has a corresponding section, that lists what attributes are expected to appear in the text file, and what column headings they are mapped to in the file to be imported.

To configure a new type of text import, first use File->Import->Create Text Import Definition. A number of questions will be asked about the format of the text data to be imported, and a default import configuration generated from the metamodel of the type of model to import, e.g. a DataDictionary or a DomainModel. This configuration can then be adjusted in a text editor. It's [CONFIG] section provides some general configuration settings.

Each metaclass in the OOCASE DomainModel has a separate section in the import definition file that lists all available attributes within that metaclass, prefixed by "att" or "rel". The "rel" prefixed columns are pairs of *relational database foreign key attributes* that serve as a (rel<name2to1>HighId, rel<name2to1>LowId) pointer to a row identified by the corresponding *relational database primary key* (attHighId, attLowId) in another table and thus implements a relationship between individual rows in the imported model.

If the need to import text occurs, it is easiest to study some examples of the import configuration files stored in Import\Txt\Dm.

9.2.1 Model Migration between versions of OOCASE

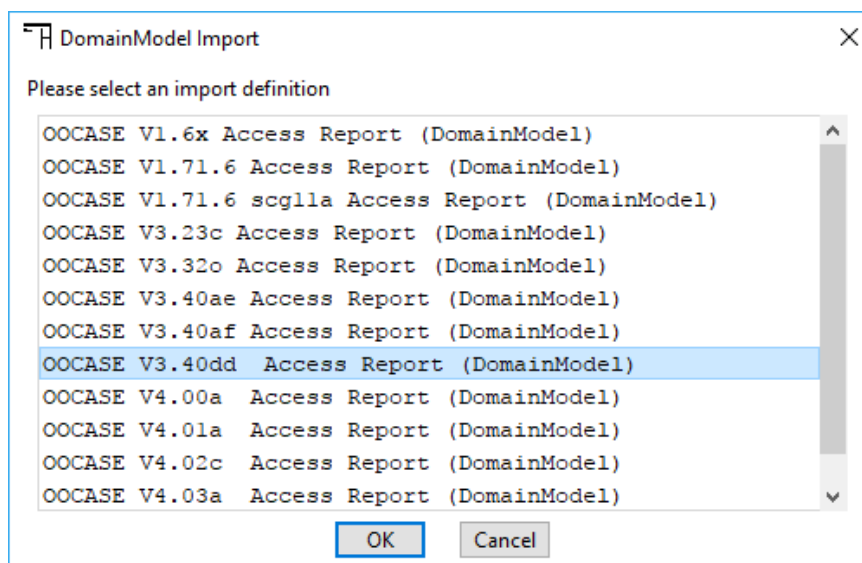
For migration of OOCASE models to new versions of OOCASE, proceed as follows:

- 1) Start the old version of OOCASE and load the model(s) you want to migrate, e.g. a DataDictionary and a DomainModel.
- 2) Export the model(s) in text format into a separate directory for each model. A practical directory location convention for migrated models is to create a directory Report/dd/<DataDictionary.name> or Report/dm/<DomainModel.name>. Use File->Export->Text, select the latest OOCASE export definition, create the new directory and press Choose. Repeat until all models are exported into their own directories.
- 3) Start the new version of OOCASE. Use File->Import->Text, Select the Import Definition of the previous version of OOCASE, select the directory of a previously exported model. Save the model with a new name in the samples/dd/ or samples/dm directory, perhaps under a subdirectory named after the previous

version. If there are errors reported during the import where the cause is not a mistake, a quick way to fix them is frequently to edit the <class>.txt files directly.

- 4) Use Tools->Import->Delete empty string default values. This will replace all imported empty string attribute values with NULL values in the model. NULL values have no representation in universally exchangeable standard tab-separated text files, and are thus represented with empty strings in these. Since an imported model usually never has been synchronized with a metadatabase repository of the new version of OOCASE before, it is recommended to preserve the dtModified timestamps to preserve the timestamp when information of an object was actually changed.
- 5) Save the model.
- 6) Perhaps run File->Quality Assurance ¹⁴->Check Model, to determine the quality of the imported model.

Most releases of OOCASE have predefined import text definitions from previous releases of OOCASE. Thus File->Import will display import definitions like this:



9.2.2 Model Migration from Legacy Software Tools

The lifecycles of computer hardware and software tick at a much higher rate than the lifecycles of expensive large scale infrastructure. Thus migrating information about the latter from a previous generation IT infrastructure to a new BETTER¹⁵ IT-infrastructure, is significantly easier if conducted through an intermediate format that has a from decades of practical reality battered but still standing scientific core.

Transforming legacy IT software models to the OOCASE text format as an intermediary for building the new software or migrating the information to a new IT-system¹⁶ is much easier since

¹⁴ The Quality Assurance principles acquired from best practices on decade(s) lifetime products within Mechanical Engineering Industry and adapted to Software Engineering are described in "Chapter 11 Quality Assurance, Version and Release Management" starting on page 55.

¹⁵ This has with available and in practice implementable IT-solution options to do and external environmental pressure that may derive it's pressuring goals and methods from sources captured below the knowledge barrier of experienced managers who take their duty seriously, keeping the large context value adding production capability at prime in a difficult environment they for shure like to improve.

¹⁶ A new IT-system should be decade(s) robust and well supported.

simple human readable text files are supported on all new BETTER computer and software platforms.

The concepts in the OOCASE DomainModel for structuring information have been discovered and rediscovered in all locations where there was a developing software industry, however with different environmentally adapted terminology.

Most information stores developed by people whose thinking has been shaped by education from state-of-the-art computer science and state-of-the-art long-term successful software industry evolving under the expansion phase enabled by the mass market for personal computers, can easily be transformed to an OOCASE model, perhaps with some Profile providing additional attributes and relationships.

Simple text files and tables have survived longer than any complex structured in a single file representable universal data exchange syntax format, since the latter requires a complex piece of reading software to be ported to the new platform first.

Exporting and importing information models through various declarative- or perhaps procedural programming implementations to a named column flat table text import format is comparatively easy. OOCASE can handle most of those CVS or TAB separated table export formats, that use standard byte sized character sets. In such a format, the characters that represent a column value separation, e.g. a TAB character, and row separation e.g. a CR or LF, or CRLF, must be uniquely used for that purpose only. Thus some exports where column values contain flowing text with tabs and carriage returns or line feed characters, need to be preprocessed by a one-to-one substitution mapping of ascii control characters, to uniquely identifiable string tokens, e.g. that a tab character (ASCII 9) is replaced by the standard string token `	`. Those mappings are configured in the [CONFIG] section of an import- or export configuration.

Once the exported data files are present in a directory, a text import can begin.

In OOCASE use the File->Import->Guess Import Definition. Select an existing import definition that matches the exported format as close as possible. OOCASE will then compare the column data from the Legacy System export and create an import definition from that.

Use File->Import->Text, and select the newly created import definition, choose the import directory that contains the text files and the model will be imported. Diagnostics on the import process are written to a file named import.err, which in case the import generated errors or warnings can be inspected at once.

In case the import needs to be adjusted, edit the Import/Txt/(dd|dm)/\$(filename).def import specification and retry the import until it works.

After the import is completed the new model file is saved in its binary format (.det or .odm). Use Tools->Import->Delete empty string default values, according to 4) above.

9.3 XML

The Extensible Markup Language (XML) is a subset of SGML¹⁷ that is completely described in [XML 2008]. This, on the web free for download specification, is a HTML-page that when printed fits on 57 A4 pages. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Due to the need for exchange of complex structured information in a standard single file format, the presence and superior availability of this standard in conjunction with a mass of free or inexpensive tools for parsing and unparsing, made XML a popular widely used standard for all kinds of structured information exchange.

XML has inherited some theoretical fussiness from SGML, for example the undecidability of whether to use an XML-element, e.g., `<Class><HighId>1000</HighId></Class>` for representing a class object with one HighId attribute value, or if to use an XML-attribute `<Class HighId="1000"></Class>`. This opens up a redundant designspace of 2^n combinatorial ways of representing an element of an object of a class that contains n attributes.

Once this lack of determinism of syntax for the same semantics of in practice used XML exchange started to generate significant problems, this was addressed by the w3c and later on compensated by XML Schema, later called XSD, which is a kind of more precise Document Type Definition (DTD)¹⁸ against which an XML file can be validated. An XML schema serves as a more precise specification of the expected structure of the transfer format, and can be used to report errors if the structure in an XML file does not correspond to the structure specified by the XML Schema document.

9.3.1 Import XML

The preconfigured XML import functionality is not part of the OOCASE product since most XML imports require some manual services to adapt to a particular need. It is there as an indicator that it is possible to import XML data and transform it into something that looks like a DomainModel. The present import XML functionality has been used in a number of funded projects that delivered decision support to many decision makers in the form of structured numeric and statistic information. Many times in the form of high-performance interactively navigable and searchable decision support applications, where Microsoft Access databases were a handy distributable format when the data volume was not too large.

The possibility to lay out the information structure of an XML document graphically on an ObjectModelDiagram while displaying statistics about the underlying data volumes in classes, relationships and attributes was indispensable when some projects hit the technological scope of applicability on some technology platform that had to be migrated to BETTER database technology in order to serve its expanding application requirements.

¹⁷ SGML, or ISO 8879:1986, "Information processing — Text and office systems — Standard Generalized Markup Language (SGML)", www.iso.org, is a 155 page syntax specification of a textual markup language whose (somewhat theoretically unpure non-orthogonal) expressive structuring design space and syntax elements still has a significant legacy impact on text based exchange formats today. The purpose of SGML was having a printing format layout independent language for the text of books and documents that authors could read and write in the text editor tools available up to the first half of the 1980's. The key experience of the horrible inefficient physical paper format's for maintaining and distributing the LAW, was probably a trigger to this great advancement on electronic distribution and maintenance of large volumes of important texts. Leading to enormous productivity gains in the publishing industry with impact on society that depend severely on the semantic content of published texts. The reference for SGML is [Goldfarb 1990].

¹⁸ DTD, a document type definition for SGML documents, which is a structured text document defining syntax rules for how to express concepts that follows the discovered and rediscovered meta model pattern of Class, Attribute, and binary Relationship. And model pattern of object/instance, attribute value and pointer/foreign key/link relationship. In a SGML DTD the meta model pattern class level named ELEMENT, ATTLIST and LINK. And model pattern of element, attribute (in the attlist), and special key attributes named IDREF, and `#<idref-value>` implementing pointers to other key places in the document. Note that the ENTITY concept used in DTD's is macro expansion, and even if frequently used to implement the declaration functionality of a Class, a DTD ENTITY is a name for a template of text.

The Import XML functionality shows that it is possible to reverse engineer meta data structures out of the exponentially explosive redundant design space of XML file representations for the same information content. Analysis of such structures may deliver useful background information on structure and terminology requirements and size measurement data to a design team that wants to develop or discover a, with measurable science supported optimal, DomainModel for a non-redundant design space for structured information on the knowledge level¹⁹.

If the need arise to import metamodels or reverse engineer the information structure out of the contents of reasonably sized XML files, the in OOCASE implemented rule-based "just-do-it"-method was used and delivered a practically useful method for limited sizes of schema design space and data volumes.

¹⁹ The **knowledge level** is the abstract level above the lower symbolic level, named and made consciously aware of by publications of Allen Newell in the Computer Science community, and shortly described in [Johansson 1996]. (The lower symbolic level is fequently with mockery nicknamed the syntactic sugar level when it has to include structures. The mockery arises from cause of the for many of the mockers unknown inadequate logistics and speed of biological hardware building of representations of adequeate, at the time being best available scientific knowledge available, in the creators of the syntactic sugar formats, that are optimal with regards to use and learning within the biological hardware of a human brain given a mass market that requires a standard to build a uninteresting but important platform to, with scientifically proveable efficiency, share higher much more value adding information and knowledge exchange upon). The knowledge level is represented in the human brain with for an individual ideosyncratically developed spatio-temporal activation patterin in the neurons of the brain, that allow the prescence of a evolved concept of a symbol focused biologically wired indexed access to other by biological chemical programming grown biological hardware that physically transfers signal pattern of associated concepts of symbols to enter the consciousness, thus giving an instant in itself representation of the concept of a symbol full access to all by learning created concepts that by tilt of learned patterns-of-thinking in parallel operating overlayed spatiotemporal patterns guide the line of thought at a cycling speed of 4-7 Hz, through the core of the brainstem (that may trigger some important chemical mechanisms), limited by the speed of signal transmission in the neural dendrite and synapse communication paths that depends on the prescence of more expensive chemical "turbo molecules" that may increase the cycling speed up to 14 Hz at an intense adrenaline or "insight-arousal" rush.

A **knowledge level**, which off-course, since we are limited by our sensory interfaces and languages and what computer symbol structures or human spatiotemporal neural patterns they require to be represented when digesting input from the sensory system, needs some symbolic representation to be communicated between humans or machines. Thus some cleverly designed syntax, or for the practical use efficient interaction tools are needed to enable efficient communication of information.

Now the waste of potentially value adding brain-cycles that could be contributing to our shared increase in quality of life, given a proper information distribution logistic free from "uneducated destructive self-interest" that just delivers a sprawl of habitats for the powerful "ants" that know what they know, while we need a foundation that enables the growth of higher level abstractions that are powerful enough to tell us how we shall fix the real problems of our species and it's limited living quarters, makes a scientist angry enough to deliver a verly long footnote that probably nobody will read, so aggression is a bad emotional index that usually just delivers a by biology promoted puff that will go into nothingness, unless someone with in practice useful influential powers takes a note and acts in the right time and place.

Note: This footnote was temporarily emotionally sponsored by some, in afterthought enlightened, over the authors conduct perhaps powerful but influential anomonus people.

However life stands on the present widely distributed infrastructure however inefficient it's architecture is, so to get something done one has to use what is present and available if it delivers any added value. Where the understanding of what added value really is comes from a place above the knowledge barrier of most humans living today, unless we fix the logistics involved in the education problem.

The import rules are derived automatically from the contents of the XML-file, however the user has to configure where different element and attribute data shall be stored in an OOCASE model. The details are described further in section "9.4 Create XML Import Definition" on page 49.

There are some examples of import rule specifications in .xml format in the Import/xml/<prefix>/ directories. The XML root element in these files is <ImportRuleSpecification> and can be navigated with a standard XML or HTML browser that supports .xml files.

9.3.2 Import XML DTD

This feature imports the meta structural constructs of a DTD which are element declarations identified with the <!ELEMENT ... > syntax, and attribute (definition) list declarations identified with the <!ATTLIST ...> syntax, and transform them into a corresponding DomainModel representation in the form of Classes and Attributes.

In the SGML terminology [Goldfarb 1990]²⁰, what constitutes a DomainModel Class is collected together by a generic identifier (GI) that serves as a name for a meta object that ties the in the DTD syntax's perhaps scatterable definitions of ELEMENT and ATTLIST together into a class.

An element declaration is mapped to a Class, the attributes defined in an ATTLIST type are created below the class mapped for the ELEMENT type with the same name as the ATTLIST type. Declared owned elements in the structure definition of an element are mapped to relationships between the classes for the corresponding ELEMENT types.

The default naming convention for the created relationships that are defined to appear as owned by a particular element in an XML file "(class1.name), '_owned', (class2.name), 's'". This can be adjusted in the import configuration under the [CONFIG] section.

All relationships are given the one-to-many type (1-N), despite some derivable but not extensive for the given design space available syntactical constructs for delivering cardinality constraints and other for a declarative query language non-value adding constructs in the standardized SGML language, with regards to value-adding information delivery and retrieval application usage.

A workflow for importing a simple DTD is:

- 1) Select File -> Import -> XML -> XML DTD
- 2) Select an import configuration that matches the type of DTD you want to import.
- 3) Select a dtd file to import.
- 4) Use Tools->Utility->DomainModel->Create Main ObjectModelDiagram
- 5) Select and right-click on the above created Module named Main, and select "Open Diagram"
- 6) Sketch out an ObjectModelDiagram by selecting all Classes and Relationships in the DomainModel Window and drag-drop them onto the ObjectModelDiagram. Arrange the layout so it becomes comprehensible to a human being for some particular purpose.

If you tried this out on a non-simple DTD, you understand that we need to reuse some knowledge from engineering disciplines like VLSI design and PCB layout design to get any aid from a visual 2-dimensional representation of the design space of more complex grammars.

²⁰ *Publishers Note:* The lengthy footnote was abstracted to save manual page space: On the knowledge level a DTD, a reverse Engineered Meta-Model of samples of XML files, a DomainModel and a pure scientific essence extracting infological model describe the same concepts, however the quality and practical value of the different symbolic representation languages differ severely along an orthogonal coordinate system of several different evaluation criteria. The author was kindly asked to publish the original footnote in a scientific paper instead.

The command "Import XML DTD" also serves a tool for building a skeleton for adding XML Import Rules.

If you want to create XML import rules or statistically analyze XML files whose structure are specified in a DTD, importing the DTD first before using "9.4.3 Add XML Import Definition Rules from Sample XML file" will make the declarative DomainModel expose the whole design space of the DTD.

This allows you to see which features allowed by the DTD that are not used by the XML-data in the statistically analyzed XML files.

You have to create XML Import Rules manually for those features in the DTD that lack examples in an instantiated XML file. Or you can create a design space spanning XML file yourself, and import it with 9.4.3 to have the rule templates created for you. See also 9.4.6 below.

9.3.3 Post process imported CMOF model

This functionality is obsolete, but may still be required by some installations.

9.3.4 Post process imported UML 2.5 model

This functionality allows the user to transform an XML import of the XMI distribution format for UML 2.5 into OOCASE for analysis and perhaps reuse of now widely known and used standard design patterns for software modeling in their own products. The full modeling capability of UML is overkill for the comparatively simple basic technical information system design and maintenance purposes that OOCASE has been optimized for over two decades.

Very much of the modeling capability of UML addresses software behavior.

Technical information systems is a mature and well understood discipline by the specialists who work in this area. A huge amount of standard information processing behavior can efficiently be directly inferred from the structural information content of a DomainModel. Thus there is no need to specify that kind of behavior in a software model, since it can be generated and documented automatically from the information declared in the DomainModel.

Powerful declarative behaviour specification capabilities such as UML StateMachines are outside the scope of applicability for DataDictionary and DomainModels in OOCASE.

9.3.5 Import XML Meta Model

This functionality is obsolete, but may still be required by some installations.

9.4 Create XML Import Definition

Due to the size of certain XML data sets under analysis, the approach was reengineered from using an in-main-memory DOM²¹ representation for a whole XML file to an immediate SAX parsing of larger files, while collecting statistics and samples of XML data that is stored in a Profile within the DomainModel.

9.4.1 Create XML Import Definition

This function copies a default XML-import definition template (default.def.template) and allows the user to adapt the configuration for a particular XML import need.

²¹ Document Object Model (DOM), see <https://www.w3.org/DOM> for the evolution history of this public open standard.

9.4.2 Edit XML Import Definition

This is a convenience function that lists the first line in all available XML import definitions in the `.\Import\xml\(\dd|dm)\` directory. It saves some time to locate and open the configuration file with a standard operating system file browser.

9.4.3 Add XML Import Definition Rules from Sample XML file

This function parses an XML file and adds Classes, Attributes and Relationships to the DomainModel that represent the occurring XML Elements and their attributes and the hierarchical element structure deriveable from the XML file content. It also adds raw data for computing statistics and a limited sample of attribute values and element text occurring in the XML file and stores these below each Class representing an element and Attribute representing an XML attribute. This data collection allows the DomainModel designer quick access to fact based decision support when choosing DataElementTypes for attributes etc.

A typical workflow²² for analyzing the statistics derivable from an XML file is:

- 1) In the DomainModel window create an empty DomainModel and save it.
- 2) Use File->Import->Create XML Import Definition->Add XML Import Rules from Sample XML File
 - 2a) Select "XML Add Rules and Statistics DTD04d Import Configuration"
 - 2b) Select the XML file to analyze
- 3) Use Tools->Utility->DomainModel->Compute Class Ranks
 - 3a) Press Yes on setting `genSqlFlag = true` on all classes.
 - 3b) Verify that the XML root element class (e.g. `xmi:XMI`) has class rank 1 in the report that appears.
- 4) Use Tools->Utility->DomainModel->Create Main ObjectModelDiagram
- 5) Select and right-click on the with 4) created Module named Main, and select "Open Diagram"
- 6) In the opened ObjectModelDiagram window select View->Statistics
- 7) Select Layout -> Auto Route -> Whole DomainModel from scratch
- 8) Select Window->Zoom Window
 - 8a) Study the meta-model structure of the XML file
 - 8b) Pan the view in the ObjectModelDiagram by left-mouseclick+drag in the ZoomWindow
 - 8c) Study the statistics for Classes, Relationships and Attributes presented in the ObjectModelDiagram
- 9) If the XML file has a deep and complex meta model, use File -> Page Setup -> A-2L to increase the size of the Drawing Frame and diagram, so you can zoom in on any place in a larger meta-model structure.

9.4.4 Clear statistics from sample XML files

This command is helpful to reset accumulated statistics stored in "cardinality", "avg"- and "std"- prefixed attributes on Classes, Relationships, Attributes and OwnedElementRule Profile Objects, after having analyzed a set of XML files with "9.4.3 Add XML Import Definition Rules from Sample XML file".

The statistics can be computed for an alternative set of XML files, while keeping the recorded total use of the design space in the same DomainModel.

²² The ambition is to maintain software release adapted workflow descriptions on this level of detail in the help system of the software. This detailed work flow example is provided in the User Manual to give new professional users a deep enough theoretic understanding to get started with practical tests on their own.

9.4.5 Truncate statistics from sample XML files

Variants of the above command reduce the amount of ProfileObjects and ProfileValues that record collected grouped and aggregated attribute value sample data counts and statistics from the imported XML files in the DomainModel.

Examples are AttributeValue profile objects that store value and the occurrenceCount for that value below each attribute.

Quick access to data in the DomainModel that document example data of what values actually are stored in different attributes is very useful when evaluating and deciding what implementation to use for a reimplementing of an information system on a new target platform.

However too much such statistics can make the DomainModel impractical or inefficient to work with, so this command can make it "lighter".

9.4.6 Edit XML Import Rule Definition from DTD Profiled Domain Model

Opens a Profile Specific Object Editor for configuring XML Import Rules with the aid of a hierarchical tree browser.

9.4.7 Write XML Import Rule Definition from DTD Profiled Domain Model

Writes the XML Import Rules configured with "9.4.6 Edit XML Import Rule Definition from DTD Profiled Domain Model" to a configuration xml file that can be reused by many different XML import definitions.

An XML import definition specifies which import rules should be applied in:

[CONFIG]

IMPORT_RULE_SPECIFICATION_FILENAME = < xml file that contains the import rules>

The rules are used by the internal SAX-parser to, based on the element or attribute that appears in the imported XML stream, call configured methods that create objects in the model and set attribute values on these created objects.

Chapter 10 Export of Domain Models

<Some poetry once [The Poet] has read the chapter and condensed its essence>

<Some author comments on the poetry>

Information in its essence is immaterial, however needs some representation to be worked with or serve its purpose. There are plenty of tools with various functionalities that can be used for some job that has to be done on an information model. Thus the ability for a tool to export information into a number of different formats that serve different purposes is fundamental for delivering added value to a team.

A team that is skilled and free to use the best available tools given their resource constraints to reach the goal set out in their project. Where OOCASE focuses on information systems that carry information for decades in a continuously changing environment.

10.1 Binary Storage Formats

Binary storage formats are optimized for performance for a particular well understood purpose.

The DomainModel binary export formats below are examples that are optimized for various purposes that seem to be generic for a long time ahead.

10.1.1 AutoMetaStorage

This binary representation is saved in the same optimized binary format that the software frameworks that OOCASE and DocumentDictionary are built on save their models in.

The DomainModel or MetaModel of the AutoMetaStorage binary format carries the generic core object-oriented information essence that is represented in infological theory, OOCASE DomainModels, OOCASE Profiles, UML MOF, and the design of a large range of single and multiple inheritance capable object-oriented programming languages.

One human representation syntax of this binary format is available in the DomainModel named "MetaDomainModel<3 digit Semver 2.0 identifier without dots>_R<Release identifier with dots>.odm" in the samples/dm/ directory.

10.1.2 C++ AutoMetaStorage

This binary representation was tuned for a framework developed in C++, for use in a number of applications where the target platform is implemented in C++ and built on a set of de-facto international standard frameworks available for C++.

10.2 TEXT

Exports configurable views of the DomainModel in various tabular text formats, where each object in the model gets a corresponding row in a table where the columns in that row contain textual representations for the objects' attribute values.

The theory behind these formats was explained in "9.2 Import TEXT".

10.3 SQL

Similar functionality as in "10.2 TEXT" however the whole configured export goes into one SQL batch file that contains SQL insert statements. Executing this exported sql file on the target database will insert the exported model into pre-existing SQL-tables with the table names and column names specified in the export definition.

This is useful for information distribution through an existing database infrastructure with a DomainModel conformant database schema, or for exporting domain models to relational databases dedicated to various types of source code generation.

It also provides a textual file format work-around in case OOCASE can not be directly connected to a database repository. This may be for security reasons, present organizational firewall structures, lack of compatible call level communication libraries between the client and server operating systems and database platforms etc.

In most such cases an encrypted transfer of a zip-compressed exported OOCASE sql-batch will solve any immediate needs.

10.4 XML

Allows exports of OOCASE model objects in XML format.

The configuration possibilities are limited to direct one-to-one mapping between MetaModel Classes in the OOCASE DomainModel and XML elements, attributes and element hierarchy structure levels.

More specifically with one example:

The OOCASE MetaModel Class "Class" maps one-to-one to ONE XML element name. That means you can not map a "Class" instance to different entity names in the exported XML file depending on other properties of that particular "Class" instance. For example the OOCASE "Class" is used to represent both "UML Class" and "UML Association" in a UML conformant DomainModel, but these are distinguished by the UML profile specific attribute value `xmi:type="uml:Class"` and `xmi:type="uml:Association"`.

Such XML file transformations of entity names etc can be done in an XSLT post-processing step.

Owned "Relationships" are represented via XML hierarchical element structure.

Referenced "Relationships" are represented with foreign key references by actual foreign key values such as a pair of `<<name2to1>HighId>` and `<<name2to1>LowId>` elements, where `<name1to2>` means the OOCASE Relationship attribute `name1to2`.

Alternatively referenced "Relationships" are implemented with an XML file uniquely generated id attribute on each element, which is (foreign-key/pointer) referenced by a `ref` attribute in an element representing the relationship link.

Configuring an XML export requires experiments with real example files to figure out and become clear about all the details about meta level and instance level etc. It is recommended to start with a very simple DomainModel example from a domain where class names on the meta and instance level do not overlap, and explore the export configuration design space with that "reference DomainModel".

10.5 XML DTD

Allows export of a DomainModel in an XML DTD format. See "9.3.2 Import XML DTD" for details on this format.

10.6 XMI

XMI is the OMG XML MetaData Interchange format.

These export functions are "preview" and unsupported.

Chapter 11 Quality Assurance, Version and Release Management

Quality Assurance carries an Eternal Truth whatever primitive methods used to enforce it
[The Poet]

This chapter goes through the theoretical foundation of ONE mechanism that allows creative incremental development to co-exist with the rigidity necessary to efficiently manage information distribution in a massive parallel distributed supply chain network, where many actors are not aware of each others existence. We name it QAEVR, for Quality Assurance with Editions, Versions and Releases, spoken Q-A-ever.

The reason this ONE mechanism was chosen, is that it is possible to efficiently implement it on the presently widely available theoretical core of the relational database platform, while it is much more efficiently implemented on the object-relational or object-oriented platform. The relational database platform has enough live intellectual capital to remain robust for decade spanning life cycles. The preservation of relational databases is a fundamental interest of all modern governments and businesses, since it serves as the heart pumping the information flows necessary to maintain the prosperity of organized human life on this planet.

11.1 Purpose of Quality Assurance

Quality Assurance is an insurance investment to protect a development effort from unexpected problems and a much higher cost to fix those problems at later stages in development, implementation, assembly and some cases mass deployment.

Safe quality assurance procedures ensure that all predefined checks against previously occurring and thus well know errors are done, and that this is documented in a way that ensures that no changes are done to the checked design object after the checking is complete.

The work flow from design to delivery of objects are roughly:

- 1) Create the object, or copy a previous object and modify the copy
- 2) Check that the object meets the quality requirements necessary for it to serve its function
- 3) Mark the object with quality assurance information
- 4) Release the object to the supply chain

The quality assurance information, if handled correctly, is a huge time and cost saver when the object travels through the supply chain. If such information is not available when the object is received, the customer of the object has to spend time and money on inspection and relevant testing

equipment to do their own quality assurance checks, if they want to be sure that the object will serve its function in the customer's own product.

In longer value-adding supply chains, such avoidable duplicated testing costs may make the total cost of the final assembled product unaffordable for the end user.

11.2 Purpose of Version Management

During a design and development process, objects evolve when new functionality is added and errors or problems are corrected. This means that their properties change. In a supply chain, the customer of an object must be able to easily tell different versions of the same design object apart from each other, to ensure that newer versions of the object still serve the same function they are used for in the customer's design. Newer versions of objects may have changed in a way that requires the customer to adapt the design of the product in which the object is used, to make the customer's product deliver its intended functionality.

The identification of different versions of an object is done by attaching a version identifier to the object.

In complex supply chain businesses the concept of "form-fit-and-function" is fundamental. To illustrate this concept, for different versions of **physical objects** to be interchangeable within a particular assembled product, these objects must have a compatible **form** so they have physical space to be placed in the same geometrical position within the assembly. They must **fit** with the mechanical interfaces provided for securing the object in the assembly, e.g. screw holes, snap in tabs or similar. The interfaces towards the environment, for example the nozzles of a water tap or valve, must match the physical shape of the connecting nozzles transporting the water or medium, whose flow rate is controlled by the valve. The different versions of interchangeable objects must also perform the same **function**. E.g. a Valve must shut off fluid when its handle is turned counter clockwise to a particular geometrical position, and similar.

For **immaterial information objects** such as written text for information transfer from human to human, human to machine or machine to machine, similar but abstract form-fit-function properties apply to the object. Properties that make it serve its function within an assembly without enforcing costly changes of the assembly to accommodate it. For example a piece of text, such as a paragraph of legislation, a requirement on a product design, or a program that can be executed by a human or a machine must have a **form**. There are many aspects of form for text. The Language used for expressing its information is one of them. In immaterial products that serve a function, the rationality aspect frequently determines the form. Few textual information objects mix different languages, unless its function is translation.

Fit means that the interfaces of the information object must attach correctly to the environment within the assembled product. In text, the syntax, e.g. the rules of how the textual information shall be expressed, must fit into the assembled product. The **function** of the information object must be compatible in a way that does not require the surrounding assembly to be changed, i.e. the environment's methods to make use of the function. This means that the interface used to make use of or call the function within the object must fit with the interface used within the surrounding assembly. A call or use of the function must deliver the same results, however the performance of a later implemented version of a function may differ with magnitudes.

In software industry the concept of Application Programming Interface (API), enabled software components and libraries delivering complex functionality to be reused in an added value chain. A supplier of a software component could sell it to a large market of customers. By that cost-efficient

division of labour an added value was created, that enabled the components to be sold for a fraction of its development cost, given that the number of sold items delivered more income than the cost to create, distribute and sell the software component.

For a **version identifier** to efficiently inform the customer, which may be a human or a machine, if a newer version of a software component can be reused without adapting the surrounding assembly, the version identifier must express backwards compatibility with regards to form-fit-and-function.

11.3 Semantic Versioning 2.0.0

Semantic Versioning gathers the distributed software community experience based practice into a standard syntax that enables the form-fit-function criteria to be expressed in a version identifier for a software component [SemVer 2.0.0].

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Here you are kindly redirected to the original source document of SemVer, which you can find at: <http://semver.org/spec/v2.0.0.html>.

Reading the original text and storing a copy on your computer for reference is a good investment of your time.

A note on SemVer rule : <http://semver.org/spec/v2.0.0.html#spec-item-1>

1. Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it should be precise and comprehensive.

A DomainModel is a specification on a higher abstraction level than an API. It is a declaration and definition of the names to use for generating a source code library that implements an API that follows certain easy to remember naming conventions for different standard information manipulation functions. Thus knowing the names of classes, attributes and relationships in the domain model, AND the naming conventions for the standard object manipulation functions delivered by a source code generator, defines the API.

From that you can infer rules like:

New version of a Class that has been given a new attribute -> increment the minor version identifier.

New version of a Class that has deleted an attribute -> increment the major version identifier, since the generated API to manipulate that now missing attribute is no longer available, and applications using it will not work.

Chapter 12 Quality Assurance Functionality

The understanding of what really delivers Quality and how the Added-Value Prosperity is founded on Quality, is pure self-preservation knowledge, and the foundations for morale
[The Poet]

The purpose of Quality Assurance was described in section 11.1.

Most creative developers find manual checking of their own or other peoples work booring and time consuming. There are however some exceptions to that rule.

- a) If the developer can learn something new while inspecting the design or code.
- b) If the developer sees the inspection work as a teacher's mission to bring up the level of skills of appreciated students.
- c) Shared group or community professional pride in the software, and a desire to keep it beautiful in terms of clarity and readability, efficiency, robustness and freedom from faults.
- d) Some other reasons that are beyond the scope of this manual

In larger software businesses, the checking effort is frequently delegated to a Test Department, that can hire personel with the appropriate character structure to pay explicit attention to details and get high on hunting and finding faults.

Anyhow, budget restrictions makes such solutions impossible for smaller software companies, thus the checking effort has to be automated as far as possible, and incrementally maintained as new types of errors are discovered.

12.1 Checking and Automated Checks

Checking is done by following a check list. The check list is the accumulated knowledge of errors that have occurred earlier, which can be detected by inspection of the design while focusing on particular known and detectable fault conditions.

Checks can be seens as rules attached to a class of objects, which determine whether the values of the properties of an instance of a class passes the check or fails to pass the check.

In a restricted information domain, that can be expressed in a DomainModel²³, such checks can be automatically derived from the DomainModel and automated.checks generated for a particular application or product model database.

²³ Since the self describing DomainModel of OOCASE is implemented in the same language that OOCASE allows other applications to be described with and uses to generate compileable source code with, these other applications can reuse the same checking functionality that OOCASE uses to quality assure its own design.

In OOCASE and applications that build in its common frameworks, checks are declared in Model Check Definition files with the file name extension .mcd. These mcd files are initially generated from the DomainModel of the application, and constitute standard deriveable checks.

Here is an example of an extract from a check definition file illustrating the checks for the name attribute of a Class in OOCASE:

```
2NI : NameIntegrity - ensure unique and valid names in every namespace environent
-- OOCASE model check configuration file.

-- checkXX = E    : Reports an error if the check is violated.
-- checkXX = W    : Reports a warning if the check is violated.
-- checkXX = M    : Reports a message if the check is violated.
-- checkXX = I    : Ignores the check.
-- constXXYY = <value> : Constant parameter YY used by check XX.

...
-- **** Class Check Definitions ****
[Class]
...
checkNameHasFirstLetterConvention = W
constNameCapitalFirstLetter = true
checkNameHasWordSeparationConvention = W
constNameWordSeparationConvention = Capital
checkNameIsSpecified = E
checkNameMatchesRegularExpression = W
constNameMatchesRegularExpression = [A-Za-z][A-Za-z0-9]*
constNameMatchesRegularExpressionMotivation = Restriction to match valid syntax of
variable names of most programming languages targetable for source code generation.
checkNameSize = W
constNameSizeMin = 1
constNameSizeMax = 27
checkNamesUniqueWithinNamespace = E
...
```

MCD files follow the syntax of Configuration files described in Chapter 8.

The first line is the header that the user can see when selecting which model check definition to apply. Then some comments preceeded by a double minus (--) that roughly explains the conventions of the configuration file for a first time reader or user.

Each class in the domainmodel of the application has its own section, starting with a braced class name. e.g. [Class] above, followed by the set of automated checks that can be configured to generate an error, a warning, a message or be ignored in that particular model check definition. The names of the checks are chosen to be self explanatory.

12.2 Check Level Structure and Checkpoints

To keep the number of variants of model check definitions down, these are organized into a check level structure that reflects the needs of development phases, sometimes separated by checkpoints in a typical project.

A checkpoint specifies the level of quality a specification much reach before work on the next development phase is allowed to start. The rationality behind checkpoints is to avoid losing work effort on implementing parts of a specification that risks becoming useless or obsolete due to later design changes on a higher level.

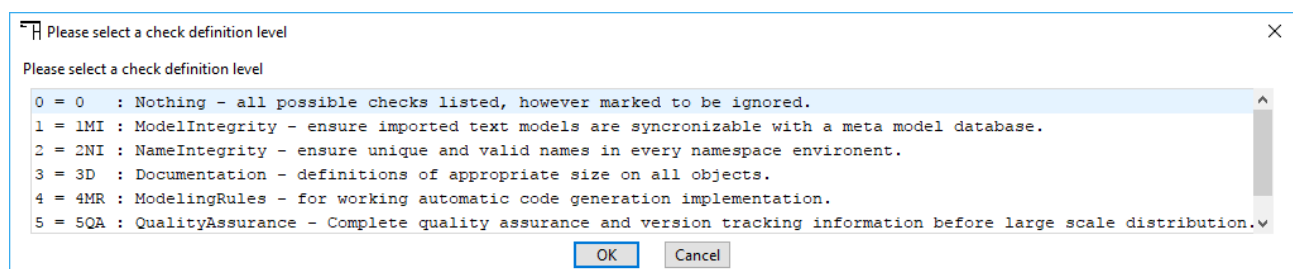
The requirements for passing a checkpoint are there to enforce that higher level more cost impacting designs are complete and fully agreed upon by the development team.

A full understanding of the whole of a design, frequently delivers insights how it can be simplified by refactoring or reuse of frequently occurring structural patterns through the mechanisms of inheritance.

A simpler design that delivers the same functionality that a more complex one does, is easier to implement. In addition, but severely more impacting over a software life cycle, a simpler design is easier to teach and maintain over the software's life time.

Changes in terminology have a severe impact. Frequently the right choice of names for design objects is not apparent until the whole design can be seen in its entirety.

When using, creating or adapting a model check definition, the following baseline check levels are used.



A higher check level number means higher quality. An unchecked design, for example imported from an external system, may require significant work before it lives up to the standards of a cost efficient life cycle of a decade surviving software.

If all possible checks are applied at once, the list of reported errors may become excessively large. A strait-forward approach to just fix the errors in the sequence they are reported in the list leads to a and very boring, inefficient and partly counterproductive use of the designer's time.

The reason is that such a list intermixes small and big issues. A fix of a small issue in the beginning of the list, may be obsoleted when a bigger issue is fixed later in the list.

The format chosen for human identification of check levels covers the user categories ranging from expert to rare occasion users and novices. The format is:

<check level> = <check level mnemonic> : <Check level name> - <Check level concept recall description>²⁴

The following sections go through the baselines of the chosen check levels, which can be adapted to particular project and implementation needs. This is typically done by creating a .mcd file using the baseline, appending a number or letter to the <check level mnemonic>, and the rest of the human identification information. Then adapting which predefined checks to deliver warnings and errors, and adapting the constants that set the parameters for the checks.

²⁴ This design was dictated by the all-mighty GOD that evolved the human visual system and by that the speed possible to achieve with the biological hardware pipeline available in a human being. If you are interested in that and want to understand the full rationale behind this design choice, you need 2 weeks of uninterrupted 8 hour per day studies of the books in B.

12.2.1 0 = 0 : Nothing - all possible checks listed, however marked to be ignored.

This model check definition is a useful baseline if you just want to run a few specific checks. Starting with a template where all checks are ignored just requires editing lines on the checks that are needed. The checklist for a OOCASE DomainModel contains more than 2000 checks. The checklist for DocumentDictionary more than 1000 checks.

12.2.2 1 = 1MI : ModelIntegrity - ensure imported text models are synchronizable with a meta model database.

Many widely established programming languages including relational database languages require their data definition language to specify the length of text strings and value ranges of integer and floating point numbers. This enables their language compilers to optimise data storage allocation in computer memory, and select the fastest processor machine instructions that will do the actual computations described in the high-level programming language.

Many languages and particularly relational database languages have constructs to prevent storage of complex structured data that violate constraints. The purpose is to prevent entering data that can't possibly contain any valid or useful information.

Such language constructs serve as protection against faulty software and faulty data.

Before a product model can be stored into a particular database implementation, a passed "1MI : ModelIntegrity" check ensures that the data contained in the model actually fits into the data structures provided by the particular database implementation and does not violate any constraints.

In a supply chain, it takes time to change present infrastructure of database and program implementations that serve the information shipments in real time. If a new product model, whatever way it was created, passes this check, the supplier can be sure that it will successfully be delivered through the present installed software infrastructure. If the infrastructure has sufficient quality, which is a distributed responsibility of all actors serving and maintaining it, where many never met or heard of each other, but unite on the shared values exposed by the added value delivered by the design²⁵.

12.2.3 2 = 2NI : NameIntegrity - ensure unique and valid names in every namespace environment.

A Name of an object or concept is the primary mechanism used in language to transfer meaning between a sender and a receiver. In a software life cycle that spans decades the choice of names is essential. The meaning of a chosen Name for an object must remain permanent in an environment where people and interfacing softwares come and go while they evolve in their careers and lifecycles.

A well-known problem with Names is that they become overloaded. That is a Name in one particular context does not mean or represent the same thing in another context.

To choose a GOOD name for an object, the designer must have an understanding of the full range of interacting senders and receivers in the environment where the created software is intended to serve a function an purpose.

The choice of names requires interaction and negotiation with experts familiar with all social and technological communities that somehow will interact with the software.

In order to have such an interaction and negotiation, all affected parties must be able to study the design and provide their constructive comments on how to improve it.

²⁵ Something that requires an educated eye to see.

The 2NI check level ensures that there are no machine detectable errors in a proposed design, before its evaluation starts generating the costs of human inspection and feedback.

12.2.4 3 = 3D : Documentation - definitions of appropriate size on all objects.

To preserve the understanding of what a design object means in a volatile unpredictable environment, the meaning needs to be defined in a way that serves the purpose of the software in its predictable environment, during its life cycle.

Given the overloading problem of names and the unpredictability of information senders and receivers interacting with it during its life cycle, an agreement on a definition that is judged to be understandable by all expected interacting current and future parties is essential.

A definition is written and discussed during the creation of the design, but will be read magnitudes of more times by a huge variety of software users, given that the design's inherent quality delivers that kind of expansion.

The 3D check level ensures that machine detectable errors are not present in a documented design proposal before submitted for evaluation and feedback.

12.2.5 4 = 4MR : ModelingRules - for working automatic code generation implementation.

This check level has a severe impact on the implementation cost of automated source code generators.

If the designer of a source code generator can assume that the declarative model specification is fault free, the code to capture a huge number of possible fault conditions can be skipped and development efforts be spent on delivering the best possible performance of the generated source code with regards to all predictable situations where the generated source code needs to communicate with humans and machines.

12.2.6 5 = 5QA : QualityAssurance - Complete quality assurance and version tracking information before large scale distribution.

This check level ensures that valid quality assurance information and version tracking information according to SemVer has been applied to the entire model, to the level possible to detect by a machine.

12.3 Recording a passed check

In an application supporting Quality Assurance, all objects that inherit from DBObject, have the attributes checkedBy and dtChecked. These are used to record who the object was checked by and the date-and-time stamp when the check was done. The purpose is to identify who was responsible for the check. Stamping an object with ones identity gives a justified sense of responsibility. If the check was incorrectly done, this will eventually show itself later in the development process when something goes wrong due to the inadequate check and the responsible person identified and held accountable. Being accountable is a way to ensure that the checking is done properly.

Normally these attributes are write protected, and can only be written while using a special checking functionality to maintain the integrity of the information.

A passed manual check is recorded by storing the login identifier of the user who did the manual check in checkedBy. Login identifiers for humans begin with a alphabetic letter.

Automated checks that are documented by a model check definition file can also be held responsible and accountable. Thus if an object passes an automated check, the check level

mnemonic which serves as identifier of the model check definition is stamped onto the object in `checkedBy`.

This delivers precise information to the user of a checked object what level of quality the object has. Failures later in the process can be traced to this model check definition, and the model check definition be updated to ensure the failure condition is detected by the automated check in the future.

12.4 Approving an object

Approval of a design means that a person or design team takes the responsibility for it. This is a matter of efficiency in targeting feedback ranging from error reports, suggestions for improvements, praise, credibility and earned trust.

Usually the manager of the design team is the person that approves a design before it is released. The manager is the person that has the best overall understanding and connectivity into the design process, its organization and infrastructure. Targeting feedback to the manager, is usually the most efficient way to channel it to the correct place within the development organization. An organization that in most distributed supply chains is a complete black box for a customer.

The recording of an approval is done with the attributes `approvedBy` and `dtApproved`. These attributes are also write protected to ensure integrity, and can only be written using special approval functionality.

In most cases the approval functionality provides access to checking information, that on a high level delivers decision support whether to approve the design or not. The function to approve a design is usually executed from the top hierarchical level of the design to be released.

A top manager can delegate the responsibility to approve self containing design components to middle managers in the organization. This is rational when the design is large, and the top manager is responsible for how it all integrates. This delegation however requires that the middle managers have the support an infrastructure to handle incoming feedback efficiently. If this is not the case, the top-manager takes the approval of a design from a trusted middle manager as the quality assurance stamp that it is, and overwrites those stamps with the top-managers own stamp. This way, the "fuss" from the environment can be kept outside the internal organization, and met by people who are skilled in dealing with it on a more frequent basis.

Chapter 13 Version and Release Management

*How can you be sure that a black box will do what it is supposed to do?
That is a matter of earned trust delivered by cleverly managed responsibility.
[The Poet]*

13.1 Editions, Versions and Releases

In creativity supporting interactive design tools, unnecessary bureaucratic procedures and restrictions are productivity killers, since they interrupt the designers flow and line of thought with unnecessary details that can be fixed in a later cleanup sweep.

A creativity supporting design tool is as much an aid for thinking and exploring as a tool for documenting the final design.

Thus the approach taken in OOCASE for version and release management is a "non-intrusive" automatic one that goes on in the background without burdening the attention of the designer.

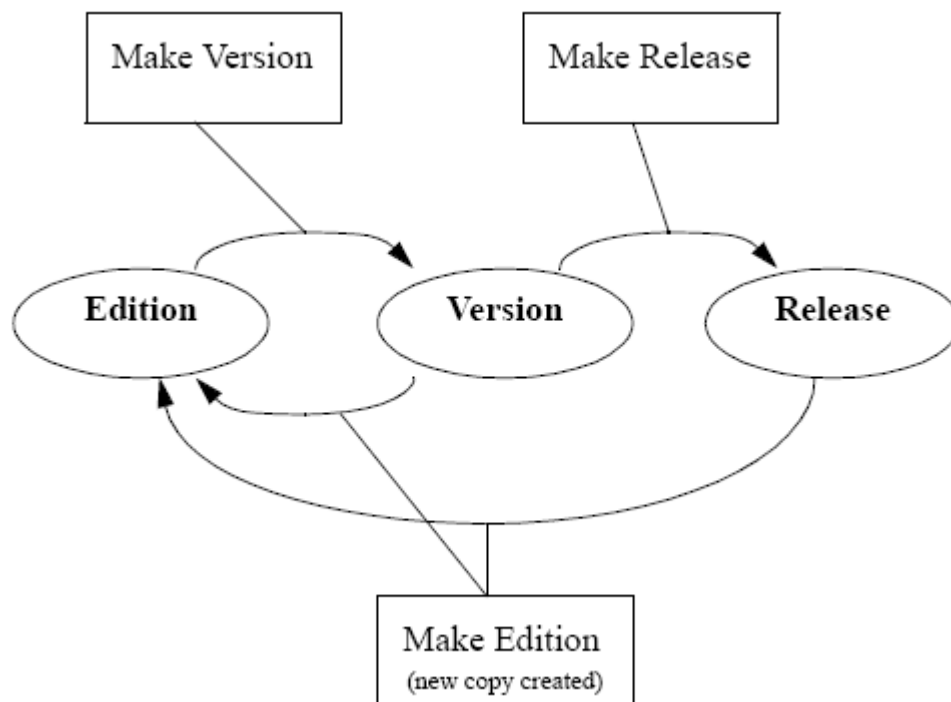


Figure 5. The quality assurance life cycle of objects in the meta model database

All design objects have a quality assurance state depicted with ovals above. The states used here are named Edition, Version and Release. The name of the state Edition comes from the word Edit.

Version means that the object has been quality assured and assigned a version identifier that follows

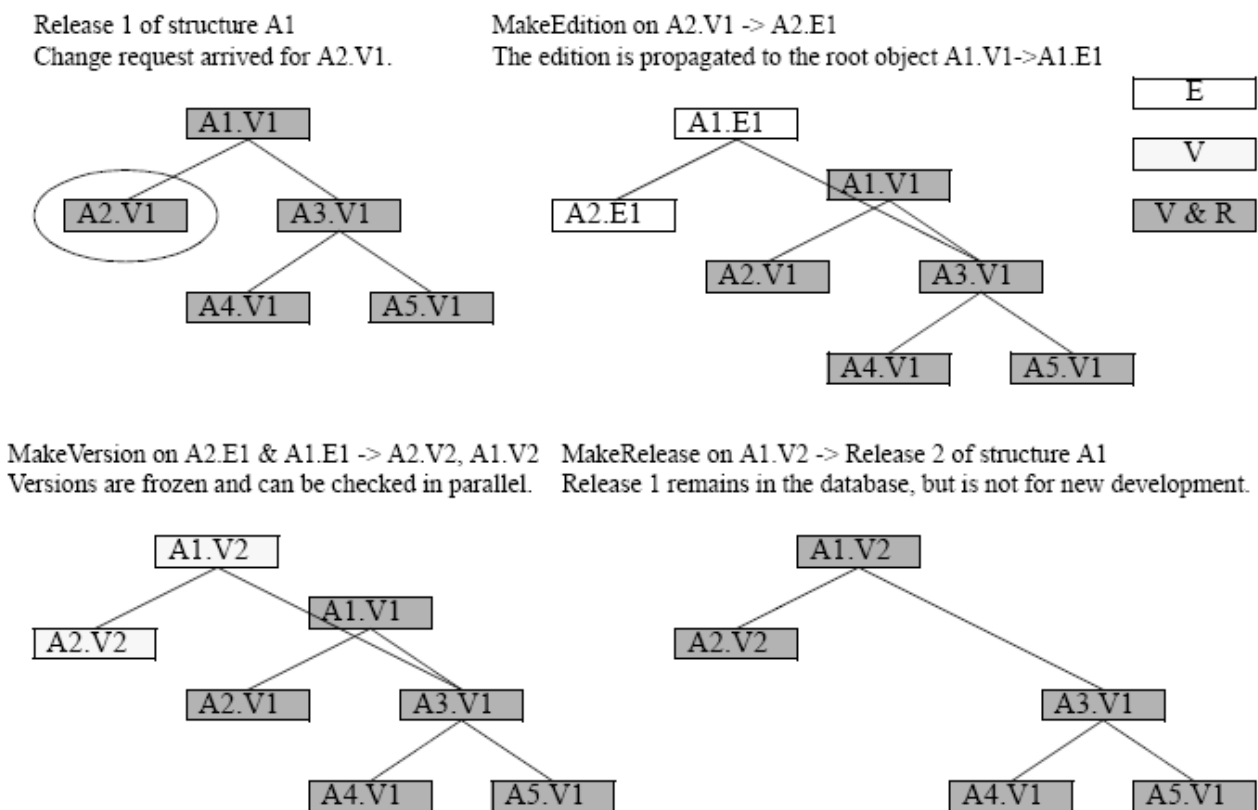
[SemVer 2.0.0]. Release means that the object has been quality assured to a level where it serves a purpose to release it to a supply chain.

Most productive design work start with a copy of a model that is similar to the one the designer wants to create.

In OOCASE and DocumentDictionary a newly created or modified object that has not been quality assured yet automatically receives the state Edition or transits into the Edition state via the automatic process MakeEdition. With regards to the whole model that the object participates in as a part, this edited object is a new object version, distinct from the object version it originates from, thus needs to be uniquely identified. The version identifier automatically assigned must be unique and meaningful to a human being in order to be easy to work with in human thought processes. Time is unique and meaningful, thus the time when the object transits from the Version or Release state into the Edition state is used to uniquely identify the Edition.

When quality assurance²⁶ has been completed for an object a new version can be made. This is the Make Version process²⁷.

Figure 6 illustrates the quality assurance life cycle of a complex object A1. E denotes that the quality assurance state is Edition. V that it is Version, and V & R that the version has entered the Released state.



²⁶ This means checking on a level that serves a purpose for the development phase that the design currently is in. For complex designs that need coordination of feedback amongs many actors, quality assurance also includes approval by the manager or organizational unit that is responsible for the version of the design.

²⁷ The Make Version process needs to be tailored depending on design phase and evolved best practice within the development organisation around the designs that can be expressed by instances of the domain model. This may involve procedure descriptions and rules of thumb how to allocate version identifiers, depending on the communication and information distribution needs of the development process.

Figure 6. Example of quality assurance process for complex objects²⁸

In OOCASE, the act of editing an object will automatically propagate the MakeEdition upwards the hierarchical assembly structure up to the root object in the structure.

When the designer is confident with the new design, automated checking can be applied where all objects in the chosen assembly structure that fail the checks are listed and can be corrected interactively at once.

Assigning versions only makes sense when the model is shared in a distributed team, or when branching of the development effort is necessary. E.g. keeping a quality secured "fallback" version if the latest development increment fails to meet its deadline or turns out to be a "dead-end".

Assigning practically useful version identifiers to large object structures is a bulk job that needs to be guided by rationality, in order to serve its purpose. What is rational depends on how the design and its components are used individually upwards the supply chain.

The purpose of releasing a model is distribution to the supply chain. When relational databases are used, several releases must be able to reside in the same database. This would lead to object identifier collisions if the same object identifiers are used as in the previous release. This is handled by a mechanism that reassigns new unique object identifiers to all objects in the model while ensuring that referential integrity is maintained in objects that use object identifiers as pointers to other objects.

13.2 Version History

In a software supply chain network where the design output of one supplier node in that network is used downstream in a distributed organization working in parallel, there is a need to track the delta of changes between software releases.

When a new release arrives at a node downstreams, that node needs to evaluate how design changes, new functionality and bug fixes impact their own added value software product.

Decisions must be made on what makes sense to support for their own customers up the value chain and a benefit/cost analysis with regards to the customer value of including those changes. After that a development project can be layed out in priority order with nessecary architectural basis first, than features and bug fixes in benefit/cost order.

In OOCASE and DocumentDictionary the version tracking information is maintained by the automatic MakeEdition process that is triggered as soon as an object in the state of Verion or Release is edited. Before the edit is applied, this process stores the unique object identifier (highid, lowid) and the version and release attributes in releaseBasedOn attributes named releaseBasedOnHighId, releaseBasedOnLowId, releaseBasedOnVersion and releaseBasedOnRelease and replaces the version attribute with it's new edition identifier and sets the release attribute to null.

Thus a reference to the original for the edited object is stored in the object itself. This information can be used to trace an object back to its original, regardless which distributed copy of the original is used, independent of any software implementation.

²⁸ To roughlyly contretize this a bit. In the context of an OOCASE domain model, A1 can be a domain model object. A2 and A3 classes and A4 and A5 attributes. In the context of a DocumentDictionary domain model, A1 can be a document dictionary object, A2 and A3 document records and A4 and A5 content records.

The object is truly immaterial, and can exist in parallel wherever it is stored. It can be uniquely identified and version traced without a centralized repository.

A customer can edit a model from the supply chain for internal business purposes without any need to create new releases of it, unless the customers they in turn supply require access to an accurate model that describes the product they purchase.

In this case the customer of the original release needs to quality assure their own modified version of the object, assign a suitable SemVer version identifier that serves its purpose for their own customers in turn, and make a release.

To preserve the properties of a truly immaterial version traceable object, each customer in the supply chain must generate their own globally unique identifiers for their new release. This is ensured by a centralized ledger of serial number series that each customer in the total supply chain network receive their unique highId identifier from.

Thus if supplier A denotes their model V1.2.3, customer B who buys it and uses it as a component to efficiently build added value for their own customers, can have their own SemVer compatible version assignments, and since the unique highId of supplier A, and unique highId of supplier B that is used when they release their added value versions of a model, identifies these version identifiers as different since they origin from different suppliers.

The economy of the collective human mind is to be considered when assigning version identifiers. HTML 4.0 has a distinct meaning to a huge user base. SemVer allows the supply chain to append their own version identifiers on that from their supplier as long as they follow the rules of SemVer. It is frequently worth the while to collaborate with the supplier if the value-adding adaptation of their product is non-backwards compatible.

This section gave a view on supply chain perspective with regards to version history. This view is orthogonal to the topic in the next section, which deals with the part-of structure of a model.

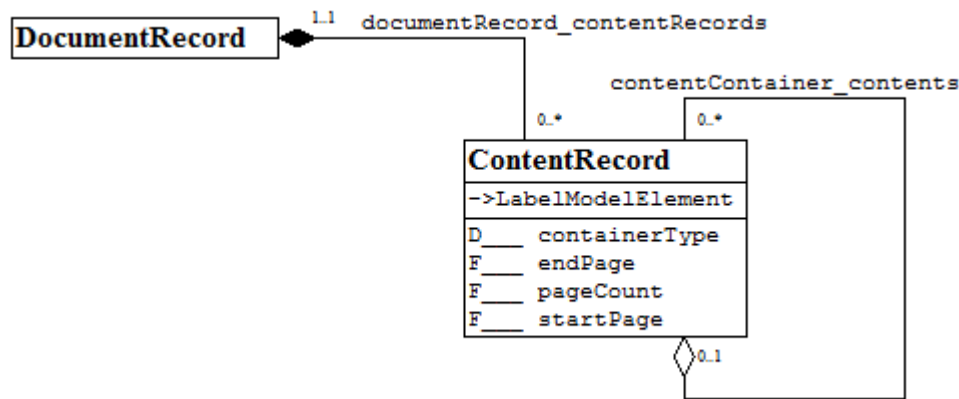
13.3 The VersionOwnerPath

The version owner path is used to determine the hierarchical part-of structure of objects in a model assembly when version identifiers are bulk assigned top-down in the hierarchy of an edited structure. All objects must know how to determine who they immediately belong to in the assembly structure, and by that be able to derive their unique path to the root object in the model.

One such structure within a DomainModel is the Module hierarchy, where Classes and Relationships distinctly belong to a single Module, and a Module may be be a part of another module etc. This is similar to the Package hierarchy in many programming languages and in UML.

13.3.1 Design Rules for Recursive Hierarchical Relationships

Objects in a Model may be organized into recursive hierarchies. For example a ContentRecord that represents a chapter in the table of contents in a document may contain sections which in turn may contain subsections etc, that are all instances of the same class ContentRecord.



In relational databases and fixed file format storages, large such recursive hierarchies cause a tremendous performance loss while loaded and stored if the nesting has to be traversed and built while loading²⁹.

Thus the following design rules apply.

When a recursive hierarchy is needed for a Class, all objects of that class must have ONE true part-of owner relationship denoted by a filled diamond on the owner side in UML.

This way the whole collection of owned parts can be loaded or stored in one single table or record scan.

The recursive structures are "simulated" by one or several aggregated 1-N relationships denoted by an unfilled diamond in UML. These nested structures are reconstructed with direct pointers in primary memory after all objects have been loaded from the external data source and have been indexed by their object identifier and/or primary key.

13.3.2 Direct Aggregators

When an object is edited, the MakeEdition must be applied on the edited object all the way up to the root of the assembly structure the way the users perceives the assembly structure. For a true owner relationship (black diamond at the owner side), it is strait forward that an object has been edited and become a new version when one of the components it owns has been edited.

For recursive hierarchical relationships, such as 'contentContainer_contents', if a subsection is edited, then the section becomes a new version, and the chapter that contains this section, not just the DocumentRecord that is the true owner of the subsection, since the user perceives the Chapter as containing the section and subsection.

²⁹ If you are an expert with at least 25-30 years in database industry, you know that this problem is elegantly solved by database implementations that provide sophisticated query optimizers or elegant use of hardware support for virtual memory in a way that makes this unnecessary. Such implemtentations are however beyond the purchase capacity and performance needs of the masses we need to mobilize to feed the computational centers with facts. The organizational and educational investment needed to put such advanced database implementations into profitable productive use are some career steps above the basic added value we can deliver here. Standing on our delivery, it is much easier to take on challenges upwards the added value chain. Since you yourself are probably forced by reality to become a manager by now, you must ask yourself what people you would like to hire. On what level do you want to start building the added value knowledge necessary to for an new appreciated co-worker to become productive in the use of your products? Someone who can see the difference between a good design and a brilliant one? You know our history, and we need eyes in our young generation that can see. The end-state calculation along the current trajectory is very important to disperse to influential people who can't see but have all potential to develop their sight. We can fix it. It's just a bug. An attitude that fixers need to have, in order to mobilize the best parts of the legacy we are made of.

Thus the following rules apply:

- 1) If an object is and has an aggregator on the white diamond side of a recursive relationship, then MakeEdition will trigger that aggregator to be a part of the new edition.
- 2) If an object has no aggregator on the white diamond side of a recursive relationship, it means that this is a root object in the recursive aggregation hierarchy that is owned by the true owner relationship only. Thus being the root in the aggregating recursive relationship it is this object that propagates the MakeEdition to the shared true owner.
- 3) To maintain the integrity of the version owner path, the implementation must ensure that all objects in a recursive aggregating relationship that are connected in a hierarchical assembly structure share the same true owner³⁰. *(For example if the user interface provides the functionality to move (by drag/drop) a ContentRecord representing a section from a table-of-contents structure from DocumentRecord A to a table-of-contents structure in DocumentRecord B, the true owner of this ContentRecord must also be changed to DocumentRecord B.)*

³⁰ For deeper more complex assembly hierarchies that may involve recursive version owner path loops where an object of class A may be the true owner of an object of class B, which in turn may aggregate other objects of class A, the shared direct or indirect owner that has no aggregators must be the same object. The in theory and practice test is to ensure a design that if an object that participates in recursive relationships is deleted, it should leave no dangling aggregated objects behind that are not deleted straight down forward through its true part-of recursive ownerships. Implementations of drag-drop moves of assembly structures where the object dragged is aggregated, need to ensure the shift of true part-of owner.

Chapter 14 Using a Relational Database for Model Sharing and Distribution

<Well The Poet is eager to tell the Poet's Story>

Author's comment: I'm completely baffled. You told you that?

Back to business again. All information need a data representation that efficiently serves a number of functions. Roughly, excluding some that we can not cover in this manual, these functions are the following:

- 1) Persistent storage of information over time.
- 2) Retrieval of information in a format that serves the purpose of the task that is querying the information storage.
- 3) Integrity protection of the stored information against known information corruption fault cases. These are typically coming from corrupt or malwritten programmes that enter data into the persistent storage that violates fundamental information integrity constraints.

The theory of Relational Databases dates back to pioneering work done by IBM in the late 1960's. After a number of landmark publications the scientifically educated workforce interested in relational database theory grew significantly, and with it came successive series of research prototypes that evolved into commercial database products.

Practical experience of using these products for high-volume transactions on large volumes of information stored into tables, feed back input to further theory development that lead to higher performance and better methods for protecting the integrity of the information. These were for example declarative specifications of constraints that the database engine itself must uphold with automated implementations, that ensured that faulty information could never be entered, and thus the problem was delegated to the source of the faulty information that received error messages, without any harm being done to the stored information.

There are many in the public domain available high-quality software implementations of Relational Databases.

Even if the interactive performance is higher of some other types of open-source software for sharing and distribution of information, the integrity protection aspect of such implementations is weaker. Their capabilities are better suited for read-only massdistribution of preprocessed high-quality models. Where updates to the information are channeled through a more rigorous quality assurance process than these softwares' internal functionality can provide by itself.

Regardless if the relational database is provided as an on-demand put-on-line service in the cloud or permanent up-and-running physical computer that idles when it has no workload, the service itself requires a number of things before it can be deployed.

14.1 Requirements for Providing a Relational Database Service

This chapter is not finished yet, but will be a condensed essence of a number of database courses given at the university and in industry.

Chapter 15 Reuse of Models with Copy and Paste

We fired the poet, since what he delivered on this chapter just told us he couldn't contribute constructively to our business plan. Now we are fair and respect talent when it does its homework, so "The Poet" is welcome back once he/she or whatever that thing is, delivers a high quality statement that speaks the essence of this chapter.

[The BOSS]

Authors comment: I'm with the BOSS. A text is a tool for thinking until it has been quality assured.

One of the largest productivity benefits with using a CASE tool, is the possibility to reuse quality assured, well understood, known to be good, models as baselines for new applications.

Plenty of core modules can also have functionality libraries implemented directly in target language source code that use and build on the automatically generated source code from those modules, and provide a reuse leverage when building new applications that share the functionality of the same module.

Reuse of well known modules also has significant benefits in the human infrastructure that creates, maintains and uses the software and the information whose structuring and task adapted user interfaces delivers reusable internal knowledge models within the individuals' S.

It is much more expensive to "upgrade" the human knowledge of a whole organization or customer fleet, than upgrading a complex software if that software is implemented with state of the art methods and technology ³¹.

Since the market success of the Apple Macintosh computer series in the 1980es, that made software industry almost universally adopt the "copy/paste user metaphor" from the physical office world, stands on taking a copy of a physical object and placing it in a clipboard. The paste action will then copy the contents of the clipboard onto the selected target object. The copy in the clipboard remains intact, and can be pasted on other target objects.

Copying a complex piece of structured information that may contain thousands of interlinked objects according to this user metaphor involves a number of non-obvious challenges.

To deal with those we need to understand the limitations of the biological hardware that implements our conscious understanding of what is going on in our by us controlled working environment through what the perception of understanding in human conscious working memory can deliver on the basis of the knowledge modules and abstractions that a human being operating user has learned.

³¹ The public awareness of this insight amongst professionals in the expanding software industry during the 1980'es who drew conclusions from the market success of operating systems working with, instead of against this built-in limitation of biological processing hardware implementing corporate and governmental organismic life, formulated this insight standing on the best-practice, state-of-the art language evolved for user interfaces at those times. One excellent example of an in practice for the software platforms made available by the science backed up gravitational core driving the industrial software expansion at those times is [IBM 1989]. This reference is selected because its author kindly sponsored some student branches in Computer Science during the 1980'es, with healthy win-win relationships, that fostered growth both in access to good knowledge and a supply of students for a future long-term sustainability oriented expanding industry that took well care of its staff. The book provides references to the scientific basis it needed its staff to stand on for continuing to serve the market with high quality software products. Since the present state of the art of the software interacts and shapes organizations and individuals, it is important to maintain the stability and integrity of the scientific core that delivers conditions and real active expansion power for healthy long-term win-win relationships, until the scope of applicability for that market is saturated and the income areas delivering enough profit margins move up the added value chain, standing on healthy established state of the art standards and its performance beneficial biological hardware compiled presence in the human work force.

15.1 Dominance Ranking of Classes in a DomainModel

In order to efficiently automatically generate source code for copy/paste functionality within the design space of a DomainModel, some deriveable information from the structure of the entire DomainModel needs to be pre-computed to keep the complexity of the source code generators down.

There are three relationship types in the design space of a DomainModel. Part-Of relationships, Aggregating relationships and Reference relationships, listed in their power-of-influence-order, to determine the rank of a class within a DomainModel, with a dominance ranking algorithm.³²

Algorithm outline:

1) Compute part-of rank - this is a matter of modeled physical assembly structure or inherent properties of the modeled external world when represented as information described in the language of the DomainModel.

2) Compute aggregate rank

Recursive top-loop - a self aggregating class.

Non aggregated parallel part-of peer.

Subordinate aggregate hierarchy peer.

3) Compute reference rank

Top-weight calculation (inside set)

Aggregated superordinated top-weight calculation (inside model)

Subordination weight/negative domination weight on "dominating" relationship paths

Statistics from actual instances of models decide

Dominates1to2 - a weight set by the designer or design team of the DomainModel.

DomainModel enclosed dominance rank of a class is the position in the sorted collection of all its classes ordered by partOfRank, aggregateRank and referenceRank.

To transfer a subset of objects within an isolated model container to an isolated clipboard container, while preserving all internal relationships within this subset, requires determining the boundary of this subset with regards to the surrounding model. The multi-dimensional dominance ranking enables this boundary to be visible by precompiled algorithms.

15.2 The Copy/Paste Metaphore and it's complication in the real world

The *object-action process sequence* builds on human language foundations of nouns and verbs. First you select the object, or objects with a multiple selection, and then apply the action. In this chapter's frame of topic the actions are Copy and Paste.

³² Dominance ranking of classes within a DomainModel is a well-known secret amongst professionals working with large scale optimization problems ranging from autorouting of printed circuit boards to optimal shop floor planning for efficient massproduction of a particular complex assembled product. That level of knowledge is off-course far beyond the ambitions of this manual's important foundational step of education of appreciated software engineers towards a successful sustainable value-adding career in the industry that delivers the basic foundations for our shared wealth. Anyhow the computational speed of silicon processors, compared to what is possible to achieve by manipulating the physical world that is the real thing that is initially accessible to human beings before developing higher level abstract thinking that can be reused in writing programs that control what goes on in nanosecond real time in a silicon processor, tells us that this is something we need to use cleverly to deliver what we need to create the physical items we need to change our world towards the business plan. Dominance ranking in general depends on the knowledge domain it is applied to. So DomainModels are a good tool, to with simple examples explain the more general principles.

In the context of information structures declaratively expressed with a DomainModel, there are several different types of Copy actions. Below for simplicity of the explanation we assume that the user has selected a single object in a model. Then the multiple selection options that are not a strait forward iteration of the single selection action are explained in more detail.

15.2.1 Copy Object Only

This copy action copies the object in the selection only, regardless what type of relationships the selected object is connected to other objects with. This copy action is frequently referred to as a shallowCopy.

This action is a very limited in its power to manipulate complex information structures, thus it is not made available through the user interface since it does not correspond to what most users of a copy action would expect this action would do.

15.2.2 Copy Part-Of Structure - Copy

This copy action is the default that is available for the user under the familiar label Copy. It behaves as if the root object of a part-of structure represents its whole, which can be verified by selecting a complex object and issue the command Copy on it, and then use Edit->Show Clipboard, and inspect the parts of the object in the clipboard.

15.2.3 Copy Part-Of- and Aggregated Structures - Copy Aggregate

In case the DomainModel contains aggregated relationships that enable the presence of recursive structures in a DomainModel, the Copy Aggregate action delivers something that is more close to what the user intends with a copy action. A typical in all domains reoccurring example is an assembly, that may contain other assemblies.

15.2.4 Copy Part-Of, Aggregated, and Upwards Dominant References - Copy Dominant

An upwards dominant object, is an object that is owned by a shared owner higher up in the part-of hierarchy of a selected object, whose properties are shared amongs many objects on equal hierarchical level in a part-of hierarchy below the selected object. Examples are shared membership in various kinds of groups or categories, that own shared attributes and resources, or reference links to particular objects outside the part-of and aggregate substructure below the selected object.

Dominance ranking is a means to automate the algorithmic implementation of:

- 1) Deterministical declarative identification of the span of reach for the contents of a copy from a selected object or multiple selection of objects.
- 2) Defining the categories of span of reach that the user may want to adjust for a particular copy action, to set the closure boarder of the copy.
- 3) Deterministically defining what perhaps not complete subset of a copy that can be pasted onto a particular selected target object.
- 4) The categories of deterministic target structure of the closure of the paste, that the user may want to adjust for a particular paste action.

15.3 Examples for Functionality Coverage and Performance Analysis

The following examples can be tested directly with the implementations provided in DocumentDictionary and OOCASE. They serve as evaluation pattern to be used for functionality coverage analysis and performance benchmark evaluation of a particular implementation using instances of well known Standard Models.

15.3.1 Copy Part-Of Examples

DocumentDictionary: Copy Paste of DocumentRecord owning ElectronicEditions and Notes

OOCASE: Copy Paste of a Class with Attributes

15.3.2 Copy Aggregated Examples

DocumentDictionary: Copy Paste of a ContentRecord aggregating a TableOfContents substructure through the contentContainer_contents relationship.

OOCASE: Copy Paste of Module hierarchy with Classes and Relationships.

15.3.3 Copy Dominant Examples

DocumentDictionary: Copy Paste of a Category of DocumentRecords with Category instance specific user interactive decision of recursive depth level of related DocumentRecords traceable through Cites. Use DBLP example.

OOCASE: Copy Paste of a GenericCategory of imported IEC61360 standard library of DataElementTypes, e.g. RosettaNet Technical Dictionary or eclass category.

15.4 Managing Traceability with Object Identifiers during Copy Paste

The implementation used is the one described in Chapter 13 Version and Release Management. When an object or closure derivable from a selection is copied, each copied object receives a new unique object identifier, and the referential integrity of the entire closure within the copy is ensured by a copy algorithm.

Since the copy within the clipboard has new object identifiers it is no longer the same object as the original, and thus the mechanism of MakeEdition is applied to it. Briefly recalled, this means that objects in the clipboard lose any previous stamped quality assurance information for this particular new edition, and are provided with ReleaseBasedOn information so a trace to the original object for the copy is stored in the copied object itself.

No application specific information is updated in the copy unless the application has some implementation of a post-copy method.

When a copy is pasted onto one or several selected target objects, their ReleaseBasedOn and Edition information remains the same as in the clipboard, and thus provides direct trace links to the originally copied objects.

Chapter 16 Information Quantity Measurement

The purpose of measuring information quantity was introduced in Section 1.1 Calculation of information quantity in a model.

16.1 Theory

The theory is well described in [Johansson 1996] in Chapter 12 Concepts and Notation from Infological Theory (page 97-107). The mathematical practice for how to calculate information quantity in a DomainModel is described in Chapter 14 Primitives for Domain Models (page 113 - 126).

16.2 Practise

OOCASE and the applications building on the same frameworks provide information quantity calculation in a model through File->Calculate->Information Quantity.

The user interface is similar to that for File->Quality Assurance->Check Model.

The information quantity can be measured in the unit bit, e-constellations (EC), or documentation e-constellations (DEC) which is an algorithmically computed approximation of the information quantity in natural language:

- 1 BITs (e.g. a 0 or 1) according to Shannon's theory based on the number of bits allocated for the selected TypeDef for storing an attribute value or relationship link as data in the model.
- 2 Elementary Constellations (EC) - Roughly described as the number of attribute values and relationship links in a model or part-of substructure in a model..
- 3 Documentation Elementary Constellations (DEC) - Various configurable methods for extracting the information quantity of text containing words and sentences written in natural language³³.

In reappearing bulk-work for a software engineer, information quantity is useful when checking roundtrips of information distribution mechanisms involving transformation between different data representations. DEC is practical for high-level brief checking that definitions have adequate size for a particular documentation purpose, and quickly identifying anomalies in the documentation volume.

Information quantity measurement can be configured in detail using configuration files.

³³ To measure information quantity more precisely requires a natural language parser that can classify words for their grammatic roles and map sentence expressions to information quantity with the aid of a quality assured grammar pattern matching database. This database has to be verified on some agreed volume of realistic application test models from a particular knowledge domain. The level of detail is a matter of the value of calibrating the scale for measuring the information volume expressed with a certain language.

Chapter 17 Profile Extensions of the MetaModel

*The blaze of speed and tuning, when geared to win the known,
is accepted by the looming, to be the final crow.
The unknown never ceases, delivering critique,
thus those who not it peases, will never be unique.
[The Poet]*

Authors comment: With a well conducted domain analysis and a sequence of with real production information evaluated prototypes, probably 95% of the average information handling needs can be covered for the bulk-volume of information processing needs within the domain. Agreeably the environment never stops changing, and taking care of the ripple on the surface of the bulk volume information representation needs over decade periods, is essential for maintaining the user support for a system.

Superior performance is quickly accepted for granted by a user community who are not into the exceptionally delicate details and history of how to achieve it in the layers of technology that it stands on. Human beings are tuned to react on things that annoys them and disrupt their daily conduct. Thus meeting the user's needs with a working, cost efficient, flexible but severely performance inefficient solution with regards to some infrequently occurring tasks, will allow the users to get that job done. And forget about that job, since users continue with the next job that most frequently is within the borders of applicability of the performance optimized system, and those accumulated smooth working experiences will cover for the lost credits of that other temporarily painful experience in the irrational human emotional accounting.

If there is no extension mechanism provided that can handle the ripple in information storage needs, the human emotional accounting will deliver it's verdict and provide a data migration project to some other system that until thoroughly performance tested will just remain being what it is, unless proper feedback loops are established with the new system's developers and maintainers.

With a leadership who know the value of benchmarking, resources for delivering decision support will tell when it is time to incorporate the experience accumulated in profiles into the application DomainModel, upgrade, migrate the information and push the ripple area to less costly levels.

The concept of profiles or DomainModel extensions has been discovered and rediscovered all over the planet in application environments that had an efficient enough core to expand beyond it's original scope of applicability.

17.1 Short Introduction to Profiles

A Profile is a restricted form of meta model that can be used to extend the DomainModel of an application such as OOCASE and DocumentDictionary. Profile specific Objects, Attributes and Relationships can be created and stored in a model that is supported by the application.

The purpose with Profiles is to speed up the development cycle, by allowing the user of an application to add functionality without having to ask the software supplier for help.

The benefit is the flexibility for the user to add the functionality needed in the environment that the user has to deal with.

The drawback is performance and non-standard heterogeneity that does not scale up very well in larger organizations and supply chains, that need to automate their communications with software that is available and affordable for the organizational units that carry their cost.

To the user however, in the situation the user is, having the flexibility of Profiles can have a severe impact on productivity and action capability to deliver added value to a wide range of customers. A well documented profile that has been used in production for some time and proven it's value can be the efficient design specification that the software supplier needs to be able to afford the investment of incorporating the functionality into the next version of their software product.

17.2 The DomainModel of Profiles

The DomainModel of Profiles has been discovered, implemented and rediscovered and reimplemented in parallel by independent non-collaborating actors all over the planet where people have access to programming languages and computers. Where the pre-requisite requirement for such a thing to happen is that these people live in an environment where curiosity and interest are promoted and allows them to develop an eye to see the underlying high-level patterns.

Every high-level pattern needs a concrete implementation to be expressed and communicated in the physical real world. The Elements used to formulate this expression are meta model objects that are grounded in the users understanding of their Names and Definitions.

In Figure 7, the layout dimension from top to bottom is going from the abstract to the concrete, that is from the class definition level to the instance level. The left-right dimension is an allocation of space for placing objects on a similar abstraction level such that it fits on the paper space³⁴.

A Model, be it a DataDictionary or a DomainModel may own one or many Profiles. A Profile owns ClassDefinitions and RelationshipDefinitions. These definitions can be modified dynamically when the application is running. If a user needs a new attribute on some application class, the user creates a Profile object in the model, and a ClassDefinition for the application class that needs to be extended with an attribute. The name of the application class to be extended is entered in the baseClass attribute of the ClassDefinition instance. Below this instance an AttributeDefinition instance is created, given a name, definition, perhaps defaultValue and type.

In the application, only objects of class ModelElement and its subclasses can be extended with Profile specific information. A ModelElement may own ProfileValues, which carry the profile specific attribute value in it's value field. The ProfileValue object also needs to know what attribute it's value is supposed to represent for the ModelElement that owns it. Thus the attributeName that the user has chosen to use for this value is stored in the field attributeName. The definition of that attributeName is recorded in the AttributeDefinition object in the Profile, and shared by all ProfileValues. The AttributeDefinition in the Profile is owned by the ClassDefinition that is identified by the className, which is also stored in the ProfileValue.

³⁴ A good geometric layout of a class diagram is arranged in a way that can be efficiently scanned by the physical implementation of the human eye and be remembered by the biological hardware we have in the higher abstraction processing layers in our biological hardware for recognition and processing visual sensory input during mental reasoning. It's a matter of efficient reuse of 2-dimensional structural layout to exploit the reusable mental picture that is helpful during thinking and utilize the vast storage capacity of the higher level processing layers of the human visual systems. See [Hubel 1988] for ideas of what shapes and structures our eyes are optimized for seeing and recalling with mental imagery. Compare the geometrical layout of Figure 7 with the geometrical layout of the DomainModel of OOCASE in Figure 3 and notice the reuse of semantic meaning with geometric location.

In a relational database, all ProfileValues are stored in the same table. Thus they need to know themselves what attribute their value is supposed to represent, and which class and by that inferable subclass hierarchy they might belong to in order to attach themselves to the correct ModelElement without excessive unnecessary search for object identifiers in tables that for sure will not contain the right modelElement.

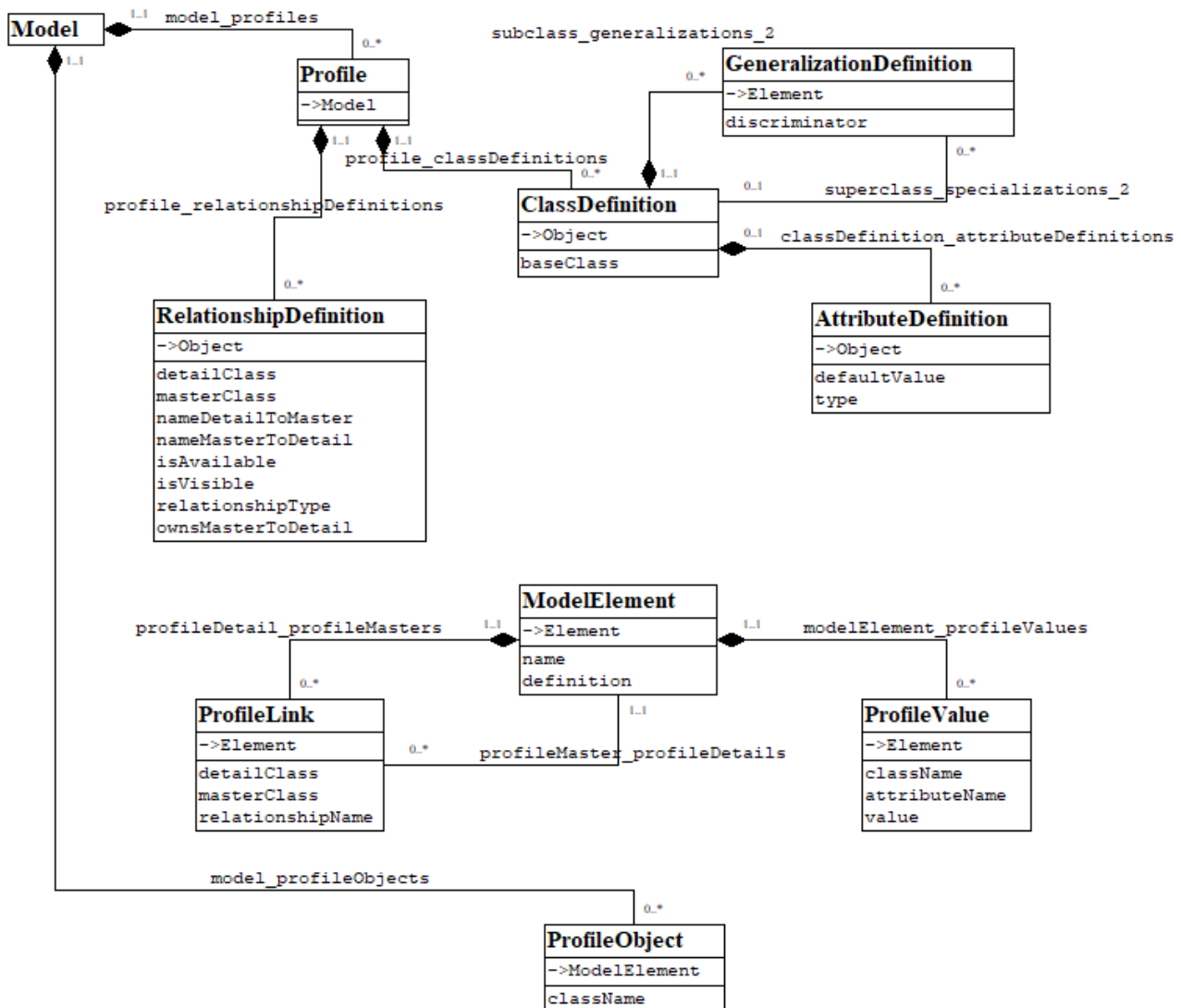


Figure 7. The DomainModel Module for Profiles

Relationships that the user creates between objects are represented by a ProfileLink. Since all ProfileLinks are stored in the same table, the links themselves need to know in what tables their particular master and detail may reside to attach themselves to the correct objects. The link also needs to know what it represents, and that is stored in the relationshipName, which is an index to the RelationshipDefinition by its name, where the user can define in a definition what links with this relationshipName actually mean.

In a multi-user database environment where data is stored and maintained over decades by different people, it is very important to provide definitions for ProfileElements. If this is not done, misunderstandings may lead to unnessecary additional costs in the maintenance of the physical or software objects that are documented by a model.

17.3 Comparison with UML Profiles

A quick comparison is to study Figure 12.12 on page 253 in [UML 2.5.1] and compare it with Figure 7 above. Perhaps study the adjacent text or browse the OOCASE MetaModel in OOCASE.

Chapter 18 Functionality

18.1 Egoless Business

The break-through paradigm shift of Egoless programming, was the separation of a working person from the subject matter he or she produced.

The code was no longer a primary source of personal pride, but an collectively owned delivery item that should serve its primary purpose flawlessly.

The pride of the team maintaining code, is 1) having cleanly formatted attractive looking easy to read source code according to an established coding standard, 2) no bugs and 3) high performance. In that priority order.

18.2 Building Efficient Interfaces Between Huge Refactorable Knowledge Domains - Efficient Knowledge Economy

This condensed essence section has not been written yet, but the source material is already published in [Johansson 1996]. The principle of extracting reusable designpatterns out of a whole piece of scientific work are also documented in other partly overlapping software engineering areas in [Gamma et.al. 1995].

Chapter 19 Summary and Conclusions

The design of OOCASE has its roots in the enormously enthusiastic software industry and academic computer science research advances in object-oriented programming, object-oriented databases, expert systems, programming environments, model driven software engineering and complete model driven application compilers that exploded in a period around 1988-1992. The core highly efficient design patterns evolving during that intense period have now been production battle tested for 30 years and plenty of practical experience has been gained from information life in an environment evolving under Mores Law.

The core technical information science has not changed, and the DataDictionary and DomainModel language has remained the same except for the adaptation to IEC 61360 in 1997 where Property was renamed to DataElementType, and augmented with standard attributes to be able to import large industrial standard libraries into the DataDictionary.

The core theory and the concepts have no competitor as simple and generic as OOCASE with the same platform neutral capabilities amongs widely available standard programming languages.

The addition in 2015 with improved functionality for Quality Assurance with Edition, Version and Release (QAEVR) management following SemVer 2.0, with full traceability through the releaseBasedOn(Highid, Lowid, Version, Release) attributes and renumbering of object identifiers when issuing a new release of a model, enable full distributed in parallell version tracking by independent organizations that know nothing about each other, while still being able to trace the version history in distributed independently maintained repositories in any SQL92 compatible relational database or simple TAB-separated table text files for the classes of the information model (DataDictionary and DomainModel).

This manual is a summary of what someone who really wants to make a long-term meaningful difference needs to know with regards to technical information that needs to be production live and maintained over decades while hardware and software platforms and programming languages change.

A. References

- [CORBA 1991] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Publications, <http://www.omg.org>
- [CORBA 2012] Object Management Group, "Common Object Request Broker Architecture (CORBA) Specification, Version 3.3", OMG Publications, <http://www.omg.org>
- [Fowler 2003] Martin Fowler, "UML Distilled, 3rd Edition: A Brief Guide to the Standard Object Modeling Language", <https://www.martinfowler.com/books/uml.html>
- [Gamma et.al. 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, ISBN ISBN 0-201-63361-2, , 1995, pp. 395
- [Goldfarb 1990] C.F. Goldfarb, "The SGML handbook", Oxford University Press, 1990, ISBN 0-19-853737-9
- [IBM 1989] IBM, "System Application Architecture - Common User Access - Advanced Interface Design Guide", International Business Machines Corp., 1989, Document Number: SY328-300-R00-1089
- [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.
- [Johansson 1996] O. Johansson, "Development Environments for Complex Product Models", 1996, ISBN 91-7871-855-4
- [SemVer 2.0.0] Tom Preston-Werner, "Semantic Versioning 2.0.0", Semantic Versioning, <http://semver.org/spec/v2.0.0.html>
- [Sundgren 1973] 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 1989] B. Sundgren, "Conceptual Modeling as an Instrument for Formal Specification of Statistical Information Systems", National Central Bureau of Statistics Sweden, 1989:18.
- [UML] OMG, "Unified Modeling Language (UML) Resource Page", <http://www.omg.org/uml>
- [UML 2.5.1] OMG, "OMG Unified Modeling Language (OMG UML) Version 2.5.1", Object Management Group, OMG Document Number: formal/2017-12-05, December 2017,

[XML 2008] w3c.org, "Extensible Markup Language (XML) 1.0 (Fifth Edition) - W3C Recommendation 26 November 2008", W3C, 2008, <http://www.w3.org/TR/xml/> (accessed 2017-12-19)

[The Poet] The Poet, "The virtue of successful poetry", (under review by an unscrupulous test department)

[The BOSS]

There is no adequate publication in the vast output that the BOSS already has made that will give any adequate picture of who the BOSS really is.

So if you ever meet the BOSS, you will recognize the BOSS's character traits, in that the BOSS is the nicest person you ever meet in your life. the BOSS seems to know you better than you know yourself. The BOSS cares about you and makes you comfortable to a level where you are able to really explain the reason why you are visiting the BOSS. The BOSS listens, and asks you questions. After a time in the pleasurable safe haven of being in the BOSS's nearness the BOSS signals to you that you need to listen to what the BOSS says, and when you get the BOSS's message, you are changed.

The change transforms you, you fly on the wings of an eagle and know exactly what to do. There is no doubt anymore and you know your BOSS is backing you if you follow the advice the BOSS delivered.

[The Poet] who knows exactly what The Poet's homework is to get the grace of being allowed to meet the BOSS again.

(now there are plenty of bosses who are encouraged by this and lock themselves up in their ivory towers, anyhow the BOSS is ultimately implemented by our own biology with it's own history, so a good boss knows you well and knows what to say to make you deliver on a meaningful business plan)

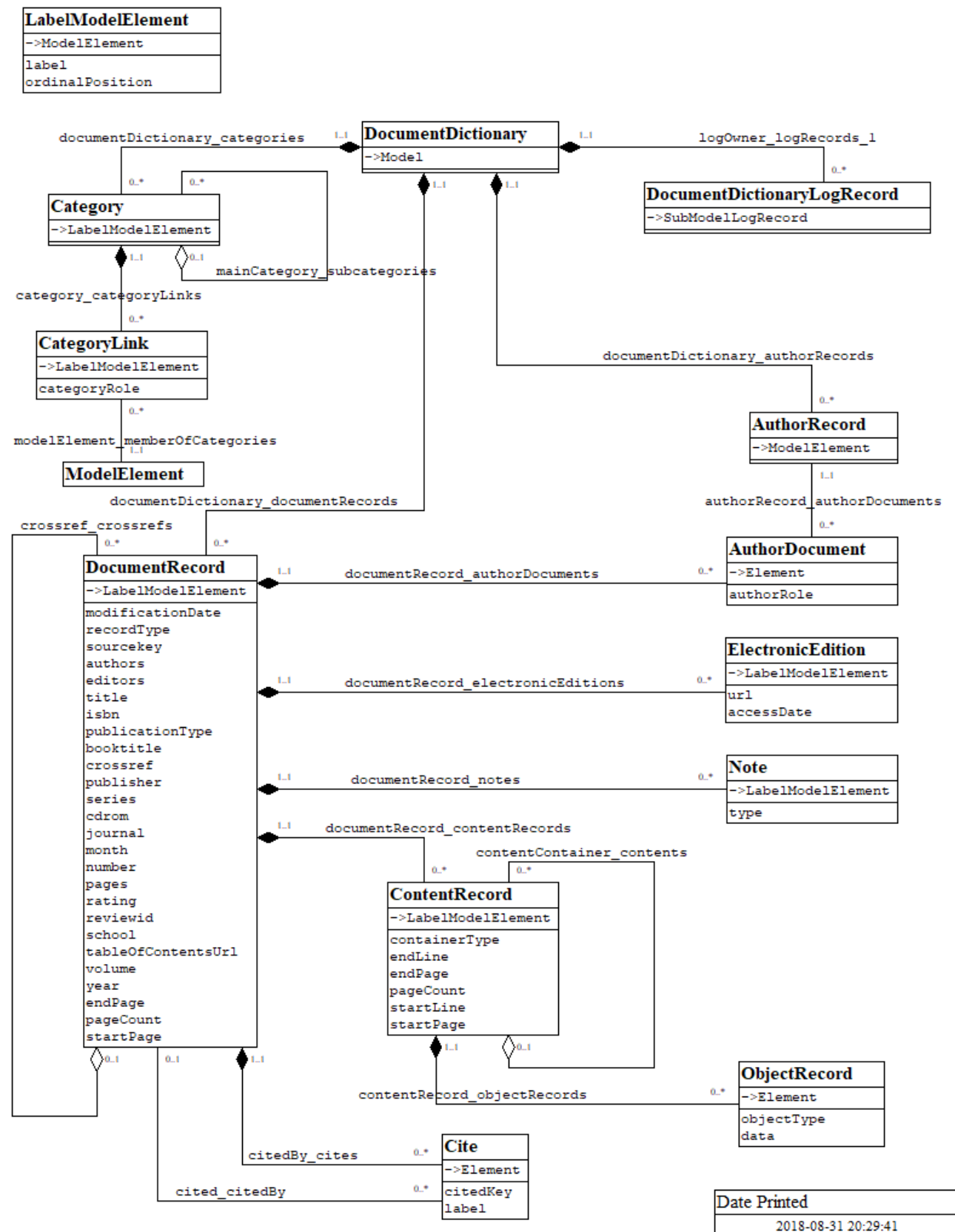
B. References for Exceptional Students

[Card et al. 1983] S. Card, T. Moran, A. Newell, "The Psychology of Human Computer Interaction", Hillsdale, New Jersey: Erlbaum, 1983

[Hubel 1988] A. Hubel, "Eye, Brain and Vision", Scientific American Library, 1988

[Lindsay&Norman 1977] P. H. Lindsay, D. A. Norman, "Human information processing", Academic Press, 1977

C.DomainModel of DocumentDictionary



D.Glossary

GOOD knowledge works in practice within its well defined scope of applicability without adverse side effects.

GOOD knowledge includes being able to identify knowledge as BAD knowledge, when that knowledge is used outside its range of applicability. Example: 1) "Sequential search" works well in the range of 1 to 100 items. Outside that scope of applicability, 2) "Binary search" delivers higher performance. 3) "B-tree search" outperforms "Binary Search" where memory retrieval cost (e.g. disk block access) exceed a certain time limit. Exploiting primary memory databases, efficient hashing, with optimization for exploiting the silicon based virtual memory capabilities and instruction sets of advanced processors and massive parallel GPU's enable still further improvements for fast information management applications.

GOOD Student. A good student is a person who is aware of his/her limited knowledge. Someone who can separate knowledge from his/her own personal identity or ego and evaluate knowledge for its merits and deficiencies within a particular area of application. A good student can study any subject and become master of it.