Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

# A Concept-Centric Environment for Software Evolution in an Agile Context

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Dirk Deridder

Academiejaar 2005 - 2006

Promotoren: Prof. Dr. Theo D'Hondt en Dr. Johan Brichau

# Abstract

*Evolving a software implementation requires that the developer has a good understanding of what the system does. In practice however a lot of domain knowledge about the software that was available when the system was first conceived is only available to the current developer in an implicit form. Furthermore, even when the domain knowledge is available in an explicit form, the developer also needs to know to which part of the implementation the knowledge applies to. Unfortunately this knowledge is often detached from the implementation which forces the developer to re-establish the link retroactively. Moreover, as a result of short term development economics, developers will often neglect to document and couple the domain knowledge about a system in favor of writing code. This results in a passive source of documentation that will quickly become out-of-date. This dissertation presents a concept-centric environment which is based upon a symbiotic integration of the programming environment (code level) and the domain knowledge environment (concept level). It allows a developer to capture domain knowledge in an explicit form, which is coupled to the corresponding implementation. Moreover it enables the active participation of the domain knowledge in providing the functionality of the software. This installs a bi-directional interaction between the concept level and the code level which can be used to create a documented and malleable implementation of a software system.*

# Acknowledgements

First and foremost I would like to thank my promotor Theo D'Hondt. He welcomed me at PROG quite some time ago and gave me the opportunity (and lots of freedom) to explore whatever research path I had in mind. If it weren't for him, there would be no PhD to defend. I probably gave him a hard time proofreading my chapters since I produced them in a rather chaotic order. So thank you Theo.

I am very grateful for the support provided by Johan Brichau during the last phase of delivering this dissertation. His input improved this text a lot so thanks for taking up the role of second promotor.

I would also like to thank my committee members for their input on the first version of this text: Prof. Stephane Ducasse, Prof. Serge Demeyer, Prof. Viviane Jonckers, and Prof. Geert-Jan Houben.

This dissertation is the direct result of being able to work in a group of highly talented and skilled people (and a bit nerd'ish, but hey, nobody is perfect). The reason why early holes in my story are now filled with rocket-proof concrete is mainly a result of testing it out at PROG's ground zero (a.k.a. PROG research meetings). So in retrospect thanks to all of you: Andy Kellens, Brecht Desmet, Coen De Roover, Dirk van Deun, Elisa Gonzalez Boix, Ellen Van Paesschen, Isabel Michiels, Jessie Dedecker, Johan Brichau, Johan Fabry, Jorge Vallejos Vargas, Kris Gybels, Linda Dasseville, Pascal Costanza, Peter Ebraert, Sofie Goderis, Stijn Mostinckx, Thomas Cleenewerck, Tom Van Cutsem, Wolfgang De Meuter, Lydie Seghers, Brigitte Beyens, Simonne De Schrijver.

A special thank you goes to Johan Brichau, Johan Fabry, Sofie Goderis and Isabel Michiels for proofreading and discussing several versions of this text. Wolfgang De Meuter and Maja D'Hondt deserve an extra credit for helping me out at the early beginning when I only had a tentative version of my table of contents. Maja is the one who mentioned the word 'symbiosis' at the right time when CoBro was still like most webpages: 'under construction'. Even though this meant a lot of extra programming, thanks since it has proven to be an extremely powerful concept.

My friend Wim Lybaert deserves a line of his own: thanks for showing me that nothing is impossible (if you use Smalltalk ;-)). We had many interesting discussions during which even the sky was not the limit.

My friends Tom Lenaerts, Luc Speleman, Frank Leemans and Marc Oste also deserve a thank you. We started our studies at this university many years ago after making the city of Antwerp unsafe. Did I ever thank you guys for providing me with the notes of the courses I accidentally missed? Or did we miss them together :-)

Before I write the last two paragraphs: I did not mention all the people that deserve a thank you because there are so many. So if you don't see your name in these acknowledgements, please write it on the dotted line: ...

A big thank you to my parents who supported me during my studies in every possible way.

The last letters I will ever write in this file go to Joke. Thanks for enduring me and for supporting me at the many times in which I was at a complete loss or absence of hope of ever finishing this thing during this lifetime. This dissertation (don't worry I won't write another one) is for you and Fenna!

# Table of Contents


**1 Introduction** **1**
- 1.1 Thesis Motivation . . . 1
- 1.2 Thesis Statement . . . 5
- 1.3 Thesis Approach . . . 6
- 1.4 Thesis Contributions . . . 9
- 1.5 Organisation of the dissertation . . . 10

**2 Software Evolution in an Agile Context** **13**
- 2.1 Software maintenance and evolution . . . 13
  - 2.1.1 Defining Software Maintenance . . . 14
  - 2.1.2 Categories of Software Maintenance . . . 14
  - 2.1.3 Continued Development . . . 16
  - 2.1.4 Software Evolution . . . 16
  - 2.1.5 Software Aging . . . 18
  - 2.1.6 Staged Model for the Software Lifecycle . . . 18
  - 2.1.7 Malleable Software . . . 19
- 2.2 Evolution in Contemporary Development . . . 20
  - 2.2.1 Traditional versus Contemporary Development . . . 22
  - 2.2.2 Model-Centric versus Code-Centric . . . 23
  - 2.2.3 Synchronisation versus Co-Evolution . . . 25
- 2.3 The Key Role of Domain Knowledge . . . 26
  - 2.3.1 Implicit Domain Knowledge . . . 26
  - 2.3.2 Communicating Domain Knowledge . . . 27
  - 2.3.3 Top Information Needs for Evolution . . . 29
- 2.4 Summary . . . 30

**3 Domain knowledge and ontologies** **33**
- 3.1 Domain knowledge . . . 33
  - 3.1.1 Defining the Term Domain . . . 34

3.1.2 About the Knowledge Associated with a Domain . . . . . 37
3.1.3 Positioning Domain Analysis . . . . . 43
3.2 Ontologies . . . . . 48
3.2.1 About Ontology and ontologies . . . . . 48
3.2.2 Different Varieties of ontologies . . . . . 52
3.2.3 Ontologies in Practice . . . . . 54
3.3 Summary . . . . . 63

**4 The Concept-Centric Environment 65**
4.1 Need for a Concept-centric Environment . . . . . 66
4.1.1 Implicit Domain Knowledge . . . . . 68
4.1.2 Detached Domain Knowledge . . . . . 70
4.1.3 Passive Domain Knowledge . . . . . 71
4.1.4 Problem Statement . . . . . 72
4.2 The Concept-centric Environment . . . . . 73
4.2.1 Explicit Concept Level Representation . . . . . 74
4.2.2 Coupled Concept and Code Level . . . . . 79
4.2.3 Active Use of the Concept Level . . . . . 81
4.2.4 Transparency for the Developer . . . . . 84
4.2.5 Thesis Statement . . . . . 85
4.3 Summary . . . . . 88

**5 The CoBro environment 91**
5.1 CoBro Architecture . . . . . 91
5.2 Concept Manipulation Language . . . . . 97
5.2.1 Concept Namespace . . . . . 97
5.2.2 Concept Creation . . . . . 100
5.2.3 Conceptification Process . . . . . 101
5.2.4 Concept and Relation Manipulation . . . . . 104
5.3 Graphical Concept Network Navigator . . . . . 110
5.4 Assorted Tools . . . . . 115
5.5 Default Core Ontology . . . . . 120
5.6 Summary . . . . . 124

**6 CoBro by Example 127**
6.1 Domain Concept Examples . . . . . 128
6.1.1 Retrieving Concepts . . . . . 128
6.1.2 Helper Messages . . . . . 132
6.1.3 Querying Concepts by Accessing Slots . . . . . 134
6.1.4 Lookup of Slots in the Parent Chain . . . . . 137
6.1.5 Creating Concepts and Setting Slots . . . . . 140
6.1.6 A Notion of Groups and Projects . . . . . 143
6.2 Implementation Concept Examples . . . . . 146

6.2.1 Retrieving Concepts . . . . . 146
6.2.2 Querying Concepts by Accessing Slots . . . . . 148
6.2.3 Coupling Domain and Concept Level Concepts . . . . . 151
6.2.4 Active Use of Concepts in the Implementation . . . . . 152
6.3 Summary . . . . . 158

**7 CoBro Applied** **159**
7.1 CoBro: The Power of Malleability . . . . . 159
7.1.1 Media Library: Standard Implementation . . . . . 161
7.1.2 Media Library: Configurations at the Concept Level . . . . 164
7.1.3 Media Library: Adding a Notion of Users . . . . . 168
7.1.4 Media Library: Configuration Validation . . . . . 174
7.1.5 CoBro-Nav: Adding a New Node Visualisations . . . . . 176
7.1.6 CoBro: Adding Terminal Value Interpreters . . . . . 179
7.2 The Origins of CoBro . . . . . 181
7.2.1 SoFa . . . . . 182
7.2.2 eVRT MPEG . . . . . 184
7.2.3 Advanced Media . . . . . 186
7.3 Summary . . . . . 188

**8 Conclusion** **189**
8.1 Summary . . . . . 189
8.2 Conclusions . . . . . 191
8.3 Future Work . . . . . 193

**Bibliography** **197**

**Index** **207**

# List of Figures


2.1 S-type, P-type, and E-type software systems. . . . . 17
2.2 The staged model for a software lifecycle. . . . . 19
2.3 Planning versus Guiding a project trajectory. . . . . 22
2.4 Evolution of models and code. . . . . 25
2.5 Reduced communication quality. . . . . 28
2.6 Mapping an evolutionary need onto an implementation. . . . . 28
2.7 The Brooks code comprehension model. . . . . 30

3.1 Encapsulated and distributed domains. . . . . 36
3.2 Vertical domains and horizontal domains. . . . . 37
3.3 Telebib2 reference dictionary. . . . . 39
3.4 The four modes of knowledge conversion. . . . . 42
3.5 A domain and application engineering lifecycle. . . . . 46
3.6 Domain and application models. . . . . 47
3.7 A layering of ontologies and layers within an ontology. . . . . 54

4.1 Improved communication quality. . . . . 70
4.2 Legend for concept drawings. . . . . 74
4.3 Reification of domain knowledge. . . . . 75
4.4 Frame-based concept representation. . . . . 77
4.5 Virtual layers within an ontology. . . . . 78
4.6 Deification of code level entities. . . . . 80
4.7 Coupling of reified and deified concepts. . . . . 81
4.8 Interacting with concepts from the code level. . . . . 82
4.9 Up / down mechanism for triggering (de)conceptification. . . . . 83
4.10 Active participation of concepts in the code level. . . . . 84

5.1 CoBro overview. . . . . 92
5.2 Interplay of the different meta and base levels. . . . . 94
5.3 Extension of the binding lookup mechanism. . . . . 99

5.4 Conceptification process of code level entities. . . . . 102
5.5 Extension of the message lookup mechanism. . . . . 107
5.6 Lookup of slots in the concept level parent chain. . . . . 109
5.7 The CoBro-Nav graphical concept browser. . . . . 111
5.8 The available CoBro-Nav node types. . . . . 112
5.9 The CoBro-Nav toolbar . . . . . 113
5.10 CoBro-Nav export to GraphViz. . . . . 113
5.11 The extended Star Browser. . . . . 114
5.12 Visualising user interface specifications in CoBro-Nav. . . . . 116
5.13 The CoBro launcher and a Smalltalk workspace. . . . . 117
5.14 The CoBro concept bag. . . . . 117
5.15 The basic concept editor. . . . . 118
5.16 The concept locator. . . . . 119
5.17 The basic concept viewer. . . . . 119
5.18 The CoBro code tool in the Smalltalk refactoring browser. . . . 120
5.19 The CoBro edit tool in the Smalltalk refactoring browser. . . . . 121
5.20 The CoBro-Nav code tool in the Smalltalk refactoring browser. 122
5.21 Part of the upper level of the CoBro core ontology. . . . . 123

6.1 A subset of a simple wine ontology. . . . . 129
6.2 A Smalltalk object inspector. . . . . 131
6.3 A Smalltalk object inspector. . . . . 132
6.4 Concept exploration in CoBro-Nav. . . . . 136
6.5 A concept bag. . . . . 139
6.6 Exploring concepts in the star browser. . . . . 141
6.7 A concept editor. . . . . 143
6.8 A Smalltalk refactoring browser. . . . . 148
6.9 A Star Browser. . . . . 150
6.10 CoBro-Nav on a concept. . . . . 152
6.11 Coupling an image to code. . . . . 153
6.12 A Smalltalk refactoring browser with CoBro-Nav. . . . . 154
6.13 Exploring a concept in CoBro-Nav. . . . . 157

7.1 Media Library class diagram. . . . . 160
7.2 Media Library main window. . . . . 161
7.3 Media Library with concept level configuration. . . . . 167
7.4 Media Library configuration in a Smalltalk Browser . . . . . 169
7.5 The novice and expert user interface for books. . . . . 170
7.6 Different user interfaces according to user expertise. . . . . 174
7.7 A dedicated CoBro-Nav node visualisation for deified classes. . 177
7.8 Nodes for a terminal concept and a terminal value. . . . . 178
7.9 A dedicated node for media library configurations. . . . . 180
7.10 The SoFa Ontology Browser . . . . . 183

7.11 Main characteristics of iMedia software development. . . . . 187

# List of Tables


2.1 Definitions for section 2.1 . . . . . 21
2.2 Definitions for section 2.2. . . . . 26
2.3 Top information needs of software maintainers. . . . . 29

3.1 Definitions for section 3.1.1 . . . . . 38
3.2 Definitions for section 3.1.2 . . . . . 44
3.3 Definitions for section 3.1.3. . . . . 48
3.4 Different kinds of repositories. . . . . 51
3.5 Definitions for section 3.2.1. . . . . 52
3.6 Definitions for section 3.2.2. . . . . 55

4.1 Definitions for section 4.2 . . . . . 86

5.1 Overview of selected core relations and concepts . . . . . 123

# Chapter 1

# Introduction

## 1.1 Thesis Motivation

In order to survive in today's highly dynamic marketplace, companies must show a continuous and ever-increasing ability to adapt. This pertinent need for change has a clear impact on the software systems that support business activities. For example when the business side of a new product or service is conceived, the software side immediately needs to accommodate it. The delivery of new products or services simply cannot be postponed just because the software system is not ready to handle them. Thus a need for change in the business domain requires that the supporting software system be adapted as swiftly as possible. The level of agility that is necessary to keep up with the business side of this 'adaptability challenge' consequently is translated to the software side. This puts contemporary software development practice under a continuous strain to achieve results within an extremely limited time-span.

Incorporating new functionality in an existing system is done during the evolution stage of a software's life cycle. This stage is mainly about continued development, but traditional maintenance activities such as bug fixing also take place. In this dissertation we focus on the continued development of object-oriented applications written in a class-based programming language.

For the remainder of this section we will briefly zoom in on the problems that are tackled in this dissertation. In essence our problem statement is centered around the following keywords:

- **Implicit** domain knowledge
  The domain knowledge about a software implementation and its associated business domain is only available in an implicit form to a developer. This is problematic for code comprehension during the evolution of the system.

- **Detached** domain knowledge
  Even when domain knowledge is available in an explicit form, it is detached

from the implementation. This is problematic since reconnecting the domain knowledge retroactively to the implementation is a difficult process.

- **Passive** domain knowledge
  Domain knowledge is neglected by developers since it plays a passive role with respect to providing the behavior of the application. This is problematic since this results both in more implicit domain knowledge and out-of-date domain knowledge.

- **Overhead** for the developer
  Documenting what is known about a software implementation and its associated business domain provides an overhead for the developer. This is problematic since it only adds to the reasons why domain knowledge becomes implicit.

### Implicit domain knowledge

A request for evolution is expressed by the customer in terms of business concepts and not in terms of implementation concepts. Suppose for example an insurance company that wants to introduce multiple ways in which an insurance product can be cancelled. When expressing this need for change, the business expert will use existing concepts such as `Cancellation Procedure`, `Insurance Object`, `Insurance Type Code`, and `Renewal Main Date`. In the existing implementation this could result in the extraction of the entangled cancellation code into a Strategy Pattern, the refactoring of a number of classes, and the extension of existing user interfaces. Because of the above and in order to be able to perform the required adaptations, the developer hence needs to map the evolution request onto the existing design and implementation. Therefore he needs to have a profound understanding of what the system does. This involves a lot of domain knowledge about both the problem domain and the solution domain. Moreover he needs to know which part of the implementation is responsible for the functionality associated with a certain business concept. This implies a link between his domain knowledge and the corresponding implementation. So, for example, in addition to understanding what a cancellation procedure means, he also needs to find out which code entities take care of the existing cancellation functionality.

Acquiring knowledge about a computer program is the mission of the code comprehension process. The knowledge that is acquired can range from structural information (e.g. what are the senders of a message) to business domain knowledge (e.g. what is an insurance renewal main date). A programming environment can provide automated support to discover the former, but not for the latter type of knowledge. It is estimated that up to 60 percent of software evolution is spent on comprehension [ABDT04]. This is because a lot of knowledge about the software system and its associated business domain is not captured in an explicit form. The majority of this knowledge remains implicit in the brains of

the original developers or is simply lost because the experts have switched company or project team. Even in a model-centric approach to software development, not every kind of knowledge is made explicit. For example the motivation of a particular design decision is usually not documented in the models. This is why a lot of time is spent on rediscovering what was known by the original developers.

*In order to reduce the possible loss of domain knowledge, we need a mechanism that makes it possible to capture this knowledge in an **explicit** form.*

### Detached domain knowledge

Not only domain knowledge is important, but also the link to the corresponding implementation entities. Empirical evidence shows that developers that evolve an existing implementation find this information extremely important [KSP04]. Mapping domain knowledge onto code is difficult in nature. This is in part because domain knowledge cannot always be identified with one specific location in the code. Most of the time the corresponding functionality is scattered throughout the implementation. Since this mapping process is extremely time-consuming, it is important that a developer can attach his findings to the code once this mapping is complete. Also, even when a case tool is used to model the software, the problem of the missing link still exists. This is the result of the dichotomy between the traditional analysis-design-implementation phases. In general this boils down to the fact that the artifacts are decoupled each time control is passed from one phase to another. Consequently even though the knowledge remains available in an explicit form, in the end it becomes detached from the implementation.

In addition, to make a software implementation more flexible towards possible future change, developers strive towards a malleable implementation. One basis for achieving such malleability is to strive towards genericity in the code (a.k.a. keeping the software soft). Genericity can be achieved by pushing as much of the code as possible towards the data level. An example is loading the different insurance cancellation procedures at runtime in a dictionary instead of making them explicit in method bodies. Unfortunately such malleability comes at a price. In general it does not improve readability of the code since a lot of details, as well as the intention of the code, become hidden in the data. This results in less explicit code, a detachment of the domain knowledge from the code, and consequently more implicit domain knowledge.

*In order to reduce the effort spent by developers on retroactively matching domain knowledge to its corresponding implementation, we need a mechanism that makes it possible to* ***couple*** *the domain knowledge to the implementation.*

### Passive domain knowledge

Spending time on documenting the domain knowledge for a software system is time that cannot be spent on development. This is why documenting is often neglected by developers in favor of writing code that contributes to the next release of the system. As a consequence the domain knowledge is not maintained properly and will quickly become out-of-date. This phenomenon is even more pronounced in agile development environments where short delivery cycles are the norm.

The reason why domain knowledge does not directly contribute to the functionality of the system is because it plays a passive role with respect to the execution of the software. Moreover there is no straightforward mapping that can be automatically applied to transform domain knowledge into an implementation and vice versa. This is in part due to the often very informal nature of domain knowledge and because it cannot always be brought back to one specific location in the code. As a consequence the domain knowledge will quickly become out-of-sync with the implementation and vice versa. Therefore the usefulness of the explicit domain knowledge will rapidly degrade over time.

*In order to motivate developers not to neglect the domain knowledge, we need a mechanism that makes it possible to involve the domain knowledge* ***actively*** *to provide the functionality of the software system.*

### Overhead for the developer

As we already alluded, developers perceive documenting what is known about the implementation as an overhead. This is not only due to the fact that domain knowledge is a passive source of information. It has also to do with the fact that writing down the domain knowledge associated with a system is often done outside the development environment. As a consequence developers need to switch frequently between environments. This results in an overhead during development which discourages developers to keep the domain knowledge up to date.

*In order to motivate developers not to neglect the domain knowledge, we need to implement the interaction with the domain knowledge environment as **transparently** as possible.*

**Summary**

Software developers responsible for evolving an existing implementation must grasp the intentions and assumptions made by the original developers of the system. This is a problem because most of the domain knowledge about the system is only available in some *implicit* or tacit form. Even in a context where such domain documentation exists in an explicit form, this documentation is *detached* from the current implementation. This means that the programmer will have to find out manually which part of the documentation matches with which part of the implementation. The main reason why domain knowledge becomes implicit and detached is because it does not yield a short-term economic benefit for the developers. As a consequence it acts as as *passive* form of documentation. Consequently it is not maintained properly and will quickly become out-of-date. This is mainly due to the fact that developers perceive documenting what is known about the implementation as an *overhead*. The problems discussed above are even more pronounced in the context of agile software development, where the next release of the system is only hours away.

## 1.2 Thesis Statement

In this thesis we present a concept-centric environment in which domain knowledge is represented in an explicit concept level. This significantly reduces the potential loss of domain knowledge and acts as a source of documentation for developers during evolution. The concept-centric environment empowers developers to make the coupling between entities that reside in the concept level and entities that reside in the code level explicit. This reduces the effort spent by developers on rediscovering the link between domain knowledge and the implementation. Furthermore domain knowledge that is captured in the explicit concept level also plays an active role during the execution of the software. This means that the concept level contributes to the functionality of the next release of the system. Since maintaining the concept level now yields a short term benefit (as opposed to the long term promise for future evolution efforts), developers will no longer neglect it in favor of writing code. Moreover the activation of the concept level can be used to enhance the malleability of the software. This is achieved by refactoring the code level to make use of the concept level. As a consequence certain adaptations become possible by changing the concept level instead of the code

level. In addition to this, the environment is conceived in a way that it is not perceived by developers as an overhead. This means that the interaction with the concept-centric environment is as transparent as possible with respect to the tools and formalisms used. This is realised by a symbiotic integration of the concept level and the programming environment. In summary the thesis statement for this dissertation is formulated as follows:

> *A **symbiotic integration** of a programming environment with*
> *an **explicit concept level** that is **coupled** to the code level*
> *supports **domain knowledge documentation***
> *in the context of **agile evolution***
> *and amplifies the **malleability** of software through*
> ***active** concept level **participation**.*

We will refer to a system as a **concept-centric environment** if it corroborates the thesis statement.

## 1.3 Thesis Approach

The conceptual framework for a concept-centric environment is based upon the following antonyms for the keywords of section 1.1:

- From implicit towards **explicit**
- From detached towards **coupled**
- From passive towards **active**
- From overhead towards **transparency**

#### From implicit towards explicit

We have seen that most domain knowledge in software systems is only available in an implicit form, which impedes the programmer's understanding of the system. So when a programmer tries to comprehend the intended meaning of a piece of code, he will need to fall back on the own domain expertise or the domain expertise of others. Since such domain knowledge forms an essential ingredient for performing software evolution, it is of the utmost importance to make it available in a more reliable form. Therefore we set up an infrastructure that is used as an ontology to represent this valuable source of information in an explicit way. The repository that holds the domain knowledge is referred to as the ConceptBase

of the concept-centric environment. The concepts are described using a *frame-based representation* that relates concepts to each other in a *concept network*. A *definition frame* for a concept is created that contains *slots* that hold a relation (which is also a concept) and a number of destinations. The destinations can be concepts or terminal values, depending on the chosen *ontological granularity*. The resulting set of slots is referred to as the *extension* of the concept. In order to minimise the cognitive overhead perceived by developers we use the same syntax and interaction mechanisms of the underlying programming language to manipulate the concept level. As a consequence, interacting with the concept level or the code level is transparent for a developer with respect to both aspects. We follow a *prototype-based approach* to avoid a developer having to commit to a premature structure for modeling the concept level. This implies that concepts are defined *ex-nihilo* (from scratch) or by *cloning* of an existing concept definition. This maximises the flexibility for domain concept reification since it is not necessary to switch between a class and an object view to alter a concept definition.

**From detached towards coupled**

Unfortunately, having access to an explicit source of domain knowledge presents only a partial solution to the problem. As we have seen, a programmer still needs to map this knowledge to the implementation (a.k.a. the code level), mostly in a retroactive way. This is unfortunate since there probably existed a clear link between the knowledge and its implementation when it was first conceived. Therefore we establish a coupling between the concept level and the corresponding code level in a proactive way. This means that we provide support for a developer to connect the entities of both levels continuously during development.

In order to be able to couple code level entities to concept level entities we automatically *deify* code level entities towards the ConceptBase. The process that generates a concept representation for a code level entity is referred to as *conceptification*. It is based on two decisions with respect to the granularity of the concept level representation: *what* to represent about the code entity (e.g. superclass, instance variables, methods) and *how* to represent it at the concept level (e.g. represent its methods as concepts or terminals). A prescription that defines what to represent and how to represent it is used to compute the *intension* of a concept. Computing this intension is done by using the introspective capabilities of the programming language.

A coupling between code level entities and domain knowledge is obtained by providing an extension for a conceptified code entity. This boils down to relating the conceptified code entity to other entities at the concept level by extending its definition frame with the necessary slots.

### From passive towards active

Since no universal function exists that can transform the concept level into the code level and vice versa, special care must be taken to make sure that both levels do not become out-of-sync. Since the economics of performing changes at the code level will yield the best short-term benefits, programmers will tend to neglect the concept level. In general they will focus on getting the code right since this results in an immediate improvement of the overall behavior of the application. Hence allocating a lot of effort to update this passive promise of documentation usually has a very low priority. As a consequence the concept level will quickly become out of sync with the code level. As a result it will be of little or no use for future evolution efforts. Therefore it is important to activate the domain knowledge. By this activation we mean that it contributes to the functionality of the software as much as possible. Thus we provide a means to write concept level statements within normal code that consult, create, update, or delete concept level entities. Moreover, the conceptification of a code level entity can be triggered from within the code by using an up/down mechanism. This enables a developer to write code that consults its own concept representation in order to perform a given task. It also lets a developer take advantage of the concept level relationships between code level entities.

By factoring out explicit decisions in the code to the concept level, the active participation of the concept level results in improved malleability of the implementation. Even though this results in less explicit code, the ConceptBase holds the explicit representation of the associated concept network. Consequently the concept level will act as documentation for future developers by revealing what is no longer visible at the code level. Moreover this results in a co-evolutionary situation that is beneficial for both the concept and the code level. An application that is built with an active concept level is characterised as *ontology-driven*.

### From overhead towards transparency

In order to promote making explicit what one knows, as well as coupling it to the implementation and using it actively from within the code, it is necessary to ensure that programmers don't perceive the concept level as an extra overhead. Part of the problem is a result of the antibiosis between the concept and the code level: spending time at the concept level results in less time available for coding and vice versa. Therefore we set up an environment in which both levels exist in symbiosis with each other. This way the use of both sides becomes as transparent as possible for the programmer. This is realised by making sure that the interaction with objects and concepts is done in a similar fashion. Hence concepts are manipulated by sending messages, and the syntax used to express this is the same as for the programming language. Transparency is further obtained with a close integration between the concept environment and the programming environment.

This means that all the tools available for interacting with the concept level are closely integrated with the existing programming tools.

**From vision towards practical implementation**

A proof-of-concept environment (CoBro) was implemented that exhibits the necessary elements to be justified as a concept-centric environment. CoBro is implemented in Smalltalk and uses a relational database to hold the ConceptBase. It is implemented in close symbiotic integration with Smalltalk. This implies that our concept language (CoBro-CML) uses the syntax and interaction mechanisms of Smalltalk. Moreover all the CoBro-tools are closely integrated with the existing Smalltalk development environment. CoBro is used as a vehicle to validate the practical attainability of setting up a concept-centric environment as proposed in this dissertation. When the initial version of CoBro was completed, we refactored elements of it to enhance the malleability through active concept level participation. As a consequence CoBro was applied to itself to test the validity of our claims. CoBro was also applied during our work performed for research projects in collaboration with industry. As a consequence, the concept level, as well as the CoBro environment were refactored on a regular basis when new insights were gained. These practice-driven adaptations to the CoBro environment make it into what it currently is: a highly flexible concept-centric environment which is based on a lightweight conceptual model.

## 1.4 Thesis Contributions

The main contributions of the research presented in this dissertation are:

- **Identification of the Need for an Explicit Concept Level**
  An analysis is performed that identifies the key role and associated problems of domain knowledge for software evolution in an agile context.

- **Concept-centric Environment**
  A conceptual framework for a concept-centric environment is set up in which domain knowledge can be made explicit, coupled to the code, and used in an active way. Obliviousness for the developer is accomplished with a symbiotic integration between the programming environment and the concept environment.

- **Concept-driven Malleability**
  A mechanism is conceived that enables a developer to invoke, at runtime, the concept level relationships that exist between code level entities. Moreover a means to achieve software malleability is provided that eliminates the drawback of less explicit code. In addition, support is offered to developers to set up the infrastructure for creating a malleable software implementation.

- **CoBro**
  A highly extensible proof-of-concept environment named CoBro is introduced and used as a vehicle to validate the feasibility of a concept-centric environment. Furthermore the prerequisites to implement a concept-centric environment are established.

## 1.5 Organisation of the dissertation

**Chapter 2: Software Evolution in an Agile Context**

Introduces agile software evolution to position the research presented in this dissertation. Moreover it identifies implicit and detached domain knowledge as problematic in the context of agile software evolution.

**Chapter 3: Domain Knowledge and Ontologies**

Dedicated to an in-depth discussion of the topic of domain knowledge. Since the approach presented in this dissertation is inspired by ontology research, we introduce the elements of importance to this research.

**Chapter 4: A Concept-centric Environment**

Provides a detailed discussion of the problems mentioned in the introductory chapters. Moreover it introduces the foundations required for the concept-centric environment we propose in this dissertation.

**Chapter 5: The CoBro Environment**

Zooms in on the practical environment we implemented as a proof-of-concept in order to validate the claims made in this dissertation. It provides a technical discussion of the different elements that constitute the CoBro environment.

**Chapter 6: CoBro by Example**

Illustrates the basic interaction with the CoBro environment. This is done

in an example-based fashion, where each example will address a particular aspect of CoBro.

**Chapter 7: CoBro Applied**

Illustrates the use of CoBro with a number of advanced applications so as to validate the claims of this dissertation.

**Chapter 8: Conclusions**

Presents the conclusions of this dissertation. We start by summarising the problem statement and contributions, after which we discuss our conclusions. The dissertation is ended with an elaboration of future research directions.

# Chapter 2
# Software Evolution in an Agile Context

The evolution of a software system can be interpreted in many ways, ranging from manual bug-fixing to self-adapting software that can alter its behavior depending on changes in the environment. In order to position the contributions of this dissertation, we dedicate this chapter to describing a characterisation of software evolution. Also we discuss why having access to the domain knowledge of the original developers is a key to successful evolution.

In section 2.1 we briefly stipulate a number of definitions for software maintenance in concert with the concept of software evolution. After this we will zoom in on the different categories of maintenance activities and on the notion of continued development. We will also position software evolution with respect to software aging and how it fits inside the staged model of the software lifecycle. We end the section by introducing the concept of malleable software, and the difference between run-time and development-time malleability.

Section 2.2 positions the evolutionary stage of software within contemporary and traditional development approaches. We broadly characterise current development approaches as being either code-centric or model-centric. We conclude with a discussion of the difference between synchronisation and co-evolution of code and models.

In section 2.3 we discuss the importance of domain knowledge for successful software evolution. Since a lot of knowledge becomes implicit after initial development, we will elaborate on the main causes for this. We end the section with an investigation of the different kinds of knowledge that are needed the most by maintainers. The topic of domain knowledge will be covered extensively in chapter 3.

## 2.1 Software maintenance and evolution

The term *software evolution* is not just a replacement for the nowadays less fashionable term *software maintenance*. Even though the former was coined more

recently and is often used as a hypernym for the latter, one cannot plainly state that it constitutes a conceptual superset. This section is dedicated to pinpoint the actual difference between both software evolution and maintenance in order to narrow down the scope of our problem domain.

### 2.1.1 Defining Software Maintenance

The IEEE1219-1998 standard [IEE98] defines software maintenance as *the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.*

One of the key statements in this definition is that it mainly concerns modifications that take place *after* the delivery of the system. A similar definition, in which this post-delivery nature is stressed, is provided by the ISO/IEC 14764 standard [ISO99]. It does not exclude however pre-delivery activities that are undertaken to facilitate post-delivery maintenance. An example of such an activity is the operational planning of maintenance in which the necessary elements (e.g., help desk, software change request tool) are established to accommodate the future maintenance activities.

The post-delivery nature as well as the reference in these definitions to faults or problems suggests that software maintenance could be characterised as the set of activities in response to some kind of *software warranty* period. The warranty period to which we refer would constitute a guarantee for the customer that the developer is going to repair or replace defective software within a certain period of time if necessary. Similar to a warranty period for hardware, the defects that are covered during this period are restricted to defects in materials (e.g., bugs in the source code) or workmanship (e.g., software does not function as specified). With respect to software, this warranty period translates into a fairly short timespan *during* the delivery stage [ABDT04]. In contrast, the maintenance phase consists of a prolonged period of time (until the end of the software's lifetime) during which the software system is actively supported. Generally speaking the main purpose of maintenance is to keep the system up and running after delivery. The defects that are repaired by modifications to the system during this prolonged period of time include a lot more than bugs. Hence maintenance goes beyond what is normally understood under a warranty period. This becomes clear when looking at the large body of work on maintenance taxonomies.

### 2.1.2 Categories of Software Maintenance

There are a lot of taxonomies for software maintenance. Most of these approach the topic from a '*why*'-perspective, which means that the maintenance-space is partitioned based on the cause or *purpose* of the maintenance. This is clearly indicated by the extremely popular bases of maintenance introduced by Swanson in 1976 [Swa76]. This work classifies maintenance activities into a corrective,

perfective, and adaptive category. Later on these categories were extended with a fourth one named preventive maintenance [ISO99]. These categories are still used as the major categories in most works of standardisation.

*Corrective maintenance* is mainly concerned with fixing failures in a system. In essence the main causes that lead to corrective maintenance are processing, performance, and implementation bugs. Under normal circumstances these should have been discovered and fixed before or during the warranty period of the system. Yet certain bugs will only show up in exceptional cases or when the system has been running for a longer period of time.

All activities that are undertaken to enhance the overall performance of the system fall under the category of *perfective maintenance*. Perfecting a system can be achieved in many ways. For example by implementing better algorithms, improving the readability of the reports generated, or by restructuring the code in order to enhance its maintainability. In the original definition by Swanson these perfective actions were intended to occur within the limits of the original specifications.

The third category of *adaptive maintenance* encompasses all changes that are needed when the operating environment changes. In the original article by Swanson, the operating environment referred to is either the data environment (e.g., a database system) or the processing environment (e.g., the operating system). Changes to one of these are considered as triggers for adaptive maintenance. In essence this has to do with the portability-aspect of the software. Since the activities in both the adaptive and the perfective category are not about corrections but about improvements, they are sometimes grouped under the heading *enhancements* [Pig96].

The last category of *preventive maintenance* is the proactive counterpart of the reactive category of corrective maintenance. It includes all activities taken before the actual need for correction arises in the post-delivery stage. Hence this includes all activities that are performed in order to avoid erroneous behavior of the system in the near future. These actions occur most often in the context of safety-critical systems.

As we have already indicated these categories reflect the 'why'-perspective of software maintenance. Alternative approaches exist such as [MBRZ02] in which a taxonomy is organised based on temporal properties (when), system properties (what), objects of change (where), and change support (how). Another interesting work is [CHK+01] where the four basic categories are re(de)fined into twelve different types of software evolution and maintenance, namely training, consultive, evaluative, reformative, updative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive.

### 2.1.3 Continued Development

Even though many researchers have adopted the terminology of Swanson, few of them use it in line with the original definitions [CHK+01]. This has led to a lot of confusion and debate in the field. One often notices that the activity of adding or altering features (beyond the original specifications) is sometimes erroneously classified under adaptive maintenance. Yet as we have described above, it is not a change to the business environment but to the data or processing environment that triggers adaptive maintenance. In some cases this cause for change is also classified under perfective maintenance. This is because the authors see the addition of new features as 'perfecting' the system to better fit a user's needs. Swanson already coined the term *continued development* for these kinds of software change. The fact that a lot of maintenance activities are actually about continued development is illustrated by research that focuses on the cost of maintenance. A well-known study by Lientz and Swanson analysed the life-cycle costs of maintenance [LS80]. In their survey they reported that only 21 % of the maintenance effort was dedicated to corrective maintenance. This stands in shrill contrast with the common perception that the activities that are labeled as maintenance are only about fixing errors. Other, more recent studies show similar results and report that over 80 % of the expenditure is for non-corrective actions [ABDT04, BR00, Pig96, CC05]. In essence a large amount of work that is classified under maintenance is actually about continued development, which brings us to the topic of software evolution.

### 2.1.4 Software Evolution

A standardised definition for the term *software evolution* is lacking [BR00], yet continued development (as well as changes to the system to facilitate this) is generally understood as its main focus. The most well-known work in the field is by Lehman, who postulated eight laws for software evolution [Leh96]. Moreover he characterised software systems as S-, P-, and E-type systems. The different definitions of these types are based on the way in which the systems interact with their environment as well as the degree that the environment and underlying problem can change. We will briefly introduce this categorisation since it allows us to indicate the kind of software we are targeting at a later stage.

The *S-type system* is the most static one. It tries to solve problems that are completely defined and for which one or more practical solutions exist (e.g., matrix calculations). The difficulty for this kind of system does not lie in finding a correct solution, but lies in obtaining a correct implementation for such a solution. Even though the problems solved by an S-type system stem from the real world, a change in the real world doesn't cause the existing implementation to evolve. It will result in a completely different problem and consequently a completely different implementation.

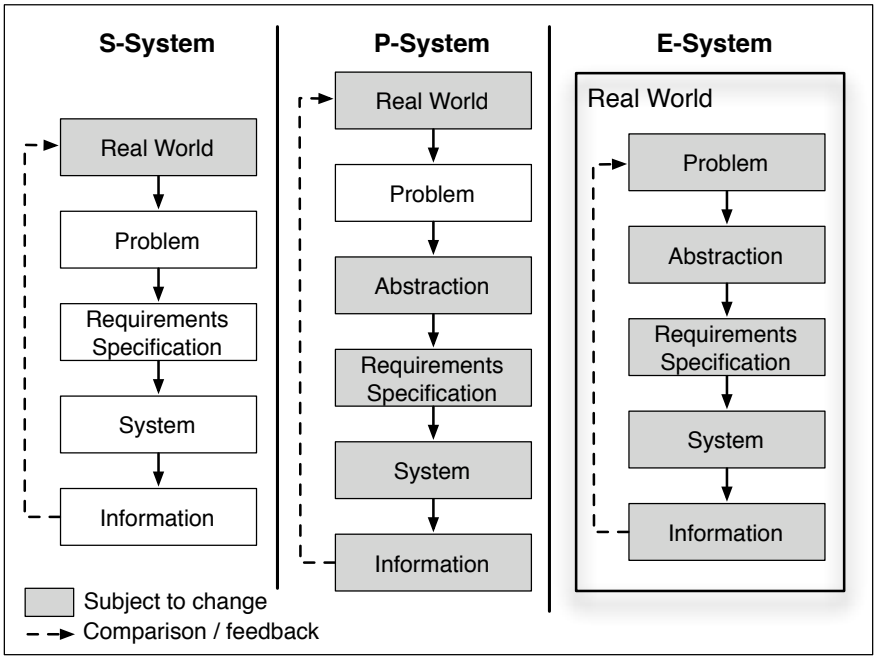



Figure 2.1: Lehman's characterisation of software as S-type, P-type, and E-type systems.

A *P-type system* is similar to an S-type system in the sense that it provides an implementation for a problem that can be completely specified (e.g., the rules of a game of chess). The difference lies in the fact that there does not always exist a practical solution to the problem that it addresses. In essence this means that only an approximate solution to the problem is feasible in an acceptable amount of time (e.g., to cover the large amount of possible moves in a game of chess). This implies that the implementation is susceptible to evolution if a solution is found that is a better approximation of the solution to the problem.

*E-type systems* are the most dynamic kind of systems recognised by Lehman. These systems are connected to the real world in a sense that if the real world changes, these systems also need to change. For example an insurance system is based on an abstraction of the real world laws that govern this business. If these laws change then the abstraction of it must change, as well as the corresponding implementation. This type of system is usually in a state of constant change, where its performance is under continuous evaluation by its users. Moreover the problem that is being solved simply cannot be 'completely' specified. In figure 2.1 we present a schematic representation of the three kinds of systems [Pfl98].

With respect to E-type systems, Lehman postulated eight laws of software evolution [Leh96]. We will only state the first law which is of importance here to illustrate the difference between maintenance and evolution once more. This

law is about *continuing change*, which means that *an E-type program that is used must be continually adapted else it becomes progressively less satisfactory.*

This need for continuous adaptation is a two-way interaction between changes in the real word and changes to the application[1]. First of all this means that a change in the real world should be reflected by a corresponding change to the software. For example changes to existing software to support the introduction of new insurance products. On the other hand, adapting the software could also have an impact on the real world. For example the existence of the software makes it possible for the insurance company to offer a wider range of products. As is shown by these examples, the kind of adaptation referred to is not about maintenance. It is about *continued development*. This implies changes to the software that extend its functionality as well as changes that improve the software's capability to accommodate possible future adaptations.

### 2.1.5 Software Aging

Software can age, yet unlike hardware it will not wear out through continuous use or of old age[2]. The main causes for software aging are characterised by Parnas as a *lack of movement* of the software and *ignorant surgery* to the software [Par94]. In short, the first cause of software aging refers to the fact that the software no longer meets today's requirements. Consequently its users will become dissatisfied and may want to replace it by a completely new software product as soon as possible. This is in line with Lehman's first law of evolution.

The second cause, named ignorant surgery, refers to software aging as a result of the way that the software was changed. This manifests itself specifically in the case where the changes were made by people who didn't understand the original design concept. Consequently the internal quality of the software decays up to the point where even the original developers would no longer understand it. In both cases, the aged software will eventually die.

### 2.1.6 Staged Model for the Software Lifecycle

This observation indicates that a software application can be in one of several stages during its lifetime. In figure 2.2 we illustrate a *staged model for the software lifecycle* by Rajlich and Bennet [RB00].

During *initial development*, a first working version of the software is developed. Moreover during this stage the development team builds up a large amount of knowledge about the problem domain. In the *evolution stage*, the developers will

[1] See our discussion of synchronisation and co-evolution in section 2.2

[2] Parnas identifies a problem similar to the wear of hardware as 'kidney failure' [Par94]. Yet this kind of problem is distinct from *software aging* since it actually refers to erroneous implementations. An example of this is a problem of degrading performance which could be avoided by deallocating memory or by archiving parts of a database in a timely fashion.

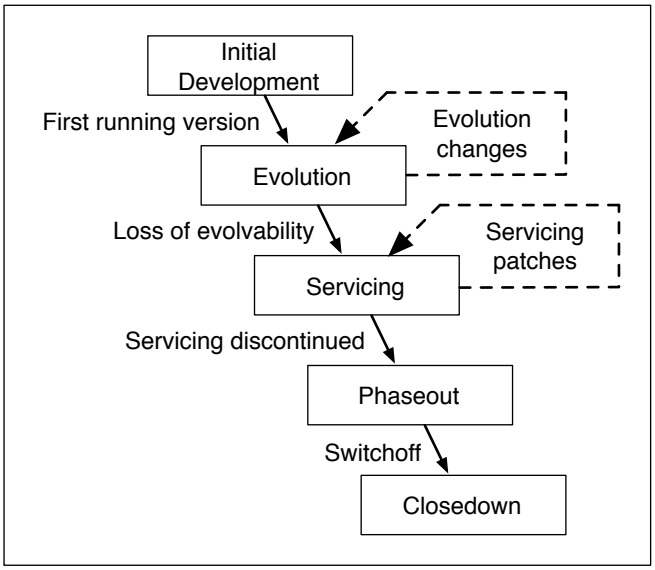



Figure 2.2: The staged model for a software lifecycle.

extend the functionality of the software. This stage is mainly about continued development, but traditional maintenance activities will also take place. This also includes refactoring the system to anticipate future evolution efforts. The domain expertise of the developers is critical during this stage. The *servicing stage* occurs when the system is still supported but no longer viable to large changes (i.e., it shows clear signs of software aging). Developers will limit themselves to minor bug fixes and small functional changes. *Phaseout* occurs when it is decided that no further investment is put in extending the software (i.e., keep it running until it is no longer useable). This is the stage in which the software will quickly become outdated and according to the law of continuing change, will soon be unsatisfactory from a user's perspective. The last stage that a software product passes is *closedown*, which means that it is discontinued and will be replaced by a new application if needed.

### 2.1.7 Malleable Software

As we have already indicated, continued development is also about making the software more susceptible to possible future changes. This implies refactoring certain parts of the software to anticipate unanticipated evolutions in the future. Even though this is a contradiction in terms, one is nevertheless able to create the software in such a way that it becomes easier to incorporate certain anticipated evolutions. Enhancing the evolvability of a system has a direct impact on its life-span. Some authors believe that the envisioned life-span should be specified as part of the requirements [Voa98]. Hence giving the developers a way to evaluate

the costs and benefits of providing a malleable implementation. Malleability can be achieved in many ways, for instance by using framework technology or by using a technique called dynamic object models [RTJ05]. Note that malleability encompasses configurability. It refers to the ease by which the software can be manipulated to behave differently than originally intended, which refers to data (e.g., new configuration data) as well as code (e.g., specialisation and extension of a framework). Configurability on the other hand is about different usage scenarios that were anticipated beforehand. For this purpose all the different configuration options are already included in the software.

A basis for a malleable system lies in striving towards *genericity* in the code. Or as put by Martin Fowler it is about trying to 'keep software soft' [Fow98]. Genericity can be achieved by pushing as much of the code as possible towards data[3]. Consider for example a person who wants to buy an insurance product. All the details for this person could be explicitly represented as attributes in a class Person. Adding a field results in an extension of the code to accomodate an extra attribute. A more generic solution would be to have a dictionary to store all the person's details. Adding an attribute in such an implementation doesn't require any code to be changed. However, even though genericity in code implies that certain extensions are easier to do, in general it does not improve the readability of the code. This is because a lot of details as well as its intention are 'hidden' in the data [Fow01]. Hence there is an unfortunate tradeoff between genericity and simplicity.

#### Development-time versus Run-time Malleability

Since a generic implementation implies that no code needs to be changed for certain adaptations, one could speak of *run-time malleability*. This means that there is no need to go into a development-state, but that changes can be applied without having to stop the execution of the software. This is not only a result of the way in which the code was written. Dynamic development environments such as Smalltalk provide a lot of support for run-time adaptations. *Development-time malleability* on the other hand refers to a situation in which the software must be stopped, and opened up for the changes to take place. Consequently the code moves from its running state to a development state.

## 2.2 Evolution in Contemporary Development

In the previous section we briefly introduced a staged model for a software's lifecycle. As was shown, an evolutionary stage is entered after the introduction of the first version of the software. In this section we will zoom in on the way software is developed, and how it reaches this stage of evolution.

[3]Note that code can also be treated as data.

| | |
|---|---|
| *Software maintenance* | The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. |
| *Software evolution* | The stage in a software product's life that follows to initial development. All modifications (e.g., extend, change, and delete functionality) to the software during this stage are considered to be evolution (a.k.a. continued development). |
| *Corrective maintenance* | The modification of a software product to correct errors after they occur (reactive). |
| *Perfective maintenance* | The modification of a software product to improve the performance or maintainability. |
| *Adaptive maintenance* | The modification of a software product to make it usable in a different software environment. |
| *Preventive maintenance* | The modification of a software product to correct errors before they occur (proactive). |
| *S-type system* | A software system that solves problems that are completely specified and for which one or more practical solutions exist. |
| *P-type system* | A software system that solves problems that are completely specified but for which there does not always exist a practical solution. |
| *E-type system* | A software system that solves problems that cannot be completely specified and for which the solution is continually adapted in accordance with the real world. |
| *Malleable software* | Software that is implemented in a way to make it more susceptible to future changes, either at runtime or at development-time. |

Table 2.1: Definitions for section 2.1

### 2.2.1 Traditional versus Contemporary Development

The role of continued development within software development practices has changed throughout the years. Traditional development approaches followed a fairly sequential, non-iterative execution of the analysis-design-implementation triplet. Customer involvement was rather low after the initial analysis phase. And at the end of the triplet, the customer would be confronted with the complete software product. The requirements for the product were fully known and specified before any actual coding was done. If this specification was not in sync with what the customer originally intended, due to the rigid approach taken, this would be catastrophic to the results produced. In such a context, continued development refers to the actions taken to adapt the existing functionality to accomodate a new business situation after-delivery. Traditional software development is often characterised as bureaucratic. This is because everything must be played by the book, and a high volume of paperwork accompanies the proceedings.

Contemporary development practices deviate from this in a sense that they follow an iterative approach with high customer interaction during the entire development cycle. Consequently they are more open to changes if errors occurred in the specification of the original requirements. Certain approaches such as *agile software development* even thrive on this by eliciting and refining requirements 'on the go'. The software hence grows incrementally and is constantly being refined. Moreover it is used actively to communicate the developer's understanding to the customer. Another discriminating factor is the point in time where coding occurs. In traditional development approaches coding was postponed as long as possible. Nowadays coding is pushed to the front of the lifecycle queue (e.g., for rapid prototyping) as much as possible since it is a good tool to validate all sorts of assumptions made by the developers. In such a context, the system is in a continuous state of evolution. So in the staged model in figure 2.2, the state of evolution also applies during initial development.

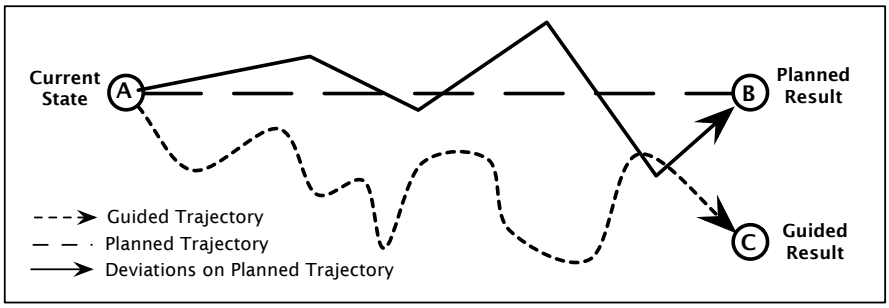



Figure 2.3: Planning versus Guiding a project trajectory (adapted from [HI00]).

Another distinction between traditional and contemporary development is made by Highsmith as planning versus guiding [HI00]. This relates to the way

that the development team works towards the final software product (see figure 2.3).

In a *planning approach* (traditional), a fixed trajectory is planned beforehand to reach a certain predefined result. In an ideal world, one should just follow the planned path to go from the current state (figure 2.3, A) to the planned result (figure 2.3, B). Hence, deviations from this trajectory are considered as mistakes that should be corrected as soon as possible.

The *guided approach* (contemporary) has an exploratory nature. The trajectory is not fixed beforehand, but is guided towards the desired result (figure 2.3, B). Deviations are not considered as harmful but as a necessary good for exploratory purposes. Following such an approach does not necessarily result in the planned result (figure 2.3, B), but in an alternative result that is much closer to what the customer actually needs (figure 2.3, C). The guided approach characterises the overall development approach in agile software development practices. It is not a coincidence that Highsmith named his work *adaptive software development*. Even though the guided approach appears to be an anti-planning approach, this is not the case. It is considered as a more realistic way to obtain software that accurately fits the needs of the customer. This is a result of acknowledging that developers and customers often don't know enough at the starting point (figure 2.3, A).

### 2.2.2 Model-Centric versus Code-Centric

Contemporary development approaches can be characterised as following either a code-centric or a model-centric vision. Broadly speaking this characterisation refers to the relative importance that is attributed to either the code or the models.

For *model-centric* approaches, the models are the primary artefact during development. Code is seen as secondary, and depending on the actual approach it is often automatically generated from the models [Weg02]. This of course implies that the model fully captures all the necessary implementation details. At the extreme end, the *waterfall method* is one of the prime model-centric examples. Every aspect of the system needs to be fully specified before any implementation takes place. Model-Driven Development (MDD) is a recent example of a model-centric approach. In MDD, models are built for the problem domain and the solution domain, which are consequently interpreted or from which code is generated.

In *code-centric* approaches, the code is the primary artefact during development. Models in this case are considered as secondary and are only built if they really contribute to the problem at hand [Weg02]. This is in shrill contrast with the model up-front approach of the model-centric vision. A good example at the extreme end of the spectrum is *ad hoc coding*. This approach to software development boils down to simply writing code without any planning or modeling.

Adaptive Software Development is a recent example of the code-centric vision. In Adaptive Software Development, requirements are elicited in an iterative fashion by using a guided approach. Note that being code-centric does not imply that the developers do not design the software. What it does imply is that the design is primarily communicated through the code instead of by other artifacts.

With respect to *documentation*, both visions counter each other. On the one hand opponents of the code-centric vision attack the clear lack of models. In their view, no models means no documentation. This is clearly troublesome if the code needs to be maintained at a later stage. Yet proponents of the code-centric approach will often state that the best (most trustworthy) documentation is the code itself. This is known by the notion of *code as documentation*. This does not mean however that the code should be the only documentation. Certain aspects of a software's design are not always apparent in the code. Also information that helped the developers to better understand the problem space, but that is no longer of importance for the computation, should be captured. Moreover as Martin Fowler [Fow05] states: 'Like any documentation, code can be clear or it can be gibberish'. We will come back to this in section 2.3.

On the other hand, opponents of the model-centric vision could attack the lack of a coding activity. Since everything must be modeled first, all the coding-specific aspects should be represented in a model as well. This could easily lead to a situation of excessive modeling in which even the tiniest implementation detail must be covered before any actual coding can be done. Moreover as stated by Jack W. Reeves [Ree05]: 'all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents'. The reason for this is that the final design that corresponds to the code is the only one that was refined during the build and test cycle. Proponents will counter this by saying that modeling the implementation details leads to an improved documentation of programming basics. In their view this will ultimately lead to a set of reusable metamodels that cover basic development practices. Also with good program generators installed, models themselves can be refined iteratively since no manual coding step is necessary.

Note that both the model-centric and the code-centric vision are at the extreme end of the spectrum (i.e., only code or only models). Most development approaches however fall somewhere in between. For instance in agile software development a notion of 'just barely good enough models' exists (Agile Model Driven Development [AJ02]). As a consequence, even in a code-centric approach, evolving the code could result in a change to the model. Analogously, for code-centric approaches that resort to manual coding from time to time, evolving the models could result in a change to the hand-crafted code.

### 2.2.3 Synchronisation versus Co-Evolution

If the code or the models are evolved then a problem of *synchronisation* occurs. The synchronisation referred to boils down to transforming the deltas on one side to deltas on the other side. This is illustrated in figure 2.4.

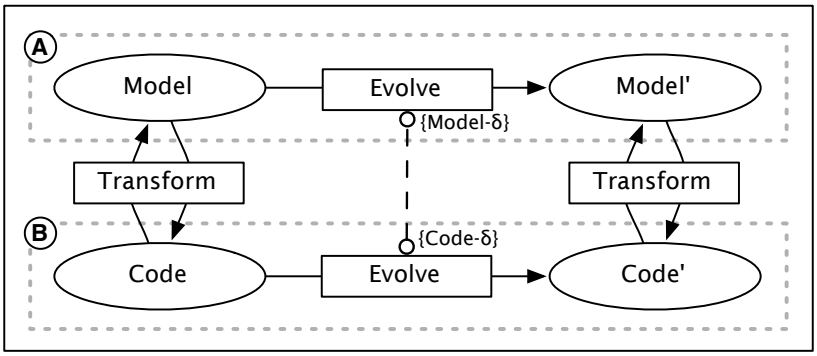



Figure 2.4: Evolution of models and code.

In a truly model-centric vision the evolution impact stays inside the upper half of the figure (A). For a truly code-centric vision it stays inside the bottom half (B). As already indicated, whenever a complementary situation occurs then a manual transformation step is needed between both sides. In such a situation, synchronisation is no longer a sufficient concept to describe the interplay between both sides.

*Co-evolution* is related to synchronisation but goes beyond keeping the model and code in conformance with each other. In biology, co-evolution refers to the influence the evolution of one organism has on the evolution of another organism. This influence can be either positive or negative for the other organism. It is clear that this doesn't simply translate into finding out how the evolutionary deltas of one organism can be mapped onto evolutionary deltas for the other. In software development this is similar since changes to the model do not necessarily result in a straightforward change in the code. The experience obtained by changing the model could result in a better understanding of how the code should be reorganised, which could result yet again in changes to the model. This (not always deterministic) interplay between both sides is what co-evolution is all about[4].

[4]This definition of co-evolution is broader than the one introduced in previous PROG research efforts (e.g., [DDVMW00, Men00, Wuy01]). In general the meaning in that work was targeted at the *synchronisation* aspect between models and code.

| | |
|---|---|
| *Code-centric development* | A software development approach where the code is seen as the primary artefact. |
| *Model-centric development* | A software development approach where the models are seen as the primary artefact. |
| *Synchronisation* | Synchronisation of code and models refer to the activity of keeping both sides in conformance with each other. |
| *Co-evolution* | Co-evolution of code and models goes beyond synchronisation by referring to the positive or negative influence that the evolution of one side has on the evolution of the other. |

Table 2.2: Definitions for section 2.2.

## 2.3 The Key Role of Domain Knowledge

A large amount of time in software evolution and maintenance is spent on *program comprehension*. Program comprehension can be broadly defined as the process of acquiring knowledge about a computer program. In the Software Engineering Body of Knowledge (SWEBOK, [ABDT04]) it is estimated that about 40 to 60 percent of maintenance is actually spent on comprehension. Canfora et al. even mention estimates as high as 90 percent [CC05]. So before any adaptation actually takes place, a lot of effort is put in trying to understand the software. This mostly boils down to building up a set of knowledge about the system by discovering or rediscovering what was known by the original developers.

### 2.3.1 Implicit Domain Knowledge

A lot of knowledge assembled during the software engineering lifecycle is documented in the various artefacts (code and models) that are created. Yet as stated by Rus et al. this is just a fraction of all the knowledge related to software that is being captured, since the majority of knowledge remains *implicit* in the brains of the employees [RLS01]. The fact that this intellectual capital is not available in an explicit form is not just unfortunate, but problematical for software evolution purposes, which we discuss later.

Many factors cause knowledge about the software to become or remain implicit. One factor is the *high personnel turnover* that characterises the business of software development. Software developers and domain experts are generally considered a volatile asset. In practice they switch companies or project teams on a regular basis. Moreover, as a result of the *complexity* of contemporary software systems, their memory about the peculiarities of these will inevitably deteriorate over time. Also, depending on what lifecycle-stage the software is in (figure 2.2), it will become more or less likely to find people that are well acquainted with the

system.

Another factor has to do with the *short-term economics* of software development. Making knowledge explicit in the form of documentation costs time which developers would rather spend on getting the code right for the next release. Unfortunately, people do not yet recognise the impact that this will have on a long-term basis [Par94]. In order to improve the situation one could only suggest the use of an 'intelligent' documentation mechanism that is not experienced as an overhead by the developers. [DH98]

The reason why certain knowledge remains implicit is also related in part to the *kind of knowledge* that remains implicit. Even in a code-centric vision, where code is seen as the prime medium to capture the design, there is a need to capture important information outside the code. This has to do with the suitability of code as a representation for this knowledge. An analogous observation applies to a development environment that follows a model-centric vision.

Consider for example the information from the problem space that did not make it into the design, such as the motivation why the developers opted for a particular architecture [Ree05, PD90]. It is most likely that this knowledge is captured in neither the code nor the models.

Also the move towards more *malleable software systems* does not improve the situation. As we already explained in the previous section, the required genericity in such systems results in less explicit code. Less explicit code means that a lot of knowledge becomes implicit in the data, which does not enhance the code's readability[5].

Note however that this need to capture information outside the code does not only apply to implementation knowledge. Also *common sense knowledge*, about the business domain in which the software operates, is often left implicit in the code or the models. For instance, it could well be that certain choices were made in the code since everybody 'knows' that the alternative is not considered an option from a business perspective.

### 2.3.2 Communicating Domain Knowledge

The problem of not having access to domain knowledge in an explicit form can be illustrated by the simple communication chain shown in figure 2.5 (adapted from [Kri98]). The percentages in the figure indicate the quality of the communication. Suppose that the requirements put forward by the customer correspond to 90 percent of what is actually needed. Then even when information is transmitted with a high quality of 90 percent (from customer to analyst, from analyst to designer, from designer to programmer, from programmer to the implementation), at the end of the chain we end up with a quality total less than 60 percent.

[5]This is mainly due to the fact that, in contrast to the structure of the data, the actual data is not documented.

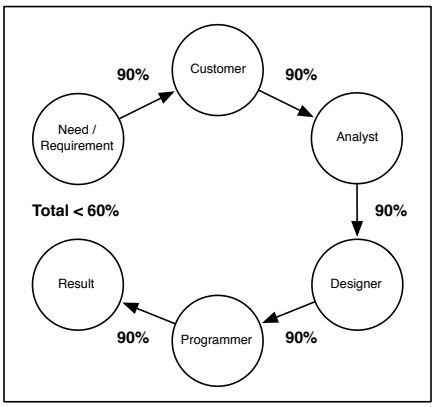



Figure 2.5: Reduced quality as a result of communication problems.

What happens is a progressive deterioration of the quality each time the information is being transmitted between people or process into an artefact. This is similar to the way domain knowledge is passed on between successive evolution steps (in that case the circles in the figure would represent developers).

This loss in quality when information is passed on is one of the main reasons why developers need to invest so much time in program comprehension before any other active like maintenance can take place. To make matters worse, there is also a loss in quality when the actual request for evolution is communicated. This is illustrated in figure 2.6. This shows a business expert who communicates a need for evolution in the business domain (first loss of quality). The need is communicated in terms of the business knowledge of the expert. Consequently the software expert needs to translate the request by using his implementation knowledge (second loss of quality). After this, the software expert can map the request onto the parts of the implementation that need to be adapted (third loss of quality).

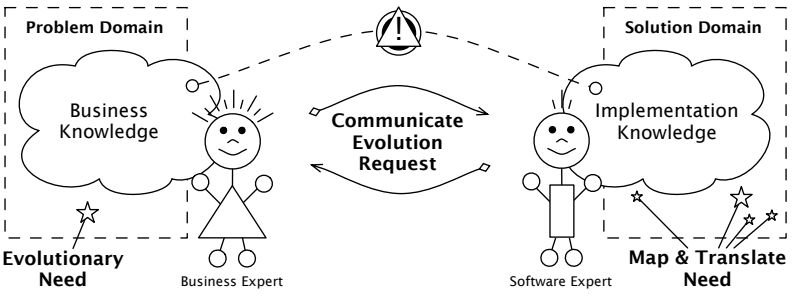



Figure 2.6: Mapping an evolutionary need onto an implementation.

| | |
|---|---|
| 1 | Domain concept descriptions |
| 2 | Location and uses of identifiers |
| 3 | Connected domain-program-situation model knowledge |
| 4 | List of browsed locations |
| 5 | List of routines that call a specific routine |
| 6 | A general classification of routines and functions such that if one is understood, the rest in the group are understood |
| 7 | Format of data structure plus description of what a field is used for in program and application domain, expected field values and definitions |
| 8 | History of past modifications |
| 9 | Bug behavior isolated |
| 10 | List of executed statements and procedure calls, variable values |

Table 2.3: Top information needs of software maintainers [KSP04].

### 2.3.3 Top Information Needs for Evolution

Koskinen et al. [KSP04] observed that *domain concept descriptions* and *connected domain-program-situation knowledge* are among the top three most frequent information types needed by software maintainers. Moreover the need for domain concept descriptions occurred the most in corrective and adaptive maintenance.

In table 2.3 we summarise the ten most frequent information needs from this survey. Note that this table also includes information that is not in general considered as domain knowledge (e.g., 4 and 5). The observation was made after analysing the empirical data of several field studies by von Mayrhauser et al. [vMV95a, vMV97, vMVH97, vMV98]. This consisted of observational studies of professional software maintainers that worked on C-code bodies of up to 40000 lines of code. The changes involved covered all the different categories of maintenance we described in the previous section.

The field study also observed that software maintainers followed a multilevel approach in which they switched continuously between program, situation, and domain models. This indicates that the connections between the different models are also important.

The observations above are confirmed by Rajlich et al. [RB00] who state that '*staff must understand the domain, solutions to domain problems, program properties, including software architecture, concept location within the code, and basic computer science principles and coding conventions*'. Also here the importance of domain knowledge is stressed together with more code specific information (i.e., obtained through static or dynamic code analysis). Moreover similar to Koskinen et al. [KSP04] they refer to the *concept location within the code*. This means that developers do not only need to have access to the domain knowledge but that they also want to know *where* it is implemented in the code.

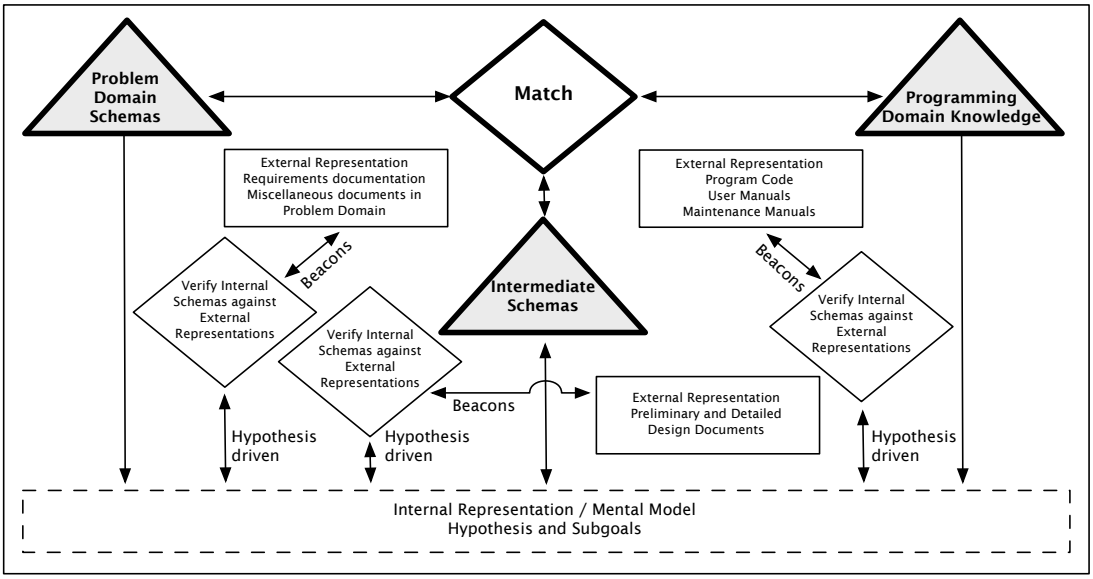



Figure 2.7: The Brooks code comprehension model.

In figure 2.7 we illustrate the Brooks comprehension model [vMV95b]. This model shows the way in which a developer reconstructs the domain knowledge that was used by the original developer. As shown, the comprehension process consists of a continuous mapping of elements from the problem domain into the programming domain.

The triangles in the figure refer to the different kinds of knowledge involved in the comprehension process. Problem domain knowledge contains real world knowledge such as the rules that govern the insurance domain. Programming domain knowledge refers to knowledge of general programming concepts or programming tools and environments (e.g., coding conventions, design patterns). The intermediate domain knowledge refers to knowledge that describes the relationships between related domains.

The verification of problem, programming, and intermediate domain knowledge with what the code actually does is core to the model. This verification consists of coupling the knowledge back to the implementation. In order to do so a developer will have to couple the often very broad domain concepts (e.g., inventory management) to very narrow distributed code entities. We will explain this in more detail in chapter 3, section 3.1.1.

## 2.4 Summary

In this chapter we pinpointed the differences in meaning between software maintenance and software evolution. In essence software maintenance is restricted to

changes to the software that are in line with the original requirements. Software evolution on the other hand is mostly about *continued development*, which is the focus of this dissertation.

We introduced Lehman's characterisation of software systems into S-type, P-type, and E-type systems. Broadly speaking the first category was the most static one, and the last category the most dynamic. This characterisation is important since in this work we mainly target *E-type systems* where there is *continuous need for adaptation*.

We further argued that this continuous need for adaptation was related to the age of the software. Therefore we discussed software aging and the staged model for the software lifecycle of Rajlich and Bennett. This allowed us to position the evolutionary stage in which a software system finds itself.

We subsequently discussed the notion of *malleable software*. This refers to systems where software evolution is anticipated by the way in which the code is implemented. The broad vision of malleability was to introduce a notion of genericity in the implementation in order to keep the software 'soft'. As we have seen, malleability results in code that can accommodate changes either at development-time or at run-time.

We have also positioned software evolution within contemporary and traditional software development approaches. Two opposite visions of development were characterised, namely the code-centric and model-centric approaches. Most approaches however use a hybrid approach. As a result a synchronisation problem arises between code and models during evolution. We discussed this and contrasted synchronisation with *co-evolution*.

One of the top information needs in software evolution is *domain knowledge*. Yet as we have discussed, in practice a lot of this knowledge becomes *implicit* after initial development. Besides personnel turnover, one of the main reasons was that documentation is usually perceived as an overhead by the developers. We have also shown that the *connection* to the code level is one of the top information needs for developers.

We dedicate the next chapter to a discussion of domain knowledge since it has a central role in this dissertation. We will also introduce ontologies as a medium to make domain knowledge explict.

# Chapter 3

# Domain knowledge and ontologies

In this chapter we define the basic concepts surrounding domain knowledge and ontologies, and how both can be teamed up to form the backbone of this research.

As described in chapter 2, domain knowledge is an essential ingredient for program understanding in the context of software evolution. This is why we start this chapter by defining the term 'domain' in section 3.1.1 and its associated knowledge in section 3.1.2. We also position domain analysis, which is performed to obtain a set of domain knowledge in section 3.1.3.

Having access to an explicit set of domain knowledge is one of the main points of this dissertation. This our entry point into the topic of ontologies, which is described in section 3.2. In order to better appreciate our view on ontologies it is necessary to present a number of existing definitions (section 3.2.1) and the different flavours of ontology (section 3.2.2). Also we present a number of practical choices that need to be made when dealing with ontologies in section 3.2.3.

## 3.1 Domain knowledge

In chapter 2 we stated that programmers need to have access to a rich set of domain knowledge to be able to understand what the system is actually doing. This implies that programmers need to become literate in the domain under consideration. Therefore we elaborate on what we mean when we refer to a domain in section 3.1.1. We distinguish between business domains and application domains, as well as between vertical and horizontal domains, and encapsulated versus distributed domains. These distinctions are necessary to be able to position our research in later chapters. Furthermore we touch the subjects of domain composition and domain coverage. This is needed to fully grasp the different levels of granularity one needs to take into account when providing a way to make domain knowledge explicit.

We also elaborate on what we consider to be domain knowledge in section 3.1.2. Next we illustrate the difference between explicit and tacit knowledge and how you can transform one into the other. This is a good starting point into the topic of domain analysis and domain engineering. Since the domain knowledge is actually discovered and transcribed during these activities, we dedicate the main body of section 3.1.3 to this topic.

### 3.1.1 Defining the Term Domain

#### Domain, Business Domain, Application Domain

A first important step in every software development project is dedicated to getting acquainted with the *domain*. The term domain can be interpreted in a multitude of ways. In its most basic meaning, a domain refers to "a sphere of knowledge, influence, or activity" [MW05]. In software engineering it is typically associated with two broad interpretations: a business domain and an application domain.

The first refers to a domain as a subset of knowledge about some area in the real world confined to a particular business area. So in that case a domain could delineate for instance a sphere of knowledge about banking, software development or insurance. Example knowledge items in the insurance domain could be `insurance policy`, `claim`, and `policy holder`. You would have to know the meaning of these items to be able to work or to develop software within the insurance domain. Often it includes knowledge about organisational and economic issues as well. For example, you would have to know the different ways in which a damage claim should be processed according to specific regulations. This interpretation of a domain is labelled as a *business domain* [WPD92].

A second interpretation for domain mainly stems from research in software reuse. In that context the term domain is used to refer to a sphere of knowledge that is defined by a set or "family" of software applications. Suppose a domain defined by interactive media (iMedia) quiz applications. Typically there are several applications that implement this domain. In this interpretation of a domain, the domain does not only encapsulate real world knowledge about the business domain of quizzes. Among others, it also contains knowledge about software development practices in that field. For example in iMedia software development it is common to make the necessary provisions for multi-platform (e.g. PDA, TV) and multi-channel (e.g. podcast, kids channel) deployment. This type of domain is referred to as an *application domain* [WPD92].

#### Domain Boundary

As already indicated, the word 'sphere' in the basic definition expresses that a domain has a boundary. This boundary is used to restrict or scope the elements

which one wants to talk about. So when we speak of "domain" knowledge, we refer to a subset of knowledge restricted to a particular area of expertise. To determine what lies inside, outside or on the boundary of the domain is one of the most important and difficult decisions to make. Sometimes it is not clear whether or not an element should be either inside or outside the domain boundary. In certain cases it is even impossible or not desirable to define a strict boundary. In general, the existence of 'grey' spots on the boundary region are almost unavoidable for real world domains. This is clear if we consider the less general definition of a domain by Prieto-Diaz and Arango [PDA91] They state that a domain can be characterised by its vocabulary, common assumptions, architectural approach, and literature, once it is recognised. Especially the "common assumptions" and "literature" in this definition clearly opens the door for a less rigid interpretation of the word boundary.

In the context of this work, we assume the domain sphere to be bound by the area for which we are building a software system. Note that this is not the same as saying that the domain elements are defined by the existing software applications. If we start from a green-field situation (a.k.a. clean slate) we can state that the business domain is constrained by the 'future' application domain. A green-field situation means that we build a system from scratch. If we do not start from a green-field situation (a.k.a. ploughed-field), we combine certain aspects from the existing application domain with the business domain into a new version of the application domain. Such a ploughed-field situation implies that we need to take care of existing applications when building the new software.

### Domain-composition, Domain-coverage

Now that we have identified a meaning for the word domain, we should elaborate on the aspects of domain-coverage and domain-composition. Depending on the size of the domain, it is well possible to have a network of several more fine-grained (sub-) domains working together. This network is called the *domain-composition*. Note that a software application is not necessarily confined to a single business domain. It is quite common to have an application domain that is covered by the interaction of several other domains. Consider again the software responsible for generating interactive quizzes. Its application domain consists of the interaction of at e.g. the following domains: quizzes, interactive media, domain-specific languages, transformation systems and graphical user interfaces. Note that some of these domains can stand on their own, which is why we speak of a network of domains instead of a hierarchical decomposition of one parent-domain.

Obtaining a complete *domain-coverage* is often only possible (and cost-effective) for very narrow and well-defined domains. A typical area in which such domains occur are mathematics. In more business-oriented domains and for practical purposes however, a partial domain-coverage is perfectly acceptable. This is supported by the insight provided in the work of Wagner where it is stated that (do-

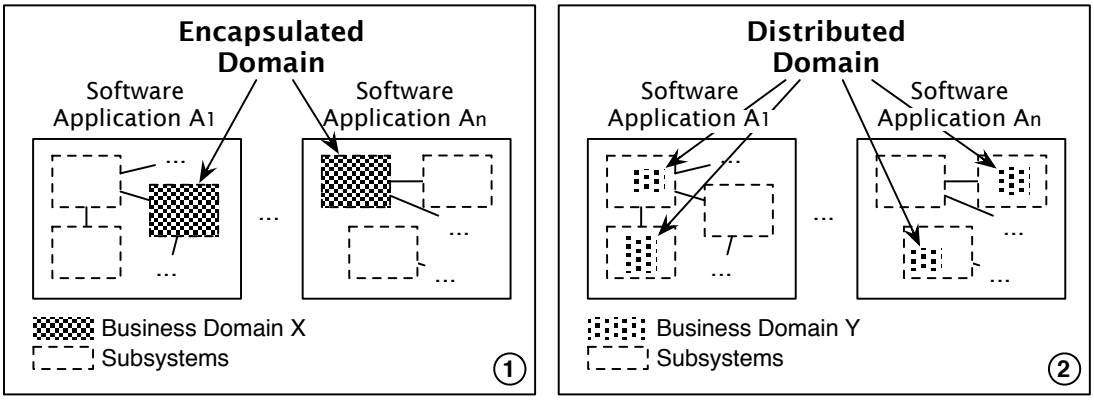



Figure 3.1: Encapsulated and distributed domains.

main) expertise should be conceived as a substantial collection of semi-complete, semi-structured and subjective know-how [Wag90]. Nevertheless, a partial coverage is not to be confused with an incomplete or faulty coverage. It is all about the granularity needed, which boils down to deciding which domain elements are to be captured and which are not.

### Encapsulated Domain, Distributed Domain

Sometimes there is a one-to-one mapping between a subsystem in a software application and the domain. This means that all the functionality that can be associated with the domain can be lifted out of the application in a clean and relatively easy way. In that case the domain is said to be encapsulated (figure 3.1 (1)). Such *encapsulated domains* are prime candidates for creating reusable software components. It is more common, however, to have a situation in which the domain functionality 'crosscuts' several subsystems of the application [CE00]. In that case it is difficult, if not impossible to lift out the corresponding functionality. This situation is named a diffused or *distributed domain*(figure 3.1 (2)).

### Vertical Domain, Horizontal Domain

From an application perspective it is common to divide domains into *vertical* and *horizontal domains* [Sim96, CE00]. This is illustrated in figure 3.2. A software application that is mainly created for use within one particular (business) domain is said to reside within a vertical domain. From a software vendor perspective this is similar to targeting a vertical market. Consider for example the set of applications that handle credit and loan applications for financial institutions. The software vendor could in that case opt for creating a domain specific architecture for this application domain. This architecture should capture all the commonalities of applications in the vertical domain. He consequently instantiates the

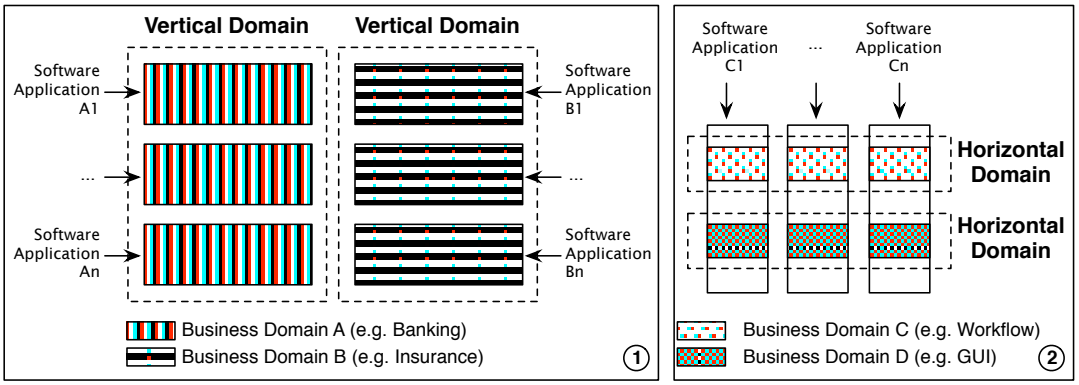



Figure 3.2: Vertical domains and horizontal domains.

baseline software infrastructure for handling the specifics (a.k.a. variabilities) of each customer.

If a software part is usable across several (vertical) domains, it is said to reside inside a horizontal domain. Examples of these are workflow management and advanced graphical user interface systems. Such applications or subsystems are usually specifically targeted for use within other applications. A workflow management toolkit makes sense when it is used within the context of processing insurance claims, for instance. Using this toolkit in isolation or outside a particular context does not make sense.

#### Summary

In this dissertation we define a domain to be the sphere of knowledge about a field for which we are developing a software system. Most often when we refer to a domain we actually refer to a business domain. In the case where certain abstractions (or conceptualisations) are used that are not part of the 'real world' we refer to an application domain. Since the main focus is on a business domain, we will be quite often confronted with partial domain coverage. With respect to the implementation, these domains themselves are best characterised as distributed domains. In table 3.1 we summarise the main definitions that were introduced in this section.

### 3.1.2 About the Knowledge Associated with a Domain

#### Domain Knowledge

Whereas a domain refers to the area about which we want to talk, the term *domain knowledge* refers to the knowledge items associated with that particular area. We consider 'knowledge' according to the common distinction made in

| | |
|---|---|
| *Domain* | A sphere of knowledge about a field for which software systems are developed. |
| *Business domain* | A domain that is constrained by a particular business area in the real world. |
| *Application domain* | A domain that is constrained by the functionality provided by a particular set of software applications. |
| *Domain composition* | Refers to the way a domain consists of several more fine grained sub-domains. |
| *Domain coverage* | Refers to how well a domain is embraced by the constituting elements of a domain description. |
| *Vertical domain* | A domain that refers to a set of applications that reside within one business domain. |
| *Horizontal domain* | A domain that refers to an application domain that is usable across several vertical domains. |
| *Encapsulated domain* | A domain for which the corresponding functionality is completely captured by a subsystem of an application. |
| *Distributed domain* | A domain for which the corresponding functionality is scattered across several subsystems of an application. |

Table 3.1: Definitions for section 3.1.1

| Name | Type | Definition | Technical |
|---|---|---|---|
| Profession | Entity Type | The principal activity in your life for which you are paid. the employment a person has with one employer. Aka occupation, employment, job, activity. | (PFN) |
| ProfessionCategoryCode | Attribute of Profession | Coded representation of the category the profession belongs to, e.g. professional sportsman, teacher, sailor, etc. | Code (ATT+CQ15) |
| ProfessionExecutionLocationText | Attribute of Profession | Free format description of the place where a profession usually is executed | Text (FTX+037) |
| PostalCode | Attribute of Address | A group of letters and/or numbers added to an address to assist the sorting of the mail | Code |
| PartyRole | Entity Type | The role a party plays in insurance, e.g. policyholder, driver, insurance provider, etc. | |

Figure 3.3: Extract of the vocabulary defined in the Telebib2 reference dictionary.

literature between data, information, and knowledge. In general terms data is considered to be a set of raw facts from which certain conclusions may be drawn. Information on the other hand adds a meaningful interpretation to this data. For our purpose we make use of the working definition of knowledge from Davenport and Prusak [DP00]:

> "Knowledge is a fluid mix of framed experience, values, contextual information and expert insight that provides a framework for evaluating and incorporating new experiences and information."

As described by Nonaka et al. knowledge, unlike information, is about beliefs and commitment [NT95]. Hence it is always in function of someones particular view or perspective. Moreover, knowledge should always serve some purpose or action. For instance the reason why a person possesses the knowledge of writing is to be able to write. The knowledge itself without the associated purpose of writing is useless. It is important to keep the purpose of knowledge in mind since it allows us to distinguish between what should be considered as application knowledge or as business knowledge in a particular context.

The knowledge items of which we speak can range from very broad and abstract concepts (e.g. `insurance policy management`) to very narrow and concrete concepts (e.g. `postal code`). Obtaining a sufficient body of domain knowledge for developing a piece of software is mainly done by consulting domain experts and analysing whatever material is available. In part, this means getting used to the common jargon (a.k.a. domain vocabulary) in which experts describe the concepts and rules that govern that particular domain. The resulting set of knowledge is consequently synthesised into a set of artefacts referred to as a *domain model*. The overall phase concerned with obtaining a good domain model is domain analysis, which will be discussed in section 3.1.3.

An example of a large body of (standardised) domain knowledge is represented in the Telebib2 platform [Pla05]. The goal of this platform is to provide the Belgian insurance sector with a set of standards and rules for electronic data

exchange and procedures. It contains an extensive data dictionary, data and process models, structures for data exchange as well as a set of maintenance procedures. An extract of domain concepts that exist in the vocabulary is given in figure 3.3.

### Explicit Knowledge, Tacit Knowledge

A widely acknowledged problem is that a lot of knowledge is not available in an explicit form. *Explicit knowledge* (a.k.a. tangible knowledge) is knowledge available in a codified way, such as documents, databases and requirements diagrams. This codification can be either formal or informal, according to the particular needs. An important property of explicit knowledge is that it can be easily communicated between individuals.

*Tacit knowledge* (a.k.a. intangible knowledge) in contrast is uncodified knowledge that is often based on personal experience, know-how, values and emotions. It is often referred to as the knowledge that resides in the heads of people (as opposed to residing in a tangible form such as documents) [OG98]. Note that "tacit" in this sense is not synonymous with "implicit".

Domain knowledge encompasses both explicit and tacit knowledge. For software development it is often the tacit kind that is crucial to ensure project success or failure. Even though certain types of tacit knowledge cannot be made explicit, a lot of it can be codified and put to good use by sharing it [DP00, NT95]. Consider for example coding conventions and best-practices in a software development company. Often this knowledge is 'implicitly known' by every programmer in that particular company. Yet for enforcing or encouraging the consistent use of these practices (or for sharing them with new employees), this tacit knowledge should be converted into explicit knowledge. This transformation process where tacit knowledge is converted into explicit knowledge encompasses more than merely writing it down, as we detail next.

### Knowledge Conversion

In figure 3.4 we show the knowledge spiral from Nonaka and Takeuchi [NT95]. As is shown there are four modes of knowledge conversion: socialisation, externalisation, combination, and internalisation. Each mode corresponds to the conversion of tacit or explicit knowledge to either tacit or explicit knowledge:

- From tacit to tacit: Socialisation
- From tacit to explicit: Externalisation
- From explicit to explicit: Combination
- From explicit to tacit: Internalisation

These conversion modes are important since they allows us to pinpoint the kinds of knowledge as well as the kind of 'explicitation' we refer to in our approach.

*Socialisation* (from tacit to tacit) mainly results in sympathised knowledge[1] that is the result of sharing experiences between individuals. Practice with experts, and observation or imitation of experts are good ways to share such experience. Field building is usually the trigger for socialisation since it provides a context for interaction. An example from extreme programming that can be categorised in this mode is pair programming. *Externalisation* (from tacit to explicit) mainly creates conceptual knowledge. The main way to externalise knowledge is by using metaphors, analogies and models. In general, dialogue and collective reflection are triggers for externalisation. Performing a green-field domain analysis (see section 3.1.3), where you need to establish a model of the domain "from scratch", could be categorised as externalisation. *Combination* (from explicit to explicit) mainly results in systemic knowledge[2] by combining existing explicit knowledge into new explicit knowledge. This can be done by reconfiguring the existing knowledge through categorisation, classification, and composition among others. The trigger for the combination mode is the networking (linking) of newly created and existing explicit knowledge items. For instance refactoring and restructuring existing software models (e.g. class diagrams) could fall under this category. *Internalisation* (from explicit to tacit) mainly results in operational knowledge by putting explicit knowledge to practice and hence gaining know-how. Studying and analysing documents, manuals, customer interaction transcripts help to transfer such experiences between individuals. Learning by doing is the best way to describe the trigger that activates this mode. A good example in which explicitly documented experiences are transferred and converted into tacit knowledge, are design patterns [GHJV95] and best-practice patterns [Bec97]. In this work we are mainly interested in the externalisation and the combination mode.

### Knowledge Acquisition

Obtaining a comprehensive body of domain knowledge is done in part by applying several *knowledge acquisition* techniques. Such techniques are mainly targeted at the elicitation, the analysis, and the organisation of knowledge coming from different sources. Some of the most common approaches to identify the core concepts are interviews, observations, and document analyses. The use of knowledge acquisition techniques should be governed by an overall knowledge management initiative. This especially applies in the context of software engineering, a knowledge-intensive activity where the organisation's intellectual capital is one of its main assets, the existence of a knowledge management policy is crucial [RLS01, RL02]. It lies outside the scope of this work to give a detailed

[1] Sympathised in the sense of an informal understanding or common feeling between people.
[2] Systemic in the sense of relating to a whole system.

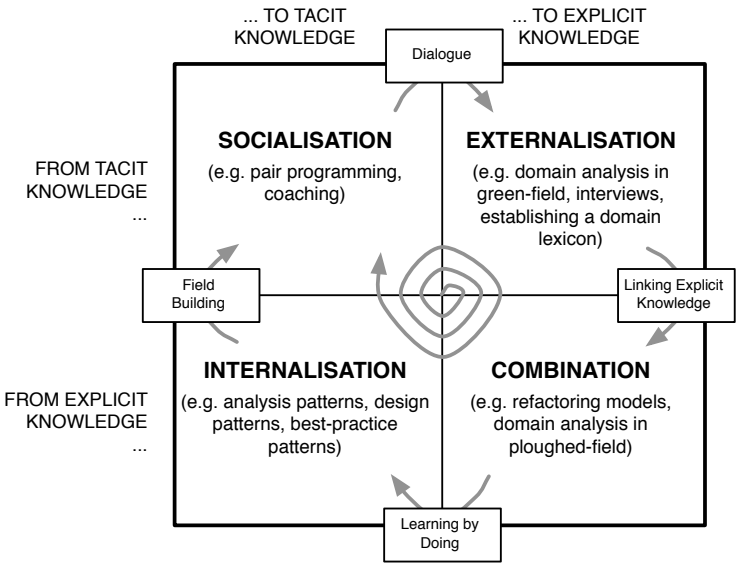



Figure 3.4: The four modes of knowledge conversion in the knowledge spiral: socialisation, internalisation, externalisation, and combination (adapted from [NT95]).

description of the different knowledge acquisition techniques. A good place to start, in the context of software development, is the CommonKADS methodology [SAA+00]. This methodology specifically focuses on engineering knowledge-intensive software systems. It also provides a rich library of templates that can be used to describe different knowledge elements.

The initial elements that constitute a model of the domain depend on the approach that is taken. When a problem-oriented approach is taken it initially consists of problem-level concepts. These are subsequently refined into concepts that can be used to describe the solution. Conversely a solution-oriented approach starts by examining a number of known solutions (i.e. software applications) to a problem. Starting with a large body of solution-level concepts that stem from the software implementation, you identify the core concepts in the problem domain. Depending on the objectives and the number of existing software applications in the domain, you either opt for one of the former approaches or a combination [WPD92]. Note however that starting from the solution domain will probably identify certain abstractions that do not necessarily exist in the problem domain. Consequently this might pollute the model with elements from the application domain whereas the intended target was the business domain.

#### Kinds of Domain Knowledge

In CommonKADS [SAA+00], knowledge is classified as either domain knowledge, inference knowledge, or task knowledge. In CommonKADS, domain knowledge is defined as the main static information and knowledge objects in an domain (e.g. `quiz master`, `round`, `question`). In the inference knowledge they describe how these static structures can be used to carry out a reasoning process (e.g. `validate`, `schedule`). Task knowledge describes the reason or context for which the system is developed (e.g. `organise quiz`).

It is not always necessary to treat these three kinds of knowledge as separate categories. All types could be labelled as domain knowledge if a broad view is taken of what the contents of a domain could be. In such a view, the domain knowledge could be focused on the domain concept taxonomy, the domain concept composition, the domain tasks, the domain activities, or a combination of these. Nevertheless we find these categories of knowledge as sub-domains in the model of the domain. For organisational issues of setting up a knowledge policy, strategies for improving knowledge transfer, and the establishment of a knowledge culture, we refer to the work of Nonaka and Takeuchi [NT95] as well as the work of Davenport and Prusak [DP00].

In the context of software engineering we can distinguish two broad categories of knowledge. The first is knowledge about the business domain for which you are building software. The second kind of knowledge can be seen as meta-knowledge. In essence it is knowledge about the processes and products for developing the software. Depending on the business domain in which you are working, certain knowledge items can shift category. For a tool vendor for instance, the knowledge about the development processes are part of the business domain and hence fall under the first category of knowledge. For a programmer this kind of knowledge fits the second category. This shift is why it is difficult to categorise knowledge items without considering the context in which they are used.

#### Summary

If we refer to domain knowledge, we always specify whether it is rooted in the problem domain (e.g. banking) or in the solution domain (e.g. software development practices), unless both are applicable. In table 3.2 we summarise the main definitions that were introduced in this section.

### 3.1.3 Positioning Domain Analysis

The process responsible for creating a domain model is referred to as *domain analysis*. In the context of software reuse, Neighbors [Nei80] is generally accredited for being the first to define domain analysis as the activity in which objects and operations of a class of similar systems in a particular problem domain are

| | |
|---|---|
| *Knowledge* | Knowledge is a fluid mix of framed experience, values, contextual information and expert insight that provides a framework for evaluating and incorporating new experiences and information. |
| *Domain knowledge* | A set of knowledge that captures the concepts and rules that govern a particular domain. |
| *Tacit knowledge* | Uncodified knowledge that is often based on personal experience, know-how, values and emotions. |
| *Explicit knowledge* | Knowledge available in a codified way such as documents, databases and requirement diagrams. |
| *Externalisation* | The mode of knowledge conversion that mainly creates conceptual knowledge, going from tacit to explicit knowledge. |
| *Internalisation* | The mode of knowledge conversion that mainly results in operation knowledge going from explicit to tacit knowledge. |
| *Socialisation* | The mode of knowledge conversion that mainly results in sympathised knowledge going from tacit to tacit knowledge. |
| *Combination* | The mode of knowledge conversion that mainly results in systemic knowledge going from explicit to explicit knowledge. |

Table 3.2: Definitions for section 3.1.2

being defined. In this definition Neighbors uses the term domain in the sense of an application domain. As we have already indicated, a major part of domain analysis consists of the use of knowledge acquisition techniques to find the domain's objects and operations. Depending on the development method that is used, the domain analysis is included in a separate *domain engineering* lifecycle or is integrated in the *application engineering* lifecycle.

### Domain Engineering, Application Engineering

We follow the distinction between both cycles as defined by Reifer [Rei97]. He defines the application engineering cycle as the traditional processes used by software engineers to develop and maintain software products. In his view the domain engineering cycle is targeted at defining a common understanding of the domain in which a set of reusable assets is created for a class of software systems. In this context, application engineering can be called "engineering with reuse" since software systems are created by making use of already available reusable components. On the other hand, domain engineering can be called "engineering for reuse" [Har02], because the prime goal is to create a set of highly reusable components. An example of a dual lifecycle where application and domain engineering are combined is presented in figure 3.5. As is shown, both cycles are similarly phased. Such a dual lifecycle is common in software engineering methods that are strongly focused on creating reusable assets. A particular example of such a methodology is *software product-line engineering* [WL99]. Remember that in such a method the definition of a domain refers to a family of software systems. The main purpose of domain analysis for this kind of engineering is to identify the commonalities and variabilities for a given application domain. For our purpose it is of interest that a considerable amount of time and effort is spent in analysing and synthesising the domain knowledge.

In part, the activities of the domain analysis phase are similar to the ones of the requirements analysis phase. However here a different viewpoint is taken. Domain analysis in a domain engineering context is generally oriented towards obtaining a reusable specification of the (entire) *application* domain. Hence, while performing it, you try to look beyond the boundaries of a single application. This means that you try to capture the commonalities and variabilities of an entire domain, and not just those necessary for one specific problem / solution space, as is the case in requirements engineering. This means that domain analysis is best suited for mature domains, where the basic concepts underpinning the domain are stable and clearly understood.

Even though domain analysis has a strong bias to promote software reuse, it is also used for capturing domain knowledge without this bias [SM89]. One of the primary goals of domain analysis, as stated by Shlaer and Mellor, is to transfer the domain knowledge accurately to the software designers and programmers. This is in part done by providing a description of the objects (as abstractions of a set

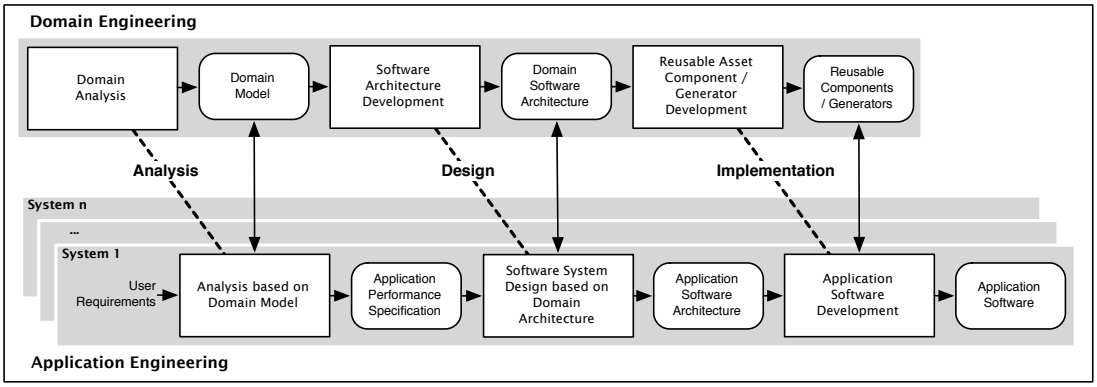



Figure 3.5: A dual lifecycle in which domain engineering and application engineering closely interact (adapted from [SEI05]).

of real-world things) and the relationships that hold between them. So domain analysis contributes to the process of converting tacit into explicit knowledge (i.e. externalisation). The result of the domain analysis phase is a reusable domain model that can be taken as input for the application analysis phase.

During the other phases shown in figure 3.5 (i.e. the domain design and implementation phase), a reusable infrastructure and a set of reusable components are specified and built [SEI05]. These are instantiated and used during the design and implementation phase of the application engineering cycle.

### Vertical Domain Analysis, Horizontal Domain Analysis

Analogous to vertical and horizontal domains a distinction is also made between vertical and horizontal domain analysis [Boo87]. When performing a *vertical domain analysis*, you study a number of software systems that are intended for the same class of applications (i.e. within a vertical domain). Conversely, performing a *horizontal domain analysis* focuses on the study of a number of different systems across a variety of applications (i.e. within a horizontal domain).

### Domain Model, Domain Lexicon

As already mentioned, one of the main assets produced by a domain engineering cycle is a domain model. The purpose of a domain model is to provide an explicit reflection of the knowledge acquired about a business domain. In section 3.1.2 we have already mentioned that this knowledge could either be situated in the problem domain (e.g. as a result of requirements analysis) or in the solution domain (e.g. as a result of the design of the software solution).

Depending on the domain or the application, certain representations for this knowledge are more suitable than others. In the case of fairly data-centric soft-

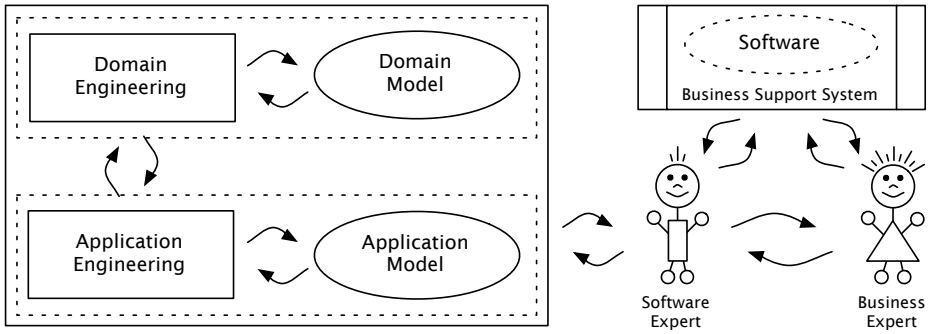



Figure 3.6: The domain engineering cycle results in domain models, whereas the application engineering cycle results in application models. Next to these models a running instantiation of these models exists with which the experts interact.

ware applications it probably captures certain aspects in the form of an Object Role Modelling (ORM [Hal01]) or an Entity Relationship (ER [Che76]) diagram. The Unified Modelling Language (UML) provides several representations, each of which captures different kinds of knowledge about the system to be built. In essence you need a way to represent domain concepts and domain rules. Often a UML class diagram is used to model the domain concepts and relate them to each other. A UML statechart diagram can be used to model the specific characteristics of a process-oriented domain concept.

Additionally a domain model should include a *domain lexicon* in which the terminology used is unambiguously defined. The main purpose of this lexicon is to improve the communication between the developers and the stake-holders. It should also be used as a point of reference for all the terminology used in the different diagrams describing particular aspects.

Figure 3.6 presents an abstraction of the way the domain engineering and application engineering interact with each other and their corresponding models. We already discussed domain models in this section. As application models we take a very broad view, namely that these range from (pseudo-)models that are close to the domain (and often of an informal nature) to models that are close to the actual implementation. Examples of the former are use cases and scenarios. In our view, examples of the latter can range from class diagrams specifying the design of the system to the actual source code. Note that in traditional development these application models are decoupled from the running software application.

In the following section we will discuss ontologies as a medium to capture domain knowledge in an explicit form.

| | |
|---|---|
| *Application engineering* | The processes/practices firms use to guide the disciplined development, test, and life-cycle support of their applications software. |
| *Domain engineering* | The reuse-based processes/practices firms use to define the scope, specify the structure, and build reusable assets for a class of systems or applications. |
| *Domain analysis* | The analysis of the objects and rules that govern a particular domain. |
| *Vertical domain analysis* | The domain analysis of a number of systems within the same vertical domain. |
| *Horizontal domain analysis* | The domain analysis of a number of systems across different domains. |
| *Domain model* | An explicit specification of domain knowledge. |
| *Domain lexicon* | Defines the terms concerning the domain to improve communication between developers and stake-holders. |

Table 3.3: Definitions for section 3.1.3.

## 3.2 Ontologies

### 3.2.1 About Ontology and ontologies

Recent years have shown an increased interest to put ontologies to work in the area of computer science. In the beginning there was a continuing debate about how you should interpret ontology within this context. Fortunately, the community has settled on using the definition proposed by Thomas Gruber [Gru93], namely an explicit shared representation of a conceptualisation that holds in a particular context. The reason why his definition is currently the most widely cited in the area is probably due to its rather abstract and generic nature. This means that according to your particular application context, you can easily instantiate it for your own purposes. Before we come back to the definition, we first present an overview on the different ways one can interpret the term ontology. This is necessary in order to better appreciate and understand it.

#### Ontology as a philosophical discipline

First of all we should observe a very basic distinction between Ontology and ontology [GG95, Gua98]. The former in which an uppercase 'O' is used, refers to the original meaning of the word. Ontology in this form denotes a philosophical discipline, more specific a branch of metaphysics that is concerned with the nature and the relations of beings. A 'being' in this case should be interpreted

as something that exists. Speaking in general terms, Ontology tries to formulate an answer to the question "What are the features common to all beings?". It is clear that defining these common features is extremely difficult and gives rise to a lot of lively discussions. The reason for discussion is of course that there does not exist one single way to look at the world. A classical example of Ontology are the ten categories by Aristotle: substance, quality, quantity, relation, spatial, temporal, situated, having, activity, and passivity. Aristotle used these categories for classifying anything that can be said about something [Sow99]. In essence he defined categories in terms of other categories by specifying their genus and differentia. The genus of a category refers to its set of defining or common properties, whereas the differentia refers to its distinguishing properties.

A conceptual analogy can be drawn to the class-based programming paradigm. In that case the genus corresponds to defining a new class by indicating its superclass (what it has in common with other classes). The differentia corresponds to extending the definition inherited from the superclass with its own set of state and behavior. An extensive resource guide for philosophical ontology is provided by Corazzon [Cor05].

#### ontology in computer science

In computer science the term ontology is written with a lowercase 'o' and as a countable noun. In this case it refers to an engineering artefact in the form of a repository. An ontology in this sense contains the definition of the relations and the nature of beings. According to Uschold, an ontology of this form should at least include a vocabulary of terms and a specification of their intended meaning [Usc98]. The specification of this meaning actually is the conceptualisation of the vocabulary. This includes definitions and how a particular concept relates to others as well. Specifying the relations between concepts results in a structured network of concepts that constrain the possible interpretations. This means that if you would translate the terms in an ontology from one language into another that this would not change the ontology conceptually [CJB99].

As described by Mayhew and Siebert, another primary distinction between Ontology and ontology has to do with the scope of the ontological investigation [MS04]. In philosophical Ontology they traditionally seek to provide a general ontology of the reality as a whole. Computer science 'ontologists' are mainly focused on developing domain-specific ontologies to meet particular application needs. Generally speaking the philosopher seeks knowledge of what exists more or less for the sake of the knowledge itself, whereas a computer scientist is more driven by practical purposes.

**Controlled vocabularies, thesauri, taxonomies, ...**

You could wonder whether controlled vocabularies, thesauri, or taxonomies could also be considered as ontologies. In essence they could since they are all targeted at obtaining a common understanding about the terminology/concepts used within a particular context. Yet they differ from each other with respect to the amount of meaning that is specified, the notations used, and the context in which they are used.

A *controlled vocabulary* refers to a set of terminology that a group of people agree to use consistently. It could range from a simple glossary listing the terms, to a terminology base with strict definitions that is governed by a central authority. Sometimes it also contains a very basic referential mechanism between the terms (e.g. the 'see also' reference). At the very least it should always resolve terminological ambiguity.

A *taxonomy* could be seen as a controlled vocabulary in which the terminology is organised into a hierarchy. There exist different types of parent/child relationships (e.g. class/instance, part/whole) that can be used to establish the taxonomy. Yet most taxonomies are restricted to one particular type.

A *thesaurus* extends the idea of relating terminology to specify its meaning with two kinds of reference mechanisms. The first kind is of a taxonomic nature and is about generalisation/specialisation (e.g. broader term, narrower term). The second kind is about relating terminology horizontally (e.g. synonym, related term, scope note). The set of available relationships in a thesaurus is usually fixed, which is in contrast to ontology practices. As a consequence you are sometimes forced to introduce ambiguities due to this lacking flexibility [DS03]. For instance consider the inconsequent use of the broader/narrower term relationship in certain situations. A concrete example is '`author`' as a broader term for '`Dostojefski Fjodor`' versus '`Brussels`' as a broader term for '`Vrije Universiteit Brussel`'. It is clear that in the former case the broader term is used to denote a class/instance relationship, whereas the latter case refers to a containment relationship. Note that the inconsequent use is not due to a lacking enforcement on the part of the thesaurus administrators. The main reason is that they have to represent the terminology with this (insufficient) set of relations. In an ontology you can define your own set of relationships which makes it possible to capture the subtleties between both uses in the example.

In table 3.4 we present a number of definitions adapted from Merriam-Webster [MW05], ordered according to the amount of meaning represented in them. Note that the definition of ontology, if considered in a particular context and scope, could easily encompass the definitions of the others.

| | |
|---|---|
| *Glossary* | A collection of textual glosses (brief explanations) of specialised terms with their meanings. |
| *Vocabulary* | A list or collection of words and/or phrases, usually alphabetically arranged and explained or defined. |
| *Lexicon* | A book containing an alphabetical arrangement of words in a language and their definitions. |
| *Dictionary* | A reference book containing words, usually alphabetically arranged along with information about their forms, pronunciations, functions, etymologies, meanings, and syntactical and idiomatic uses. |
| *Thesaurus* | A book of words or information about a particular field of concepts, usually with a cross-reference system for use in the organisation of a collection of documents for reference and retrieval. |
| *Ontology* | A particular theory about the nature of being or the kinds of existents. |

Table 3.4: A subjective ordering of 'repositories' according to the amount of meaning represented.

### Intended purpose of an ontology

An important aspect of an ontology is the intended purpose. Uschold and Gruninger identify three basic uses for ontologies [UG96]. The first is to improve communication between humans and/or computers. This is mainly done by providing an unambiguous representation of the concepts used. A second area of applicability is to achieve inter-operability among computer systems. In such a case the ontology acts as an inter-lingua. The last category is the least specific and concerns all uses of ontologies for improving the process and/or the quality of engineering software systems. Among others they foresee possible improvement for reuse and maintenance in this context.

### Ontological commitment

An important aspect of an ontology is the referential role that this explicit specification (the concepts) plays. This means that it is used as a work of reference and that there exists a strict commitment from the users towards the meaning of these concepts i.e. an *ontological commitment* is enforced. These commitments define the terms that allow you to reason about the world (or a specific part of the world). Consider for example defining '`tape`' as '`something sticky to hold broken things together`' within the domain of '`restorer of antiques`'. Whenever you use '`tape`' in this domain, people will know that you are not going

| | |
|---|---|
| *Ontology* | The branch of philosophy which is concerned with the nature and the relations of being. |
| *ontology* | An explicit, shared representation of a conceptualisation that holds in a particular context. |
| *Conceptualisation* | An abstract, simplified view of the world that we wish to represent. |
| *Ontological commitment* | An agreement to use the elements of a particular ontology in a consistent and coherent way. |

Table 3.5: Definitions for section 3.2.1.

to use it to record TV programs. However this does not exclude the existence of different viewpoints on the same concept. What this does tell us is that within one ontology there must be a consensus on which viewpoint is taken. Nevertheless if both viewpoints are necessary (or no consensus to discard one of them is reached), they could both be represented as separate concepts within the same ontology. The way to do it is to create two disjoint concepts and clarify the difference in perspective between the two.

**Summary**

At this point we come back to the definition for ontology by Gruber [Gru93]. He states that an ontology is an explicit, shared representation of a conceptualisation that holds in a particular context. The 'conceptualisation' in this definition refers to an abstract, simplified view of the world that you wish to represent for some purpose. That there must exist a kind of agreement between the users of the ontology is indicated by the word 'shared'. The purpose of making it explicit is for being able to share and use the conceptualisation as a point of reference. If we instantiate this definition in the context of this dissertation we get that an ontology represents a certain view on a business domain, in which you define the concepts that live in this domain in an unambiguous and explicit way for the purpose of building a software system. Interesting survey articles that introduce the basics of ontologies are [Kal02] and [UG96].

### 3.2.2 Different Varieties of ontologies

Ontologies come in different flavours. In part this has to do with the ontology representation that was used, since the associated expressiveness and formality leads to different capabilities. More important for us however are the granularity, and the intended purpose of the ontology. Both these aspects of an ontology are closely intertwined.

The granularity of the ontology boils down to the level of detail at which concepts are defined. This is of course in function of the intended purpose of the

ontology. A broad distinction can be made between ontologies that are general enough to be (re)used across different domains and ontologies that are dedicated to a specific domain [CJB99, Gua98, GGM+02]. The former type are referred to as *upper ontologies* (a.k.a. top-level ontologies), whereas the latter type are denoted by *domain-specific ontologies* (a.k.a. application ontologies). A possible layering of ontologies is shown in figure 3.7 (1).

An example upper ontology is the Suggested Upper Merged Ontology [NP01] which is used within the IEEE Standard Upper Ontology Working Group [SUM05]. It is intended to be used as a foundation for domain-specific ontologies by providing a set of definitions for general-purpose terminology. This means that it provides the core concepts or core ontology used by the underlying domain-specific ontologies. Similar to horizontal domains and vertical domains (section 3.1.1), an upper ontology is sometimes referred to as a *horizontal ontology* whereas a domain-specific ontology is sometimes referred to as a *vertical ontology* [Lin03].

*Foundational ontologies* are intended to be used as the core of other ontologies. Even though a foundational ontology can be considered as an upper ontology, it is intended to be used by existing upper ontologies as well. An example of such an ontology is Dolce [GGM+02]. Dolce is specifically targeted to be reused by a broader audience. Consequently the concepts in it are often extremely general and abstract.

The use of upper ontologies, to provide the core concepts for your own ontology, implies that you submit to their view on the world. Certain authors argue that trying to establish a set of upper ontologies is not feasible. Their main argument is that there is no self-evident way of dividing the world into concepts (see also section 3.2.3.2). Hence they believe that standardisation efforts are in vain. On the other hand, upper ontologies are thought to be extremely useful as a medium for standardising communication and as a way to facilitate knowledge sharing. Be what it may, such top-level ontologies can provide a good source of knowledge about abstractions for e.g. space-, time-, matter-, and event-related concepts.

Using an upper ontology in your own application context boils down to specialising the available concepts until you reach a level that is useful for your application domain. As we have already indicated, the set of concepts that you obtain in this way is a domain-specific ontology. Note however that stand-alone domain-specific ontologies that are not an extension of an upper ontology also exist.

Domain-specific ontologies are often subdivided in domain ontologies and task ontologies [Kav03]. The former defines the concepts associated with a specific domain, whereas the latter defines the concepts related to the execution of a particular task or activity. Whether this is a useful distinction depends on the definition of 'domain' that is used, since the definition of a domain ontology could entail the definition of a task ontology. The importance of this distinction lies in the fact that both flavours of knowledge are to be represented.

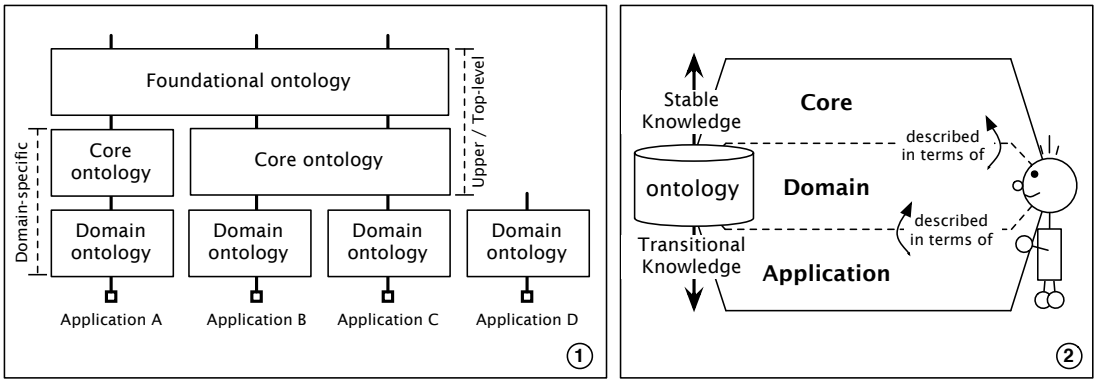



Figure 3.7: (1) A possible layering of different types of ontologies and (2) different layers within one ontology depending on the viewpoint of the user.

Until now we have considered 'types' of ontologies. Yet there is a similar division *within* an ontology itself. This is illustrated in figure 3.7 (2). As is shown you can distinguish between 3 layers: core, domain, and application. Concepts in the core layer are considered to be representing stable knowledge whereas concepts in the application layer exhibit a more transitional behaviour. As depicted in the figure, the concepts in the application layer are described by using the concepts provided in the domain layer. We could say that the domain layer concepts are used as an ontology for the application layer concepts. The domain layer itself makes use of the core layer. The core layer provides the most primitive concepts that are needed. This is similar to the distinction made between the layers in model driven development as presented by Bezivin and Ploquin [BP01]. The core layer can be thought of as a meta-language (e.g. EBNF) to specify the concepts in the domain layer. The domain layer itself will be used as a language (e.g. an ANSI C grammar) for specifying the concepts in the application layer (e.g. an ANSI C program). All layers within the figure are virtual. Consequently depending on the viewpoint of the current user (i.e. the current application context), the concepts can shift to different roles.

### 3.2.3 Ontologies in Practice

> "The philosopher wants to make sure that the ontology behind the system accurately reflects the world as it is and that any logical theory behind the design is sound and complete. The informatician may very well want the same thing, but realises that perfection must be sacrificed in the interest of finishing the job on time." [MS04]

We have seen that the word ontology has a different meaning in computer science and in philosophy. In the former case it refers to an artefact, and consequently the

| | |
|---|---|
| *Upper ontology* | An ontology that describes very general concepts like space, time, matter, object, event, and action, independent of a particular problem or domain. |
| *Foundational ontology* | An ontology that describes concepts that can be used as a foundation for building upper level ontologies. |
| *Domain-specific ontology* | An ontology that describes the vocabulary, tasks, and activities related to a generic domain. |
| *Vertical ontology* | A domain-specific ontology defining concepts native to a vertical domain (e.g. domain-specific ontologies). |
| *Horizontal ontology* | A general ontology that spans multiple domains, where it defines concepts for a horizontal domain (e.g. upper level ontologies). |
| *ontology granularity* | The level of detail at which concepts inside an ontology are defined. |

Table 3.6: Definitions for section 3.2.2.

question 'what is an ontology' requires a practical answer with some guidelines on how to build it. We have also seen that an ontology in this sense should not be considered without the application context (or the intended purpose). Hence it is important to keep this application context in mind while making a number of practical choices for developing the ontology. In no particular order, you need to:

- Define the domain and scope of the ontology
- Decide to follow a class-based or prototype-based approach
- Consider existing ontologies for (re)use
- Decide on a suitable knowledge representation and degree of formality
- Identify a set of concepts to populate the ontology
- Establish a connection between the ontology and the application

We will elaborate shortly on each of these in the following subsections. This is not intended to be a complete guide that covers all aspects of ontology development. Certain aspects such as ontology merging, ontology alignment, and ontology evolution are not covered in this dissertation. A good starting point to get a hands-on feeling of building an ontology is the article by Noy and McGuinness [NM01].

#### 3.2.3.1 Define the domain and scope of the ontology

An ontology cannot be separated from its purpose within an application in the sense that if we want to encode concepts we always have to decide what aspects of reality we choose to represent [CJB99]. Consider for example an ontology in the domain of computer hardware. If the ontology is to be used by a teaching system for computer novices we focus on different aspects of reality than if it were used in the context of a hardware manufacturer. In part this boils down to defining the level of detail of what we want to represent inside the ontology. For the teaching system, the *concept granularity* could well be at the level of abstract components inside a computer such as the central processing unit, hard disk drive and random-access memory. It is clear that this granularity is not always adequate when deploying an ontology for a hardware manufacturer. He might want to represent the types of resistors used on a particular model of motherboard for instance. Related to the granularity of the ontology content, is the knowledge representation that is used. This has to do with the expressivity of the representation as well as with the associated inferential mechanisms. We will discuss this in section 3.2.3.4. Note that defining the ontology domain and the scope of the ontology is similar to defining the boundaries of a business or application domain and the coverage of such a domain. As we will see in section 3.2.3.3 a number of ontologies are available for reuse within an application. If the ontology you are going to develop is specifically intended to be reused across a large set of applications or within a broader community, this influences the choices made during the other steps.

#### 3.2.3.2 Decide to follow a class-based or prototype-based approach

In computer science, ontologies are mostly used similarly to the class-based object oriented paradigm. This means that concepts in an ontology correspond to classes that can be used as a blueprint to describe the things that exist in the real world. Hence in order for some concept to exist, there should first exist a class that describes the characteristics of that type of concept. Philosophically this is in line with the Aristotelean view on the world. As we have seen in section 3.2.1, Aristotle defined a number of categories in which all concepts in the real world should fit. Most ontology applications adhere to this *classification* approach. Consequently you instantiate the concepts inside the ontology to represent the entities that exist inside your application. For example a '64-bit PowerPC G5' processor is an instance of the concept 'central processing unit'. This implies a strict division of ontology development and ontology usage.

The class-based view of the world is not the only one. A number of philosophers have argued against this rigid classification-based way of looking at the world. In this case we move towards ontologies used analogously to the prototype-based object oriented paradigm, where the actual objects are represented instead

of searching for classes that capture the commonalities of a group of objects. The main critique on the class-based view is that there are no single objectively right ways to classify things. The argument, among others, is that a person's perception of reality is highly biased by personal experience and cultural background. The most notable opponent of the class-based view is Wittgenstein, who states that it is extremely difficult to define beforehand which characteristics of concepts are essential and which are particular. To strengthen his case he uses the concept of a game (e.g. a ball game, chess, tennis) as an example [Wit04] . With this example he illustrates that even though there are a lot of similarities and relationships between games, it is impossible to find a unique set of properties that is common to *all*. The most common argument in favor of prototypes is that people apparantly find it easier to deal with specific examples first before generalising them into abstract concepts (whereas a class-based view works the other way around) [Tai97].

A class based approach seems to work extremely well for taxonomising 'concrete' things in the world such as plants, and animals. Nevertheless describing or capturing precisely 'what' these things are remains difficult. In part this is due to the fact that classical definitions (i.e. a single set of jointly sufficient and individually necessary conditions) are only possible for a relatively small set of concepts. An example that is similar to the game-example of Wittgenstein is the lemon example of Swartz [Swa97]. Even though everybody knows the extension of a game or a lemon (i.e. an actual set of games or lemons), it is extremely difficult if not impossible to describe their intensional definition (i.e. a definition that holds for all kinds of lemons or games)[3]. Moreover, opinions to whether a certain property is essential or not may differ. In part these differences in opinion are stimulated among others due to the fact that classes are non-overlapping and mutually exclusive. This means that something must belong to the *extension* of one class or the other, but that it cannot belong to both. Consequently a choice needs to be made due to this rigourous membership restriction. In computer science and more specific in domain analysis, this restriction is cumbersome since we need to capture a lot of things that are not concrete objects. An example of this is the approval procedure of a loan in which a knowledgeable financial advisor makes a personal assessment. This assessment is often based on know-how instead of a predefined set of fixed rules. The reification of such knowledge calls for a more flexible way of organisation. This can be done with *categorisation*, where the membership of something in a group is non-binding. An in-depth discussion of the difference between classification and categorisation is provided in

[3]The difference between an *intensional* or *extensional* definition of a concept is important. Consider for example the term `person`. The intensional definition refers to the specification of attributes and properties that are common to all persons. This is similar to representing a person as a class, which makes it possible to use it as some kind of formula. On the other hand, the extension of the term refers to the actual set of all persons that conform to this intensional definition.

an article by Jacob [Jac04]. Note that the choice of following a class-based or prototype-based approach has a strong impact on the steps described in the other subsections.

#### 3.2.3.3 Consider existing ontologies for reuse

One of the main purposes of an ontology is to get an explicit account of the meaning of the main concepts in a certain universe of discourse. Consequently an ontology is intended to be (re)used by different people at least within one domain. Ideally an ontology can be used within a variety of domains (section 3.2.2). Even though a lot of ontologies are proprietary and can only be used for a specific application or domain, several initiatives exist that are targeted at such a broader audience. Hence to make a headstart when developing your own ontology, it is important to look at what is already available on the market. Besides a number of closed commercial ontologies (e.g. Cyc [Cyc05]), a lot of open ontologies exist as well (e.g. OpenCyc [Ope05], SUMO [NP01], WordNet [Wor05]).

When selecting or assembling a set of already existing ontologies, special care should be taken for aligning the concept sets (a.k.a. *ontology alignment*). Consider for example the division of the world at the first level of concepts under the root. In SUMO we see a distinction between an `Abstract Entity` and a `Physical Entity`. The latter is further subdivided into `Content Bearing Physical Entity`, `Object Entity`, and `Process Entity`. In Cyc we see a distinction between an `Individual Thing`, a `Partially Intangible Thing`, and a `Mathematical Or Computational Thing`. This means that if you would like to use a part of each of these ontologies, that you need to merge both sets of concepts. Note that this is not a simple union, but that this requires a careful alignment of the semantics of the concepts that are classified under this upper-level. Besides this, adopting a deep taxonomy with rich semantics also imposes a risk of obfuscating the intended meaning. Since it is often difficult to grasp the semantic subtleties that occur in a domain other than your own, it is dangerous to go shopping for concepts in a careless way. Especially when merging ontologies, ambiguous concept definitions might occur when concepts are taken out of their defining context.

#### 3.2.3.4 Decide on a suitable knowledge representation and degree of formality

A large set of languages is available to represent the concepts in an ontology. The best known are Ontolingua, RDF, SHOE, OWL, and FLogic. A number of them stem from research in expert systems and knowledge bases (e.g. LOOM [Bri93], CycL [Cyc05]). The main categories of such knowledge representation schemes as identified by Mylopoulos [Myl80] are: logical, procedural, network, and structured representation schemes. This does not mean however that a representation

implementation must fit exactly into one of these classes. It is common to have a language whose main features fit mostly in one category but which exhibits features of other categories as well.

Well-known examples of the *logical* representation class are Prolog and Description Logics. These languages use expressions in logic to represent the knowledge elements and are especially suited for inferencing. *Procedural* representation schemes make use of a set of instructions to represent the knowledge. An example of this is a production system in which the knowledge is expressed by if-then rules. *Network* and *structured* representation schemes capture knowledge in a graph by representing concepts as nodes, and relationships as arcs. Semantic networks and Conceptual graphs are examples of the former. Structured representations can be seen as an extension of network representations since they represent nodes in the graph as a data structure with slot-value pairs. Examples of these are frames and objects. It should be noted that currently a lot of new ontology languages are created for use in internet applications (more specifically the semantic web). Such languages are often based on XML, since ontology interaction in these applications is often one of the prime requirements.

Whether to prefer a formal logic representation language over e.g. a less formal structured representation language depends again on the application context of the ontology. In this light we would like to present a quote of Nonaka and Takeuchi ( [NT95], page 67):

> "Once explicit concepts are created, they can then be modeled. In a logical model, no contradictions should exist and all concepts and propositions must be expressed in systematic language and coherent logic. But in business terms, models are often only rough descriptions or drawings, far from being fully specific. Models are usually generated from metaphors when new concepts are created in the business context."

Hence one needs to consider the level of formality that is required and that is possible in the application context. The benefit of a formal representation has a clear impact on the inference capabilities, as well as the possibility to specify an exact semantics of the concepts. As indicated by Corcho et al., a trade-off should be made between the readability, the expressiveness, and the inference properties of the ontology representation language [CGP00]. This also relates to the granularity of the knowledge representation. For example concepts in semantic networks can only be denoted by using simple symbols, whereas in frame-based systems one can define complex structures to define a concept [LS98].

In a minimal form one should be able to specify concepts and relationships. Most ontology representation languages combine the features of the four categories by Mylopoulos (logical, procedural, network, and structured representation schemes). Defining a concept is mainly done by listing its slots or attributes (a.k.a. properties, roles). Example slots of an insurance `policy holder` could be

`name`, `age`, and `profession`. Corresponding values for these slots could be `'Homer Simpson'`, `45`, and `nuclear safety inspector`. In the case of `profession` one sees that this slot-value does not necessarily point to a terminal fact, but that it could point to another concept. This means that we have defined a `profession`-relationship between both concepts. Facets (a.k.a. role restrictions) on these slots are used to specify among others allowed value-types, cardinality constraints, and default values. So they provide a way to put restrictions on slots. The relationship we mentioned earlier is an open association with no predefined semantics. However over the years a number of relationships have become common, namely to facilitate *classification* (`instance-of`, `member-of`), *aggregation* (`part-of`, a.k.a. meronymy), *generalisation* (`is-a`, `subclass-of`, a.k.a. subsumption), and *partitioning* (`group`, `context`) of the concepts.

Note that we have restricted ourselves to an extremely brief overview in this section since we do not focus on the creation of a new ontology specification language. An evaluation of ontology specification languages according to a set of desired features is given in [CGP00] and [BGH02]. An overview of ontology languages was also produced by the OntoWeb project in [Bec02].

#### 3.2.3.5 Identify a set of concepts to populate the ontology

Populating an ontology is an iterative activity. In fact it is similar to performing a domain analysis in which you undertake a number of steps to capture a good set of domain concepts. As we have already seen in section 3.2.3.3, a sub-activity of this step could be to identify already existing ontologies for reuse. Nevertheless besides this you need to create a set of custom concepts as well. Depending on whether you use a class-based or prototype-based approach, you need to determine the initial classes or the initial concepts ('instances') in your ontology.

In a bottom-up development approach you start by defining the most specific concepts. Consequently you iteratively group these into more general concepts. An example extract of concepts in the Wine ontology of Noy and McGuinnes [NM01] are `Margaux`, and `Pauillac`. These are consecutively grouped into `Medoc`, `Bordeaux`, `Red Wine`, and `Wine`. A top-down development approach starts by defining very general (and often extremely abstract) top-level concepts. Examples of these from the code-level ontology of Welty [Wel95] are `Software-thing`, `Software-object`, `Software-value`, and `Software-action`. At the bottom-level one would reach concepts such as `self-variable`, and `assignment`. Of course a combination of both approaches is also possible. Once you have this set of concepts, you need to define them more clearly. In part this is done by establishing a hierarchical taxonomy (with vertical relationships) in which you relate the concepts to each other with the 'kind of' relationship (a.k.a. 'is a', 'instance of'). Next you need to define the properties or attributes (a.k.a. slots) of each concept. This is done by relating the concept to other concepts with horizontal relationships, or by describing terminal facts about the concept. An example of this is `Pauillac`

`has-color Red`.

An important decision that must be made when modeling the ontology is to choose whether to represent a concept as a class or an instance. Note that this decision still needs to be made, even when a prototypebased approach is followed. This is because the need to be able to model abstract and concrete concepts is often dictated by the application context. Noy and McGuinness introduce two rules of thumb to guide a decision on this issue [NM01]. The first rule states that the most specific concepts in the ontology are individual instances. Whereas the second rule states that concepts should be represented as classes if they form a natural hierarchy. The main benefit obtained by using a prototype-based approach is the fact that you are not forced to model all concepts as classes before you can define their corresponding instances. You can introduce (or for that matter simulate) a class-structure in a prototypebased environment if the need arises.

Christopher Welty and David Ferrucci have examined the class versus instance problem in detail [WF94]. To illustrate their point they use an example in which they represent an `eagle` as an instance of the class `Species`. In such a case problems arise if you need to include particular eagles in your ontology as well. For instance the eagle `harry` should be represented as an instance of `eagle` but this would lead to a situation where you have instances of instances. Remodelling the ontology where `Eagle` is a subclass of `Species` leads to other problems. Since for example the statement 'the eagle is an endangered species' is not understood to apply to one particular eagle `harry`, but to the class of eagles (i.e. `harry` is not necessarily endangered). They solve the problem by introducing the notion of spanning objects, where the same concept plays the role of instance or class according to the universe of discourse in which it is used. It is clear that switching the role of such a concept is not an issue in a prototype-based ontology environment.

#### 3.2.3.6 Establish a connection between the ontology and the application

It is important to obtain a loose coupling between the ontology and the application. With such a loose coupling you achieve a certain amount of independence between both parts. Consequently it would become possible to use the application with a different ontology and vice versa. This way one could achieve ontology reuse as well as application reuse for different contexts. Mind though that changing the semantics of concepts in the ontology could change the way the application functions. This means that certain parts of the software is malleable through concept evolution. This leads to a certain degree of flexibility for adapting the way the application behaves or the results that are being produced. Achieving such malleability is not just a matter of maintaining a flexible connection between both components on a technical (code) level. More importantly it

has to do with how the application uses the semantics of the concepts inside the ontology. As we already indicated in section 3.2.3.1, it is important to take the application context of the ontology into consideration. Speaking in broad terms one could distinguish between a passive and an active use of the concepts in the ontology. This is an important distinction since the concept-centric environment we present in the next chapter will be categorised as ontology-driven.

A certain group of applications use the concepts of an ontology in a 'passive' way, for example to facilitate searching through a large amount of multimedia items. This type of application broadly corresponds to an *ontology-aware* application [Gua98]. The ontology concepts in such an application are used as metadata tags that direct the search activities. As indicated by this example, the word 'passive' refers to the fact that the application itself does not depend on the actual concepts inside the ontology. In essence one could state that the application behaviour does not change when the semantics of the concepts is changed. What could happen is that different search results occur, but what will not happen is a change in the overall application behaviour. Consequently since the concepts in this kind of application are mainly consulted and used passively, the runtime software malleability could be considered as low.

Applications that make use of the ontology concepts in an 'active' way depend on certain concepts in order to function properly. This type of application broadly corresponds to an *ontology-driven* application [Gua98]. An example of this kind of use is an insurance application in which the rules for processing a claim as well as the meta-model used, are stored as concepts. As a result, such an application cannot function properly without the existence of these core concepts in the ontology. This is mainly a result of factoring out part of the behaviour to the ontology. The ontology becomes an active component in the whole of the application. This is analogous to the development approach used in dynamic object models [RTJ05] where object types (classes) are represented as data (objects) in order to improve the adaptability of a software system. In practice you initially need to bootstrap such an application by hardcoding certain concepts inside the application. At a later stage, when the ontology and application infrastructure are more developed, you can eliminate these hardcoded elements by pushing them to the ontology. In this kind of application the consultation of the concepts is done in part to ensure the applications routine functioning. The concepts that are consulted are used actively, since their semantics are interpreted by the application during its execution. In such an application the runtime software malleability could be considered as high, depending on how far this kind of genericity is realised.

## 3.3 Summary

In this chapter we elaborated on domain knowledge and ontologies. We distinguished between several types of domains such as application domains, business domains, vertical domains, and horizontal domains. We also characterised the knowledge associated with a domain and discussed the difference between explicit and tacit knowledge. In addition we positioned domain analysis in the context of domain engineering. The topic of ontologies was discussed in detail since we will base our medium to capture domain knowledge on it. This is why we distinguished between different varieties of ontologies and a number of practical considerations for building an ontology.

In the next chapter we weave the knowledge presented here into our approach.

# Chapter 4

# The Concept-Centric Environment

In the previous chapters we introduced the background information that is necessary to position and understand the work of this dissertation. This chapter introduces our approach as a concept-centric environment that is based upon a symbiotic integration[1] between the code level and the concept level.

Section 4.1 presents the need for such a concept-centric environment. It starts by defining the problem context for this dissertation. Next we present an analysis of the different problems encountered in the background chapters. These problems are all related to the absence of domain knowledge in the context of continued development. We start by analysing the reasons why domain knowledge becomes *implicit*. This is a problem since domain knowledge forms an essential ingredient for code comprehension. On top of that, domain knowledge is often *detached* from the implementation. This is unfortunate because there probably existed a clear link at some point in time. Moreover this coupling is one of the top information needs by developers if they are evolving an implementation. One of the main reasons why domain knowledge becomes implicit and detached is because it plays a *passive* role during the development of the software. This why developers are hesitant to spend time on making explicit what they know, since it does not help them to deliver the functionality of the software system. In addition, documenting an implementation is perceived as an *overhead* by developers. Consequently even when domain knowledge was made explicit, it will quickly become out-of-date and is no longer useful as documentation. We end the section with the problem statement which is based upon these problem keywords, namely: implicit, detached, passive, and overhead.

Section 4.2 discusses the concept-centric environment we propose in this dissertation. It is structured according to the following antonyms of the problems mentioned above: implicit versus explicit, detached versus coupled, passive versus

[1] With *symbiotic* we refer to a mutually beneficial relationship between the concept level and the code level. With *integration* we refer to the fact that both are closely combined so they form a whole.

active, and overhead versus transparency[2]. Broadly speaking we explore the basic underpinnings of the explicit concept level in this section. These foundations are validated with a practical implementation in the next chapter. As we will discuss, domain knowledge is made *explicit* by using a frame-based representation which is stored inside a ConceptBase. In order to be able to *couple* domain knowledge and implementation entities, we discuss the underlying mechanism that takes care of deifying[3] code level entities into the concept level. This makes it possible to connect both elements at the concept level. Subsequently we clarify how the concept level can *actively* contribute to the overall functionality of the software system. Among others this activation results in an improved malleability of the implementation by refactoring it, so that it makes use of the concept level. To alleviate the overhead for developers we stress the importance of *transparency*. This is mainly achieved through a symbiotic integration between the code level and the concept level. We end the section by synthesising all these elements into our thesis statement.

A proof-of-concept implementation of the environment is presented in chapter 5. Examples that illustrate the interaction with the environment can be found in chapter 6.

## 4.1 Need for a Concept-centric Environment

In order to survive in today's highly dynamic marketplace, companies must show a continuous and ever-increasing ability to adapt. This pertinent need for change has a clear impact on the software systems that support the business activities. The level of agility that is necessary to keep up with the business side of the 'adaptability challenge', consequently imposes itself on the software side. This puts contemporary software development practices under a continuous strain so as to achieve acceptable results within an extremely limited time span. In chapter 7 we present a characterisation of an example business context (iMedia) that illustrates the strain that can be put on software development practice. The business dependence on software and the need for agile software evolution is clearly expressed in the following quote [FK00]:

> *"The overall setting is characterised by on the one hand an increasing business dependence on reliability of software infrastructure and on the other hand rapid change and reconfiguration of business services necessitating rapid software development and frequent change to that software infrastructure."*

[2]With *transparency* we refer to the obliviousness of the concept level for a developer.

[3]In contrast to *reification*, where one makes something abstract more concrete, *deification* refers to making something concrete into something abstract.

In this dissertation we focus on continued development (chapter 2, section 2.1.3), which boils down to the adaptation of an existing implementation to incorporate new features (either anticipated or not). This also encompasses organisational and methodological issues, but our focus is on the programming side.

We focus our investigation on object-oriented applications that are written in a class-based programming language. The reason for this is that it is currently considered as the ruling paradigm for developing business software. Moreover we use a number of intrinsic properties of object-orientation[4] that are essential for a practical implementation of a concept-centric environment (see chapter 5). The applications that are of particular interest to this dissertation are E-Type systems (chapter 2, section 2.1.4). In summary this type is characterised as being in a constant state of change. Additionally these systems are closely connected to a real world domain, which means that there exists a two-way interaction between changes in the real world and changes to the application. Such an interaction is also referred to as co-evolution (chapter 2, section 2.2.3). As a result of this close connection to the real world, a lot of domain expertise is involved in the development of E-type systems. However we do not consider knowledge-intensive systems that are best developed by structured knowledge engineering approaches such as CommonKADS [SAA+00]. This is because our focus is on the programming side (and not the knowledge management side) of evolving object-oriented software.

We focus on development practices that are geared towards achieving malleable software. As we have described in chapter 2, section 2.1.7, continued development is also about making the software more open to incorporate future changes (a.k.a. anticipated evolution). An important activity in software evolution to achieve this flexibility is refactoring [Opd92, FBB+99]. Refactoring is targeted at enhancing the internal structure and the readability of the implementation so that changes can be more easily incorporated. Even though we will propose to refactor part of the code to enhance the software malleability, we do not target the activity of refactoring itself. Software malleability is important for E-Type systems, since it facilitates adaptations in an extremely limited time span.

We distinguished between two opposed visions of software development in chapter 2, section 2.2.2. We position ourselves in between the pure code-centric and model-centric approaches. In essence this means that we want to balance the importance that is attributed to both the code and the models. As will become clear in section 4.2, the kind of domain knowledge that we target and the way that this knowledge is made explicit is beneficial to both visions.

In the following sections we describe a number of problems that are encountered during the evolution of object-oriented E-Type software applications. These

[4]The properties we require are not intrinsic to class-based programming languages, but to object-orientation in general.

problems are described from a worst case perspective. Broadly speaking we identify three main sources of problems that are relevant for this dissertation, namely implicit, detached and passive domain knowledge.

### 4.1.1 Implicit Domain Knowledge

Up to 60 percent of the time spent in software evolution is dedicated to program comprehension (chapter 2, section 2.3). This is because a developer needs to understand what a system does before any adaptation can take place. In essence program comprehension is about rediscovering and reconstructing the domain knowledge that was available to the original developer. The fact that a lot of domain knowledge is implicit for the developer is problematic for evolution. Even though this is a rather plain observation, there is empirical evidence that domain concept descriptions are the most essential information needed by software maintainers [KSP04]. In chapter 2, section 2.3 we identified the main factors why knowledge about a software system becomes implicit. In essence they can be summarised as follows:

- **High personnel turnover**
  Software developers and domain experts are a volatile asset. They switch companies or project teams frequently. Moreover depending on the stage the software is in, it will become less likely to find people that are well-acquainted with the system.

- **Complexity of software systems**
  As a result of the complexity and the size of contemporary software systems, the peculiarities of a particular component are often lost over time. This is not only a result of the difficulty people have to grasp the entire system. It is also a consequence of the large number of dependencies between the different elements.

- **Short-term software development economics**
  Spending time on documenting a system is time that cannot be spent on development. As a consequence documentation is often neglected by developers in favor of producing code that contributes to the next release of the system. This happens especially in the context of agile development where continuous integration is mandatory. A development team will build and test the software many times a day. Consequently the next release of the system is not weeks but hours away. It is clear that in such an environment developers cannot be burdened with specifying their domain knowledge if it does not have a clear short-term return.

- **Kind of knowledge**
  Even in a model-centric approach to software development, not all knowledge is made explicit. For instance the motivation for particular design

choices is often left undocumented. Also common sense knowledge about the business domain in which the system operates is often unavailable in the code nor the models (e.g. as a result of a generic implementation). Even though certain design decisions may be clear for experts in the field, novice developers will often have difficulty to understand the (hidden) reasons for a particular choice. This poses a problem especially in agile software development where a guided approach is followed (chapter 2, section 2.2). A guided approach has an exploratory nature, thus deviations from a planned project trajectory are not considered as harmful but as beneficial. The motivation behind certain deviations could provide future developers with the necessary insight in particular implementation choices. Moreover it could also prevent them from reinventing the *wrong* wheel over and over again[5].

- **Genericity of the implementation**
  Continued development is also about refactoring the software to make it more susceptible to future changes (chapter 2, 2, section 2.1.7). A basis for enhancing the malleability of an implementation lies in writing generic code. Unfortunately generic code results in less explicit code. A lot of details become hidden in the data that is loaded at runtime, which has a negative impact on the readability of the code. Moreover this results in more knowledge to become implicit since generic code contains less explicit information. Less explicit code is not helpful for code comprehension since it has a negative impact on readability. As a result a trade-off exists between flexibility and complexity [Fow01, Fow98].

- **Reduced information quality**
  Each time information is passed on in a communication chain, there is a deterioration of the quality; not only when an evolution request is passed on from a domain expert to a developer, but also between successive evolution steps in the software. Each evolution step results in certain parts of the implementation to be rewritten. These transformation steps can cause a loss of information. In chapter 2, section 2.3.2 we illustrated the deterioration of the quality of information when it is passed in a small communication chain. In figure 4.1, we revisit the chain presented in that section but now with a shared repository in which part of this knowledge is made explicit. Without making any quantitative claims, one could at least expect that the total quality at the end of the chain is going to be larger than the original 60 percent. Moreover, a developer needs to map the need for evolution onto the implementation. This results in another deterioration of

[5]Documenting the reasons why a particular solution is chosen is important information for developers. This for instance tells them which alternatives have been tried and why they were not considered as valid solutions. As a consequence, the developer can use this knowledge in order to decide how to refactor the system to accomodate the new situation.

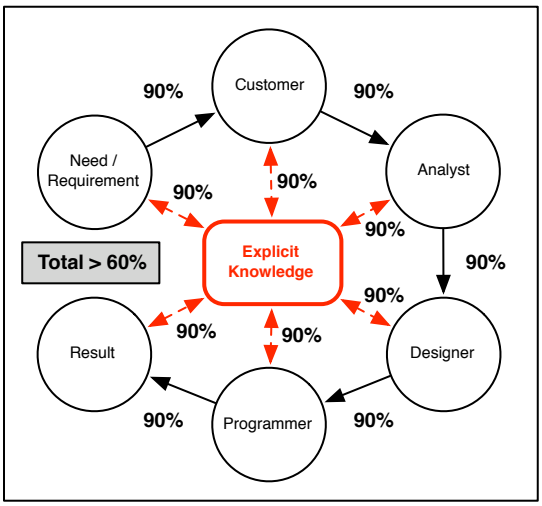



Figure 4.1: Improved quality of communication with an explicit representation.

the quality because the need is formulated in terms of the business domain. It is the developer's responsibility to translate and map this need into the implementation domain.

We have seen in this section that domain knowledge becomes implicit for different reasons. This is problematic in the context of software evolution, since it is the main asset for program comprehension. Hence this indicates the need for an explicit medium that captures the relevant domain knowledge.

### 4.1.2 Detached Domain Knowledge

Making the domain knowledge available in some explicit form is only a partial solution to the problem. Developers need to know to which part of the implementation the knowledge applies to. Unfortunately the knowledge is often detached from the implementation. This means that there is no explicit link available between the knowledge and the code. Empirical evidence shows that program-domain-situation knowledge is one of the top three most needed information of developers (see chapter 2, section 2.3.3). Consequently they are not only interested in domain concept descriptions, but also in how they can couple this knowledge back to the implementation. This matching process was illustrated by the Brooks comprehension model which was described in chapter 2, section 2.3.

Mapping domain knowledge back to code is *difficult in nature*. This is mainly because domain knowledge cannot always be brought back to one specific location in the code. In contrast to encapsulated domains this is especially problematic for distributed domains (chapter 3, section 3.1.1). In an encapsulated domain the functionality that is associated with the domain can be lifted out of the soft-

ware in a relatively easy way. In a distributed domain however the functionality is scattered throughout the implementation. As a consequence there is no direct one to one mapping between the domain and the implementation since the domain functionality crosscuts the application. Since the mapping process between knowledge and code is difficult and time-consuming, it is important that a developer can attach his findings to the code once the mapping is complete.

The problem of detached domain knowledge is still valid when a case tool is used to model the application. First of all this has to do with the *kind of domain knowledge* we refer to that remains implicit. In the previous section we already indicated that certain domain knowledge is not recorded explicitly in the models or the code. An example of this is the common sense knowledge about the business domain in which the system operates. Secondly there exists a clear *dichotomy* of the traditional analysis-design-implementation phases. Broadly speaking this refers to the fact that the artifacts are decoupled each time control is passed from one phase to another. As a consequence, knowledge that is available in an explicit form will become detached from the implementation in the end. This dichotomy is less apparent when an agile iterative development approach is followed. This is because the distinction between analysis, design and implementation is less clear-cut in such an approach. However since only those things that are considered to be important are documented by models in agile development, it is even more important to have a clear coupling between the models and the implementation.

Also depending on the software system, this coupling between the knowledge and the documentation is important. For example in a framework developers need to know which parts to instantiate and which methods to override. This kind of knowledge is closely related to particular elements in the implementation. Thus it is important to keep this knowledge coupled to these particular elements.

In this section we have seen that developers do not only require explicit domain knowledge. They also need to know to which part of the implementation the knowledge applies to. This indicates the need for a mechanism that helps developers to make this link explicit.

### 4.1.3 Passive Domain Knowledge

When domain knowledge is made explicit and when it is linked to the implementation, developers need to keep this information up-to-date. Yet this is not evident as a result of short-term software development economics (see section 4.1.1). First of all this is due to the fact that developers do not perceive documenting as an activity that contributes to meeting their next deadline. As a consequence it is often neglected in favor of writing code. This is because the contribution of writing code to the next release of the system is clearly visible. One could say that the domain knowledge plays a *passive role* with respect to the execution of the system. We mean this in the sense that it does not participate actively to provide the functionality of the system.

Secondly, writing down the domain knowledge associated with a system is often done outside the development environment. This is perceived as an overhead by developers since they frequently need to *switch between environments*. A seamless integration of tools in such case is beneficial for reducing the overhead introduced by the documentation system.

From this you could conclude that the domain level and the implementation level live in *antibiosis*. This means that an adverse relationship exists between both sides in which one is adversely affected by the other. In essence this boils down to the fact that spending time on the concept level means less time for the code level.

In the context of agile software development, the fact that domain knowledge plays a passive role poses an even bigger problem. Here, requirements are elicited and refined on the go during the implementation of the system. Hence the software grows incrementally and is under constant revision. As a consequence it is unfeasible to have a conventional documentation system. If domain knowledge is made explicit in such a context, it will always be out-of-date unless the developers spend a lot of time on updating it each time the code is changed. It is clear that unless the domain knowledge contributes to the implementation, that developers will not or cannot dedicate any time to it.

In this section we have seen that one of the main reasons why domain knowledge becomes implicit and detached is because it plays a *passive* role. As a result it is not kept up-to-date and its usefulness as documentation will quickly degrade. This indicates the need for a mechanism that enables the activation and a way that alleviates the overhead perceived by a developer.

### 4.1.4 Problem Statement

Software developers responsible for evolving an existing implementation must grasp the intentions and assumptions made by the original developers of the system. This is problematic because most of the domain knowledge about the system is only available in some *implicit* or tacit form. Even in a context where such domain documentation exists in an explicit form, this documentation is *detached* from the current implementation. This means that the programmer will have to discover manually which part of the documentation matches which part of the implementation. The main reason why domain knowledge becomes implicit and detached is because it does not yield a short-term economic benefit for the developers. This means that it acts as as *passive* form of documentation. Consequently it is not maintained properly and will quickly become out-of-date. This is mainly due to the fact that developers perceive documenting what they know about the implementation as an *overhead*. The problems discussed above are even more pronounced in the context of agile software development, where the next release of the system is always only hours away.

In summary the following problems need to be addressed in the concept-centric environment we describe in the next section:

- **Implicit Domain Knowledge**
  The concept-centric environment must provide a mechanism that makes it possible to capture domain knowledge in an explicit form. This will reduce the possible loss of domain knowledge.

- **Detached Domain Knowledge**
  The concept-centric environment must provide a mechanism to couple the domain knowledge to the implementation. This will reduce the effort spent by developers on retroactively matching the knowledge to the corresponding implementation.

- **Passive Domain Knowledge**
  The concept-centric environment must provide a way to involve the domain knowledge actively in order to provide the functionality of the software system. This will motivate developers to pay attention to the domain knowledge, since it potentially yields a short-term benefit. Moreover an enhanced malleability of the software can be achieved by devoting time on the concept level.

- **Overhead for the Developer**
  The concept-centric environment must be built in a way that it is not perceived by developers as an overhead. This means that the interaction with the concept environment must be as transparent as possible.

## 4.2 The Concept-centric Environment

In the previous section we identified four key points related to the absence of domain knowledge in the context of evolving a software system. We already hinted at the requirements for the concept-centric environment we discuss in this section. We will now couple these problems to the different elements of our solution. This is based upon a symbiotic integration between a concept environment and a programming environment. In essence we base the discourse of this section on the following 'antonyms' for the problems that were identified:

- From implicit towards **explicit** (section 4.2.1)
  Discusses our view on an *explicit concept level representation.*

- From detached towards **coupled** (section 4.2.2)
  Describes how we envision the *coupling of code and concepts.*

- From passive towards **active** (section 4.2.3)
  Elaborates on the way we see the *active participation of the concept level.*

- From overhead towards **transparency** (section 4.2.4)
  Explains the *transparency* that is achieved by a *symbiotic integration* of the code and concept environments.

The following subsections are structured according to these cornerstones of our solution. We conclude by presenting the thesis statement for this dissertation in section 4.2.5. In figure 4.2 we present a legend for the main elements in the drawings of this chapter.

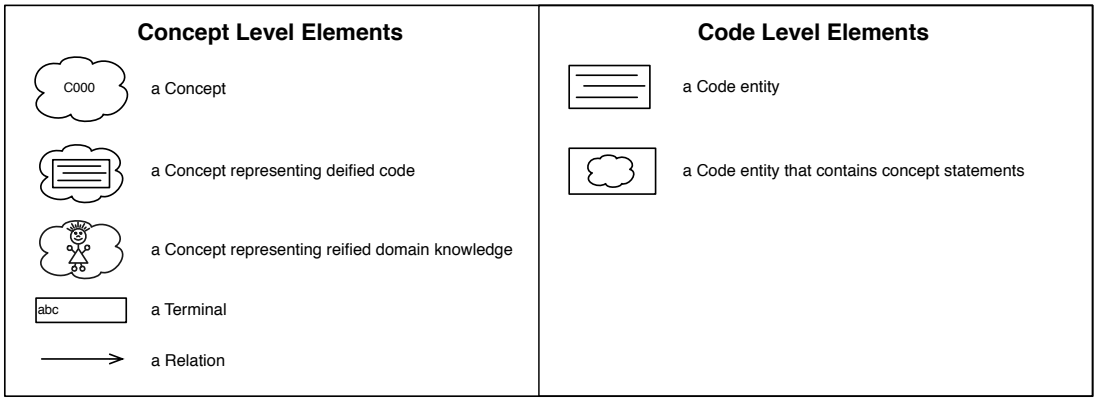



Figure 4.2: Legend for the concept drawings.

### 4.2.1 Explicit Concept Level Representation

Having access to the underlying domain knowledge of a software system is of paramount importance during evolution. This is why we set up an ontology inspired infrastructure in which the domain knowledge is made explicit.

Let us consider the definition for an ontology we presented in chapter 3, section 3.2.1: "an ontology is a shared and explicit representation of a conceptualisation". If we instantiate this definition then the explicited set of domain knowledge, the *ConceptBase*, will act as an ontology. The ConceptBase first of all contains an explicit representation of the domain knowledge associated with the application domain under consideration. As a consequence it acts as an explicit representation of a conceptualisation. Secondly, the developers and the domain experts commit to using the domain concepts as defined in the ConceptBase. Hence it is not only an explicit representation, but also a shared representation.

We do not focus on formal ontologies and foundational ontologies in this dissertation. In part this is motivated by the fact that we do not wish to introduce elements that can be perceived as an extra overhead by the developer. We envision an open lightweight representation that does not introduce an extra overhead for a developer during the evolution of a system in an agile context. We return

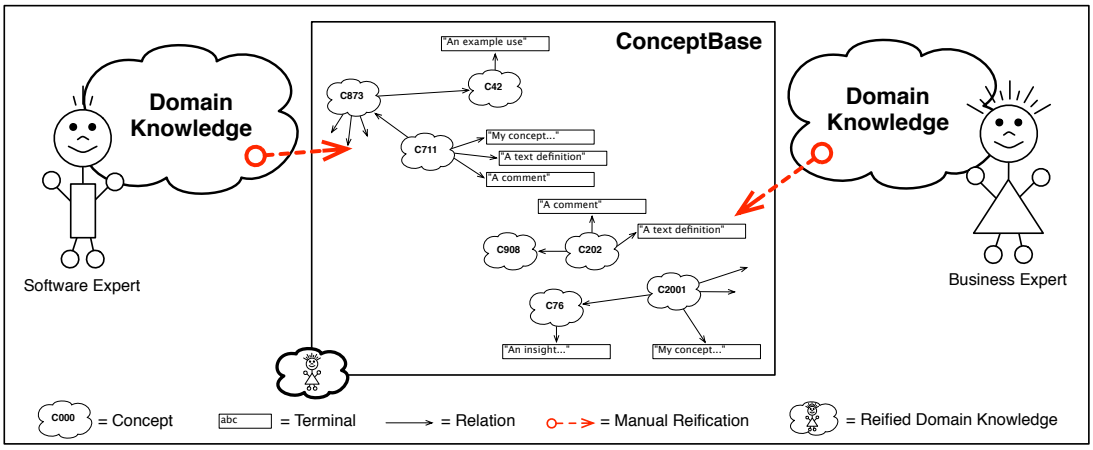



Figure 4.3: Domain knowledge is manually reified inside the ConceptBase.

to this in a subsection below on the knowledge representation for the ontology. Note that even though it is desirable, we do not set ourselves the goal to end up with a collection of domain concepts that are reusable across different domains.

In figure 4.3 we illustrate how domain knowledge is made explicit inside a ConceptBase. This process where experts manually write down the relevant domain knowledge is referred to as *domain concept reification*. For example for an insurance software system it can be the reification of (problem domain) concepts such as `insurance object`, `contract`, `claim`, .... It can also be about (solution domain) concepts that describe certain assumptions made in the design of the code (e.g., why was a composite pattern used). As shown in the figure, both the domain knowledge of the software expert and the business expert is reified inside a concept network. The resulting ConceptBase contains reified domain concepts (represented by a cloud with a stick person inside) that are related to each other or to terminals.

In the following subsections we discuss the explicit concept level representation. We will first delineate the domain and the scope of the ontology after which we explain how the domain knowledge is represented in a ConceptBase.

### Domain and scope of the ontology

In chapter 3 we defined domain knowledge as the set of knowledge that captures the concepts and rules that govern a particular domain. We broadly distinguished between business domain knowledge and application domain knowledge. Generally speaking, the distinction corresponds to the origins of the domain knowledge. Business domain knowledge is rooted in the business (e.g., insurance, banking) whereas application domain knowledge is rooted in the software applications that are used in a particular business (e.g., insurance software, banking software).

We want to go towards a systematic capture of domain knowledge instead of the opportunistic way where it is only recorded from time to time. Hence we advocate a situation in which the domain knowledge required to understand and maintain a complex piece of software is no longer obscured within the code or the heads of experienced developers. This systematic explicitation of domain knowledge is essential to be able to cope with the current time-to-market driven software development environments.

In order to limit the scope of the ontology (chapter 3, section 3.2.3.1), the domain knowledge to which we refer can be broadly characterised as domain concepts and domain relationships (chapter 3, section 3.1.2). We do not consider for example rules that make it possible to infer new knowledge. We also do not focus on how to obtain this domain knowledge. For that purpose there already exists a large body of work in the field of knowledge management and knowledge engineering (chapter 3, section 3.1.2 and 3.1.3).

As we have seen in the discussion of the knowledge conversion process there are several conversion modes. In this dissertation we focus only on the externalisation mode (tacit to explicit) and the combination mode (explicit to explicit). In summary, externalisation creates new explicit knowledge (e.g. green-field domain analysis), and combination combines existing explicit knowledge into new explicit knowledge (e.g. restructuring existing domain models).

Even though it is not a constraint, we mainly envision problem domain knowledge that is associated with a vertical distributed domain (chapter 3, section 3.1.1). Generally speaking this knowledge is associated with a domain that refers to a set of applications within one business domain (e.g. insurance). Distributed in this context refers to the fact that the corresponding functionality is scattered throughout the implementation.

### Knowledge representation for the ontology

Making implicit knowledge explicit requires some kind of formalism in which the developer can express this knowledge. For our application context we choose an open lightweight formalism. This is intended to minimise the overhead perceived by developers when using the formalism. Moreover, the openness of the formalism is required to enable the developer to customise it according to the particular task at hand. As we will discuss in section 4.2.4, we prefer to use the same syntax and interaction mechanisms of the programming language to manipulate and access the concept level. This is in line with minimising the difficulties encountered by users as described by Shipman and Marshall [SIM99]:

- minimize the *cognitive overhead* perceived by the users
  (e.g. by using a syntax that is familiar to the user)

- not enforcing the user to commit to a *premature structure*
  (e.g. by not enforcing the use of class hierarchy if it is better to model in a

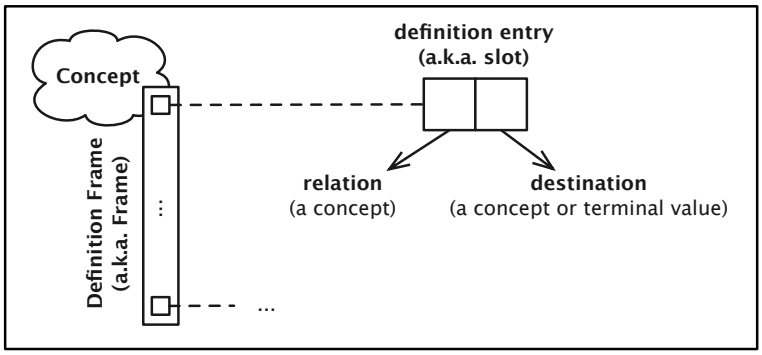



Figure 4.4: A concept is represented by a frame which holds a collection of slots.

different way)

- making it possible to *tailor the representation* according to the situation in which it is being used
  (e.g. by providing an open formalism that can be extended if necessary)

We use a *frame-based representation* (chapter 3, section 3.2.3.4) in which domain concepts are related to each other in a *concept network*. The concept network itself represents the ontology and is stored in the ConceptBase. The notion of frames was first introduced by Minsky in 1975 [Min75]. A frame is used to capture a typical situation or view of a situation. It represents knowledge as structured objects which consist of named *slots* with attached values. These slots can be used to relate a particular frame to other frames, thus forming a network of networks.

In figure 4.4 we illustrate the representation of a domain concept in a frame-based fashion. A concept is defined by a *definition frame* that contains its *definition entries* (a.k.a. slots). The definition entries consist of two elements: a *relation* and a *destination*. Destination values can be another *concept* or a *terminal*. A terminal represents a value about which we do not wish to record extra information in the concept network other than its value. Hence its definition frame does not contain definition entries but just the value that it represents. In essence the choice of representing an element as a terminal or a concept defines the *ontological granularity* in the concept network. The types of terminals that are supported in a concept network are extensible. This makes it possible to adapt the concept representation according to the particular needs of the developer. For example if necessary it should be possible to relate a concept to a drawing that clarifies its meaning. The relation that is used in the definition entry is also represented as a concept in the network. As a consequence the concept network is said to be self-describing, since the meaning of a relation is also described in the concept network itself.

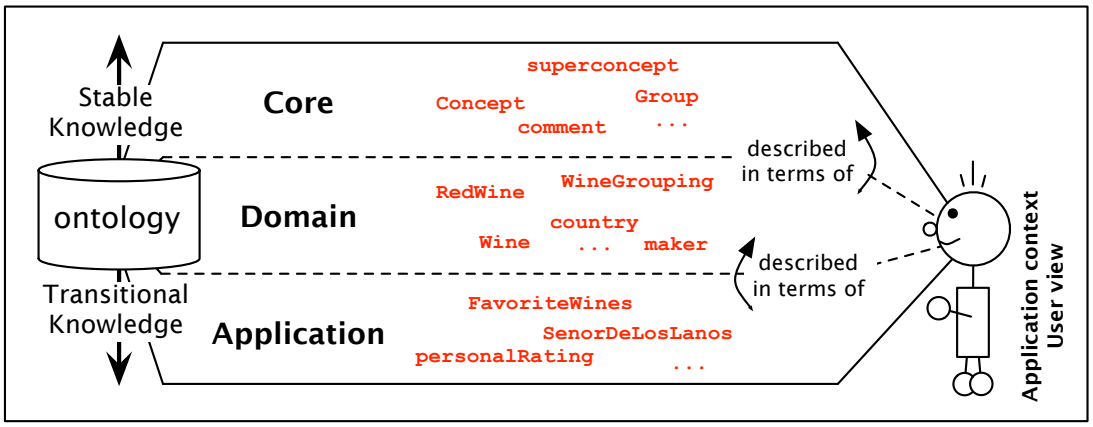



Figure 4.5: Virtual layers within an ontology.

We choose to follow a *prototype-based approach* for the ConceptBase (chapter 3, section 3.2.3.2). In a class-based approach the concepts are represented as classes in the ontology. So before a concept can exist, there must first exist a class that describes the general characteristics of that type of concept. In contrast a prototype-based approach represents the actual concepts without the requirement to specify the class first. Analogously to prototype-based programming languages this results in a more natural and highly dynamic way of interacting with the concepts. For instance, you do not have to switch between the class and object view to alter a concept definition. The choice for a prototype-based approach follows naturally from our goal not to enforce users to a premature structure. A class-based approach on the contrary would force the user to commit to a taxonomy of concepts from the start. Therefore it also enforces the use of existing relations (e.g. is-a) to set up the hierarchy. As a consequence a mismatch is often created if the pre-defined interpretation is not in line with the user's intention (chapter 3, section 3.2.3.5). Moreover, given the application context we envision, we want to maximise the flexibility a developer has to reify his domain concepts.

### Virtual layers in the ConceptBase

In chapter 3, section 3.2.2 we have already indicated that an ontology contains several layers. In figure 4.5 we extend the figure of chapter 3 with a number of example concepts from a wine ontology application. As is shown, the concepts from one layer are described in terms of the concepts from another layer. For example specifying the `FavoriteWines` grouping requires the `WineGrouping` concept, which in turn uses the `superconcept` and `Group` concepts. The core layer of the ontology represents the most stable concepts, whereas the application layer represents concepts that have a transitional nature. These layers are virtual, and depending on the viewpoint of the user of the ontology, concepts can switch roles.

The role played by a concept depends on the application context. For instance if the user is modeling the wine domain, then the `WineGrouping` concept will reside in the application layer since it is being modeled. This also implies that the `WineGrouping` concept in such an application context is more likely to be changed. Hence, the application context for the ontology is important.

In chapter 3, section 3.2.3.6 we distinguished between ontology-aware, and ontology-driven applications. This distinction is based upon the way that the concepts from the ontology are used by the application (either passively or actively). Up until now, the ontology is only consulted by a developer as a source for domain knowledge during software evolution. We can thus characterise the concept-centric environment as *ontology-aware*. As we shall see in the following sections, we move towards an environment that is better characterised as ontology-driven. We will reflect on this in section 4.2.3.

### 4.2.2 Coupled Concept and Code Level

Having access to an explicit set of domain knowledge is a first step to support developers during software evolution. Yet as we have seen in section 4.1.2, developers also need to know which part of the implementation the knowledge applies to. In this section we set up a mechanism that makes it possible to make the link between the concept level and the code level explicit. This way a developer can attach his domain knowledge about the code to the corresponding code entity.

In the following subsections we explain the underlying mechanisms that are required to couple concepts to code. We first explain how code level entities are conceptified into the concept level. Next we discuss a number of decisions that need to be made with respect to the granularity of conceptification. We end by illustrating what it means to couple domain concepts and conceptified code entities at the concept level.

#### Conceptification of code level entities

In figure 4.6 we illustrate how a code level entity is automatically deified towards the ConceptBase. The process of generating a concept representation for a code level entity is referred to as *code level entity conceptification*. At the left-hand side of the figure we see a snippet of a Smalltalk definition of a `Manager` class. On the right-hand side we see the concept network that is the result of the conceptification process[6]. The link between the code level entity and its concept level counterpart remains explicit at all times. In essence this means that both are inextricably linked to each other. As shown in the figure, the class `Manager` is represented as a concept that is related to amongst others three terminals representing its instance variables (`name`, `salary`, and `manages`). It is also related to a super

[6]Note that this example is for illustrative purposes only. See chapter 5 for a full example in which we explain the granularity of conceptification in more detail.

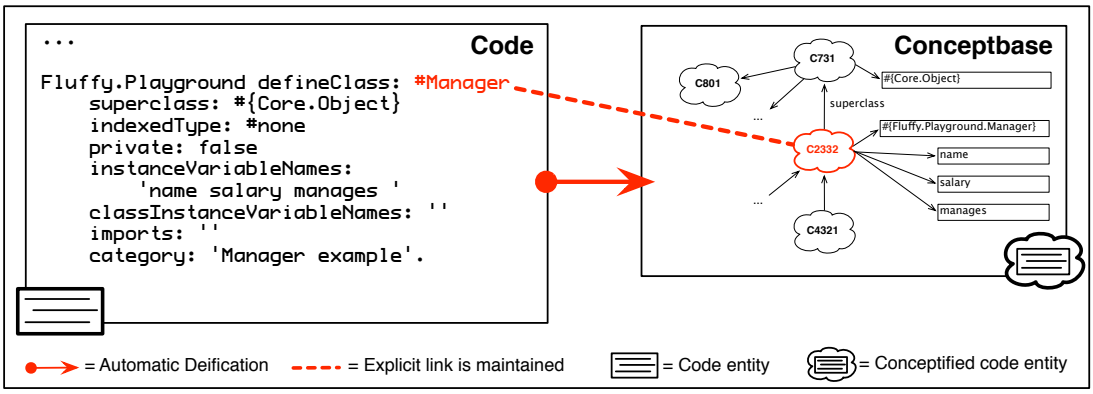



Figure 4.6: Code is automatically deified inside the ConceptBase.

concept representing its superclass `Core.Object`. Note that this super concept is inextricably connected to the `Core.Object` code level entity.

The conceptification process requires access to information about the code level. Hence this requires support for reflection from the programming language. Deciding what to represent at the concept level for a code entity is the first choice to make with respect to granularity (see chapter 3, sections 3.1.1, 3.2.2, 3.2.3.1). In essence this means deciding which slots to compute for the concept level. You can restrict, for example, the available slots at the concept level to instance variables, methods, and a superclass. The second granularity decision is how to represent the information you obtain about the code entity. This boils down to specifying whether the value of a computed slot contains a terminal or a concept. You could for example represent the value for a superclass slot as a concept and the value for an instance variable slot as a terminal. If a value is represented as a concept then it can be related to other concepts or terminals at the concept level. When a value is represented as a terminal then it has no definition frame but just the value. As a consequence it cannot be defined further at the concept level.

Depending on the amount of information that can be consulted about the code level entity, the process will be able to generate a richer definition for its concept level representation. For example if the process can consult the code level to obtain information about the superclass of the `Manager` class, it will be able to provide a link to it at the concept level. Since conceptification is an automated process, it follows a prescription based on the granularity that was chosen. This prescription defines what to represent and how to represent it, and is referred to as the *intension* of a concept (chapter 3, section 3.2.3.2). Note that since the concept representation of a code entity is computed, changes to the code entity are reflected at the concept level as well.

At the concept level, there is no difference between code level concepts that

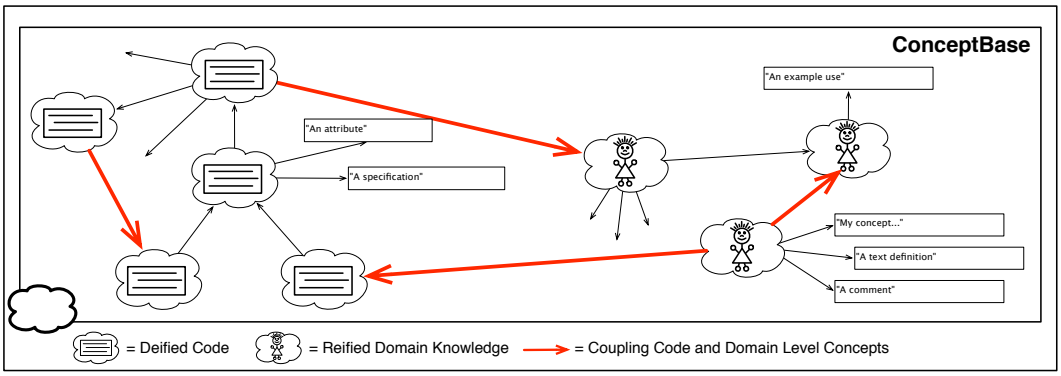



Figure 4.7: The reified domain concepts and the deified code concepts can be coupled inside the ConceptBase.

are the result of deification and domain level concepts that are the result of reification. As a consequence both can be manipulated in the ConceptBase using the same mechanisms. Furthermore this implies that the definition of a code level concept can also be extended with slots at the concept level. Slots that are added this way represent the *extension* of a concept (chapter 3, section 3.2.3.2).

The coupling between code level entities and domain knowledge is realised by providing an extension for a conceptified code entity. Since there is no difference between code level concepts and domain level concepts, they can be related to each other by adding a relation to a concept. This is illustrated in figure 4.7. As shown, you can add relations to the concept network in the ConceptBase between concepts representing reified domain knowledge (cloud with stick person inside), concepts representing deified code level entities (cloud with rectangle inside), or a combination. This means that it becomes possible to connect a domain concept such as e.g., `Insurance Policy Holder` to a conceptified class. This means that deified and reified concepts are treated uniformly from a technical perspective (hence they are represented by a simple cloud symbol if the distinction is conceptually unimportant).

Since the link between the code level concept and the code entity is explicitly available, it is possible to install a bi-directional navigation between both levels. This implies that it is possible to go from the code level to the concept level and vice versa. We discuss this further in the next section.

### 4.2.3 Active Use of the Concept Level

In section 4.1.1 we discussed several reasons why domain knowledge becomes implicit during software development. Here we will mainly focus on short-term software development economics and the genericity of the implementation. In summary the former referred to the problem that developers neglect making ex-

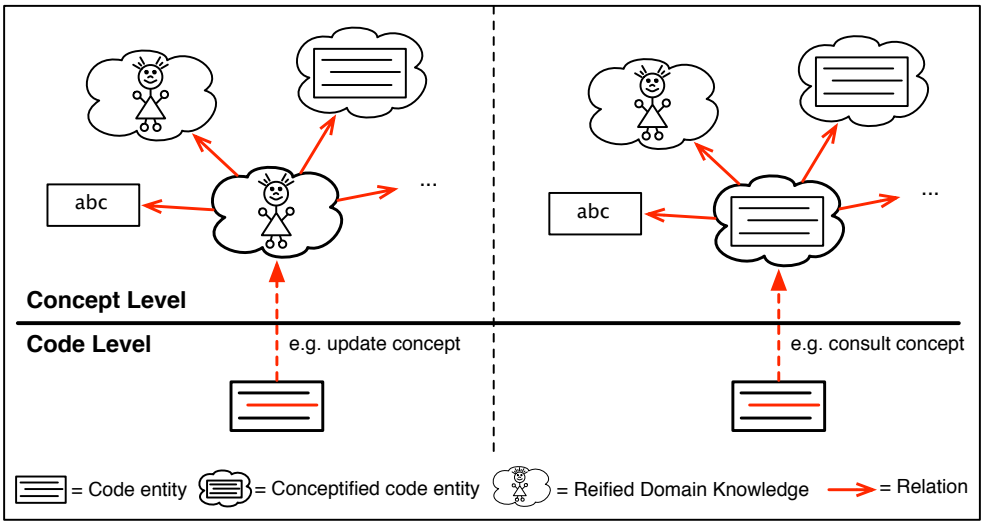



Figure 4.8: Consult, create, and update concepts from within the code level.

plicit what they know in favor of producing code. The problem associated with the genericity of the implementation was essentially about the fact that genericity results in less explicit code. As a consequence a lot of domain knowledge becomes implicit when developers strive towards malleability to help them realise their part of the adaptability challenge. So the mere existence of an explicit concept level that is coupled to the code level is not enough to eliminate the problems we discussed in section 4.1.

Therefore the concept-centric environment must facilitate the active participation of the concept level at the code level. Consequently:

- developers are stimulated to spend time at the concept level to keep it up-to-date

- developers can achieve a malleable implementation without the drawback of making knowledge implicit

The way we activate the concept level is by making it accessible from within the code level. Consequently it must be possible to write concept statements within normal code that consult, create, update or delete concept level entities. This is depicted abstractly in figure 4.8. The figure shows that both domain concepts and deified code entities can be involved in the computation at the code level. The drawing is separated in a domain concept and code entity concept part to illustrate that both 'types' of concepts can be accessed from the code level for example. For example in the left-hand side of the figure we see that an existing concept is updated from within the code (e.g., a slot is added programmatically).

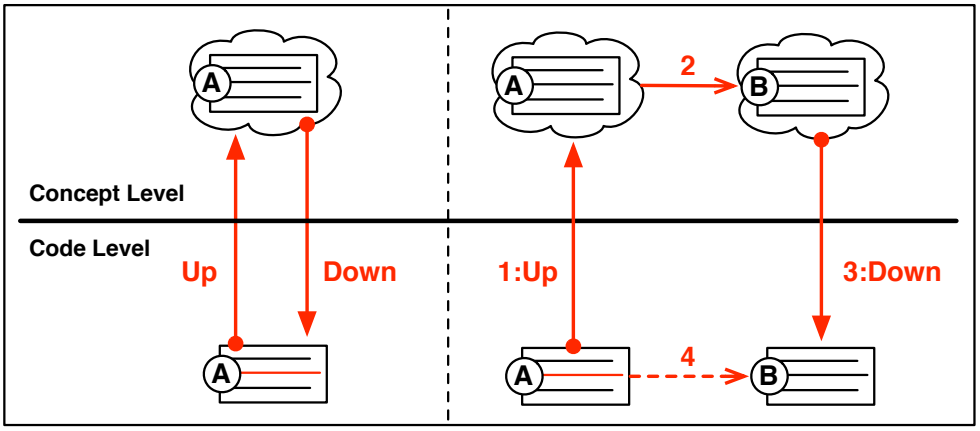



Figure 4.9: Code entities can be upped to, and downed from the concept level.

In the right-hand side we see the consultation of a deified concept (e.g., consulting a slot that points to a terminal)[7].

Besides interacting with concepts, support is also provided to trigger conceptification by writing a statement at the code level. This gives a developer the means to 'up' a code level entity to the concept level during the execution of a method. This enables a developer to write code that consults its own concept representation to perform a given task. A corresponding 'down' operation exists, which returns the code level counterpart of the conceptified entity. This is illustrated by the left-hand side of figure 4.9.

The availability of an up/down mechanism lets a developer take advantage of concept level relationships between code level entities or between code level entities and domain level entities. This bi-directional navigation is shown in the right-hand side of figure 4.9. The code entity A up's itself to the concept level (1), where it has a relation to another conceptified entity B (2). Consequently code entity A can down the conceptified entity (3) to its code level counterpart. Thus code level entity A can interact with code level entity B (4), by the aid of the relation at the concept level.

When an active concept level is installed, a developer can refactor the implementation so it becomes driven by the concept level. This is similar to the development approach followed in dynamic object models (chapter 3, section 3.2.3.6), where object types are represented as data to improve the malleability of a software system. As a consequence, the concepts play an active role in the functioning of the software system. Hence our system is better characterised as *ontology-driven* instead of *ontology-aware*. This implies that certain adaptations are realised by changing the concept level instead of the code level, hence resulting in

[7]See chapter 6 for examples of the different interactions that can take place between the code level and the concept level.

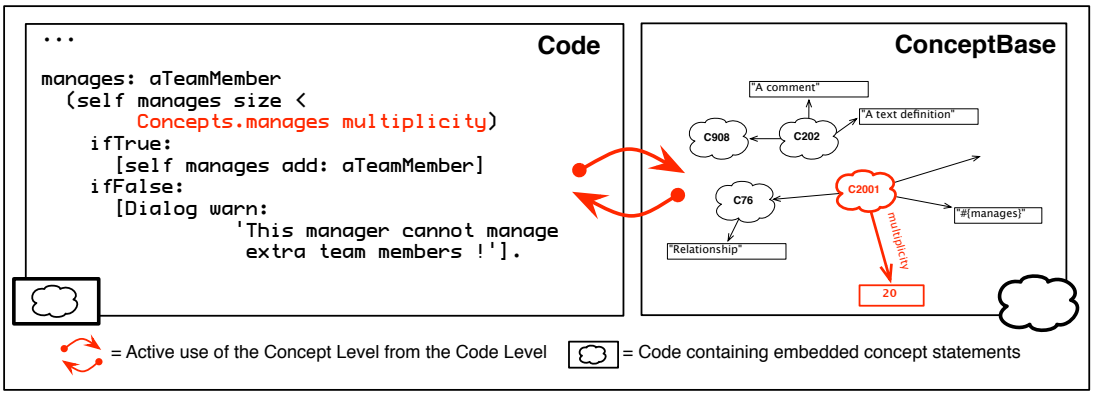



Figure 4.10: Concepts can be accessed and used actively from within the code.

run-time malleability of the software (chapter 2, section 2.1.7). Even though this results in less explicit code, the corresponding concept network will contain what is no longer visible at the code level.

We illustrate the consultation of a domain concept from within the code in figure 4.10. The figure presents the `manages:` method of a `Manager` class on the left-hand side. The `manages:` method accepts one argument, namely a team member, who is managed by a project manager. Deciding whether the project manager can manage an extra team member is done by consulting the concept level (right-hand side). In essence the code consults the multiplicity of the `manages` concept, and uses the value returned to execute the conditional statement.

The activation of the concept level is the enabler to turn the antibiosis between the concept and code level into a symbiosis. This implies that an interaction must be set up that is advantageous to both sides. So spending time on the concept level must become advantageous to the code level and vice versa. As a consequence, the benefit of maintaining an explicit concept level is no longer limited to a documentation role for the purpose of code comprehension. It plays an active role during development, which results in an interplay between code and concepts during the implementation of the software. With this we refer to an interplay where code is refactored to make use of the concept level to improve the malleability of the system. Refactoring the concept level to strengthen what can be done at the code level is part of this interplay. This installs a co-evolutionary relation between the code and the concepts.

### 4.2.4 Transparency for the Developer

We want to achieve transparency for the developer with respect to how the interaction with the concept level is perceived. This is achieved by setting up a *symbiotic integration* of the concept level and the code level.

A first step towards transparency is to use the same syntax to represent concepts as the syntax of the programming language. As a result, the developer is not required to learn another one. Moreover there is no need to switch between syntax each time a developer works on either the concepts or the code. Hence the programming experience is not disturbed by the concept language, since syntax-wise there is no visible difference between concepts or code. This is the reason why we did not use an existing ontology language such as OWL, or RDF (chapter 3, section 3.2.3.4).

A second step to obtain transparency is to ensure that interacting with concepts and objects is done by using the same mechanisms. Hence, concepts are manipulated by sending messages. This was already indicated in figure 4.10, where the `multiplicity` slot of the `manages` concept was accessed by sending the corresponding message. Moreover, in order to be able to refer to concepts, a referencing scheme is needed that is similar to the one used by the programming language (e.g. to refer to a class). This was also indicated in figure 4.10, where the `manages` concept was accessed by the reference `Concepts.manages`.

The third step to achieve transparency lies in the close integration between the programming environment and the concept environment. This means that all the tools available for interacting with concepts are closely integrated with the existing programming tools. Consequently, there is no need for a developer to continually switch between environments. Moreover it is desirable to provide an open malleable implementation of the concept environment. This enables the developer to adapt the environment to fit more closely to the particular task at hand. As a consequence, the openness of the environment contributes to the fact that developers will no longer experience the concept level as an overhead.

In the beginning of this section we referred to a symbiotic integration to achieve transparency. Similar to endosymbiosis in biology we envision a situation in which the concept level (in this case the symbiont) lives inside the programming environment (in this case the host). The symbiosis is obtained by using the same syntax and mechanisms as the programming language. It is strengthened by a close integration of the concept environment and the programming environment. In table 4.1 we summarise the main definitions that were introduced in this section.

### 4.2.5 Thesis Statement

#### From implicit towards explicit

The fact that most domain knowledge in software systems is only available in an implicit form, hampers the programmer's understanding process of the system. This means that when a programmer tries to comprehend the intended meaning of a piece of code, he will need to fall back on his own expertise or the expertise of others. Since domain knowledge forms an essential ingredient for performing

| | |
|---|---|
| *ConceptBase* | Holds the ontology for the software which is represented in a frame-based, prototype-based concept network. |
| *Domain concept reification* | The process where domain experts manually write down the relevant domain knowledge. |
| *Code level entity deification* | The process of generating a concept representation for a code level entity (a.k.a. conceptification). |
| *Concept intension* | A prescription that defines what to represent about a code level entity and how to represent it. The conceptification process will use this prescription to compute the corresponding slots at the concept level. |
| *Concept extension* | Slots that are added manually to a concept definition and that are not captured by the concept intension. |

Table 4.1: Definitions for section 4.2

software evolution, it is of the utmost importance to make it available in a more reliable form.

Consequently we foresee a ConceptBase to represent this valuable source of information in an explicit way.

### From detached towards coupled

Unfortunately, having access to an explicit representation of domain knowledge (the concept level) presents only a partial solution to the problem. A programmer still needs to map this knowledge to the implementation (the code level) in a retroactive way. This is unfortunate since there probably existed a clear link between the knowledge and the implementation when it was first conceived.

Therefore we establish a coupling between the concept level and the corresponding code level in a proactive way. This means that a developer will be able to connect the entities of both the concept level and the code level.

### From passive towards active

Since no universal function exists that can transform the concept level into the code level and vice versa, special care must be taken to make sure that both levels do not become out-of-sync. As we have seen, since the economics of performing changes at the code level will yield the best short-term benefits, programmers will tend to neglect the concept level. In general they will focus on getting the code right since this results in a consequential improvement of the overall behavior of

the application. Hence allocating a lot of effort to update this passive promise of documentation usually has a very low priority. As a consequence the concept level will quickly become out of sync with the code level. Subsequently it will be of little or no use for future evolution efforts.

We stress the importance of ensuring that the domain knowledge plays an active role. With this active role we mean that it contributes to the functionality of the software. Additionally, in the context of agile evolution, this active participation will result in an improved malleability of the implementation. This is achieved by factoring out explicit decisions in the code to the concept level. Even though this results in less explicit code, the ConceptBase holds the explicit representation of the associated concept network. Moreover this results in a co-evolutionary situation that is mutually beneficial for the concept and the code level.

### From overhead towards transparency

In order to promote making explicit what one knows, coupling it to the implementation, and using it actively from within the code, it is necessary to ensure that programmers don't perceive the concept level as an extra overhead. In section 4.1.3 we already indicated that part of the problem was a result of the antibiosis between the concept and the code level.

Therefore we set up an environment in which both levels exist in symbiosis with each other. As a consequence the use of both sides becomes as transparent as possible for the programmer. This is realised by making sure that the interaction with objects and concepts is done in a similar fashion. The transparency is further obtained by a close integration between the concept environment and the programming environment.

### Thesis Statement

This brings us to the thesis statement of this dissertation:

*A **symbiotic integration** of a programming environment with*
*an **explicit concept level** that is **coupled** to the code level*
*supports **domain knowledge documentation***
*in the context of **agile evolution***
*and amplifies the **malleability** of software through*
***active** concept level **participation***

## 4.3 Summary

In this chapter we presented the problem context, problem statement and thesis statement for this dissertation. We discussed our concept-centric environment which is based upon a symbiotic integration between the code level and the concept level.

The following requirements were set for the ontology-driven concept-centric environment:

- **Explicit concept level representation**
  The concept-centric environment must provide a mechanism that makes it possible to capture domain knowledge in an explicit form. This will reduce the possible loss of domain knowledge. The concept level is represented in a frame-based, prototype-based ontology. Concepts are defined by relating them to other concepts or to terminals in their definition frame. The corresponding concept network is stored inside a ConceptBase.

- **Coupled concept and code level**
  The concept-centric environment must provide a mechanism to couple the domain knowledge to the implementation. This will reduce the effort spent by developers on retroactively matching the knowledge to the corresponding implementation. Code level entities are deified towards the concept level by a conceptification process that follows a prescription to generate the concept representation. This is referred to as the intension of the concept, which is based on the granularity chosen for what is represented at the concept level and how it is represented.

- **Active participation of the concept level**
  The concept-centric environment must provide a way to involve the domain knowledge actively so as to provide the functionality of the software system. This will motivate developers not to neglect the domain knowledge, since it potentially yields a short-term benefit. Moreover an enhanced malleability of the software can be achieved by devoting time on the concept level. Concept statements can be written within normal code so that domain concepts and conceptified code entities can be involved in the computation at the code level. An up/down mechanism facilitates a bi-directional navigation between the code level and the concept level.

- **Transparency for the developer**
  The concept-centric environment must be built in a way that it is not perceived by developers as an overhead. This means that the interaction with the concept environment must be as transparent as possible. The syntax of the programming language is used to represent concepts at the code level. The mechanisms to interact with concepts are based on the mechanisms

to interact with objects. The programming environment and the concept environment are implemented in symbiotic integration with each other.

In the next chapter we present the proof-of-concept implementation of the concept-centric environment. The interaction with the environment is illustrated by example in chapter 6.

# Chapter 5

# The CoBro environment

In order to validate our research we created a proof-of-concept environment named the Code to Concept Browser (CoBro). CoBro is implemented in Smalltalk and uses a relational database to store the concepts that are present in the system. The main constituents of CoBro are the Concept Manipulation Language (CoBro-CML) and the Graphical Concept Network Navigator (CoBro-Nav). In addition to these, the environment contains a number of tools that support developers in constructing and exploring concept networks.

This chapter focuses on the technical underpinnings of CoBro. Its conceptual counterpart was discussed in chapter 4, whereas the actual use of the elements presented here will be illustrated with examples in chapter 6.

We start by presenting an overview of the architecture of CoBro in section 5.1. Here we also motivate the choice of Smalltalk and a relational database as implementation medium. In section 5.2 we explain CoBro-CML. The discussion focuses on the underlying mechanisms that take care of accessing and manipulating concepts. Section 5.3 focuses on the features of CoBro-Nav and discusses the way it can be extended. We also present the integration with an existing tool named the Star Browser. This browser provides an alternative way to navigate through a concept network in a tree-based fashion. The other elements of CoBro are discussed in section 5.4. This section outlines the support that is available to manipulate a concept network. We end the chapter in section 5.5 with a discussion of the default core ontology of CoBro. We also explain how it can be customised and how it is used by the CoBro environment

## 5.1 CoBro Architecture

In this section we provide an overview of the CoBro environment and its different parts. We also illustrate the interplay of the various meta and base levels that are necessary to understand how CoBro functions. In addition we motivate the choice for Smalltalk and a relational database.

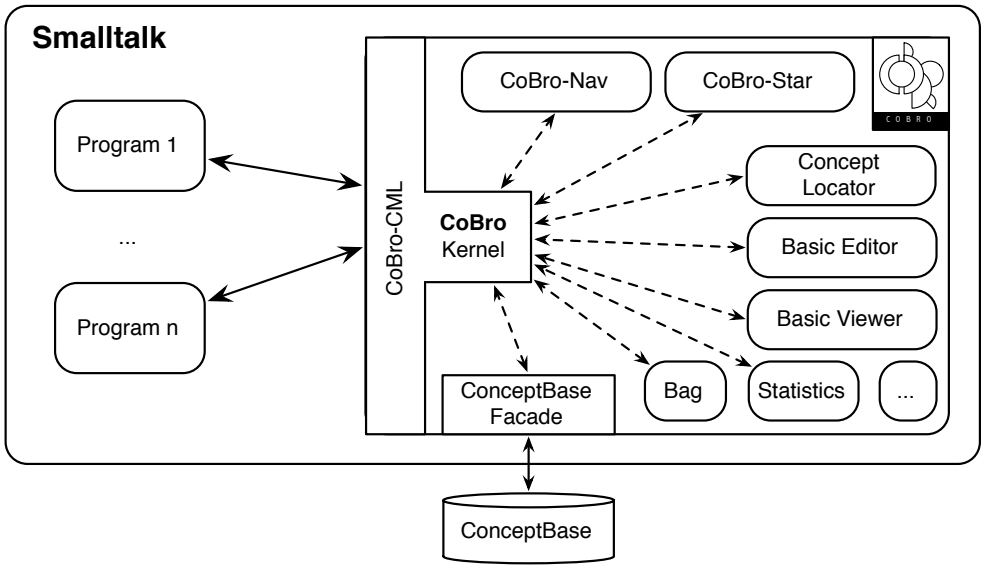



Figure 5.1: An abstract view on the CoBro environment

In chapter 4, section 4.1.4 we put forward the following requirements for a concept-centric environment:

- **Explicit concept level representation**
  The concept-centric environment must provide a mechanism that makes it possible to capture domain knowledge in an explicit form.

- **Coupled concept and code level**
  The concept-centric environment must provide a mechanism to couple the domain knowledge to the implementation.

- **Active participation of the concept level**
  The concept-centric environment must provide a way to involve the domain knowledge actively to provide the functionality of the software system.

- **Transparency for the developer**
  The concept-centric environment must be built in symbiotic integration with the programming environment.

The proof-of-concept environment we discuss in this chapter was built to meet each of the above requirements. Figure 5.1 presents an abstract overview of the CoBro environment. As is shown, developers and Smalltalk programs interact with the concept-centric environment through CoBro-CML which is part of the CoBro kernel. CoBro consists of the following elements which we will discuss in detail in the remainder of this chapter:

- **CoBro Kernel**:
  The core of the system that is closely integrated with Smalltalk for the purpose of transparency through symbiosis (section 5.2).
- **CoBro-CML**:
  All the interaction with the concept level is done through the Concept Manipulation Language (section 5.2), consequently it is an integral part of the CoBro kernel.
- **CoBro-Nav**:
  A tool that allows you to navigate and explore concept networks in a graphical way. (section 5.3).
- **CoBro-Star**:
  An extension to the standard Star Browser in which you can explore a concept network in a tree-based fashion (section 5.4).
- **Concept Locator**:
  A tool that helps you locate a concept if you do not know its reference (section 5.4).
- **Concept Bag**:
  A tool that groups a set of concepts so they can be manipulated together by other tools (section 5.4).
- **Basic Concept Viewer**:
  A tool that shows the elementary details of a concept definition (section 5.4).
- **Basic Concept Editor**:
  A tool that helps you build or manipulate a concept definition (section 5.4).
- **CoBro Statistics**:
  A basic visualisation of statistics of the ConceptBase usage (section 5.4).
- **CoBro Integration with the Smalltalk Refactoring Browser**:
  The Refactoring Browser is the main tool used to program in Smalltalk. We have closely integrated the CoBro tools within the Refactoring Browser, so they are easily accessible for the developer (section 5.4).

The CoBro environment is implemented in Smalltalk. The particular Smalltalk implementation we used is VisualWorks Smalltalk 7.3[1]. The ConceptBase that holds the concept networks is implemented as a relational database. More specifically we used the open source database PostgreSQL 8.10[2]. Since both VisualWorks and PostgreSQL are cross-platform implementations, CoBro can run on all major platforms.

[1] `http://www.cincomsmalltalk.com/`
[2] `http://www.postgresql.org/`

In the following we briefly motivate the choice of Smalltalk and a relational database as an implementation medium for CoBro.

## Motivation for Smalltalk

Smalltalk is a pure object-oriented, dynamically typed, reflective programming language. Even though it is a class-based language, the statement 'everything is an object' still holds through the existence of metaclasses. Classes, methods, and user interface specifications (to name a few) are also accessible as objects. One of the inherent properties of the Smalltalk philosophy is that everything in the environment can be adapted. This means that you can alter the behavior of the language-core or the development environment from within Smalltalk itself. Another important aspect is that you can inspect and manipulate objects in real time. This results in a highly interactive way of programming that is especially suited for agile development.

Since Smalltalk is a reflective language, it has a meta-level interface that facilitates introspection and intercession. These reflective capabilities are required to implement CoBro. There is also an interplay between the meta and base level within CoBro and within the ConceptBase. Moreover there is a similar interplay between Smalltalk, CoBro and the ConceptBase as well. It is important to distinguish between these interplays, so we present an abstract overview in figure 5.2. In the following we discuss how each interplay contributes to the realisation of CoBro.

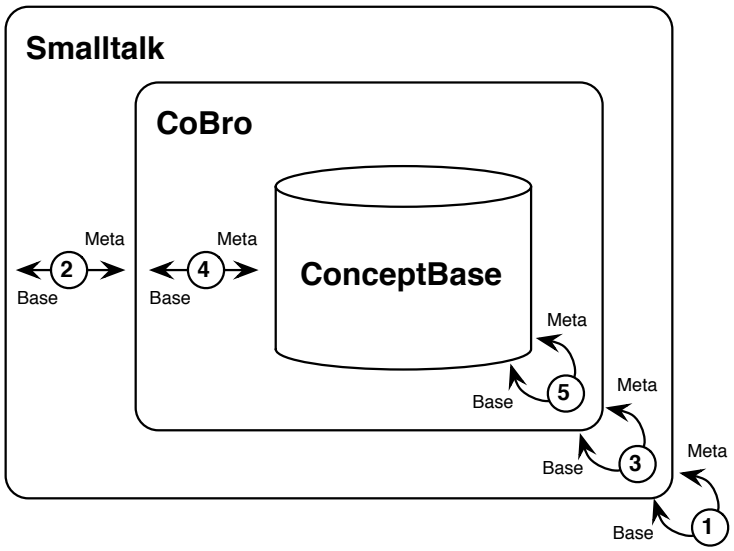



Figure 5.2: Interplay of the different meta and base levels.

**Smalltalk ↔ Smalltalk**
The first arrow in the figure represents the basic interplay that makes Smalltalk a reflective language. These reflective properties are mainly used to implement CoBro in symbiosis with Smalltalk. Together with Smalltalk's open implementation this results in a high degree of transparency.

**Smalltalk ↔ CoBro**
The second arrow represents the interplay between Smalltalk and CoBro. Since Smalltalk code entities can be 'upped' into concepts and vice versa (i.e. deification or reificiation), the CoBro system acts as a meta layer. This interplay is used to obtain the coupling of entities at the code level to entities at the concept level. Moreover it makes it possible to use concept level entities actively from within the code level. This active participation results in an improved malleability of the software. As a result of the close integration with Smalltalk, this interplay is transparent for the programmer.

**CoBro ↔ CoBro**
The third arrow indicates the interplay within CoBro itself. CoBro implements a meta layer interface through which concepts can be manipulated. This meta layer can be adapted for instance to change the way it reacts to concept slot accessing. Moreover, the meta layer of CoBro uses concepts as well to implement its behavior. This means that CoBro can be extended and adapted through the concept level, and accessing the concept level is done through CoBro. Broadly speaking, this interplay results in a more malleable implementation of CoBro.

**CoBro ↔ ConceptBase**
The fourth arrow indicates the interplay between CoBro and the ConceptBase. Since part of the behavior of CoBro depends on the active use of concept level entities, adapting the concept definitions in the ConceptBase has a direct impact on the way CoBro behaves. a core ontology from the ConceptBase. This interplay improves the malleability of CoBro.

**ConceptBase ↔ ConceptBase**
The fifth and final arrow represents the interplay within the conceptbase itself. This interplay is different from the previous ones in the sense that it concerns the interplay between model-entities. More specifically it refers to the fact that concepts from the base level are described in terms of concepts from the meta level. Such a division of the ConceptBase is virtual and, depending on the view of the current use, the concepts can play different roles (chapter 3, section 3.2.2). The main benefit of this interplay is that the concept level is self-describing.

Note that CoBro can be implemented in any language that has reflective capabilities similar to Smalltalk. Preferrably the language environment has an

open implementation so that it can be extended with CoBro specific elements. In essence implementing the CoBro environment was done by using the following properties of Smalltalk:

- **Introspection**
  The *conceptification process* (section 5.2.3) of code level entities requires a mechanism to inspect the representation of these entities. Broadly speaking, the intension for a conceptified code entity is assembled by consulting the Smalltalk environment. The *granularity* of the conceptification process is constrained by the code entities that can be reified. For instance if methods cannot be reified then it is not possible to conceptify them.

- **Intercession**
  Implementing CoBro in symbiosis with Smalltalk requires a way to adapt the default *binding lookup* (section 5.2.1) and *method lookup* (section 5.2.4) mechanisms. Intercession is also required to make it possible to adapt the CoBro behavior at runtime by changing the concept network (section 5.3).

- **Open implementation**
  The symbiotic integration we envision requires an open implementation. Also a close integration of the CoBro tools with the programming environment is needed for a non-obstrusive synergy between the concept level and the code level (section 5.4). This enables a developer to work with concepts and objects in a transparent way.

## Motivation for a relational database

The fact that we use a relational database to store the ConceptBase is transparent to the user of the system. We also took special care to make it possible to switch the storage medium without affecting the surrounding tools. The database schema we implemented stores the concept network as a graph. The main reasons why we opted for a relational database are:

- **Sharing**
  An important aspect of an ontology is that it is a shared representation of a conceptualisation (chapter 3, section 3.2). Since the sharing aspect of relational databases is evident, this was a point in favor. Moreover if the ConceptBase is used by different developers, it could be installed at a single point of access.

- **Relationships**
  A prime constituent of a concept network are relations (chapter 4, section 4.2.1). In essence a concept is defined by relating it to other concepts or terminal values. Consequently the choice of a relational database was quite evident.

- **Querying**
  One of the main operations on a concept network is querying. Concepts are often not only located by their reference, but also through the values that are related to them.

- **Volume and Persistency**
  It should be possible to store a large volume of concepts persistently.

## 5.2 Concept Manipulation Language

CoBro-CML is the language we created to manipulate concepts in the CoBro environment. The language is an extension of the base language Smalltalk. Consequently it uses the same syntax and uses message sending as the prime way to manipulate concepts in the ConceptBase. It can be used to create, modify, delete, and query concept definitions. Similarly to Smalltalk it contains a number of inspector-alike utilities that make it easier for the programmer to navigate through the concept network.

Through CoBro-CML it is possible to couple code level entities to concept level entities in a transparent way. This transparency is achieved by the close symbiotic integration with the base language. Hence the ConceptBase can be manipulated from within existing Smalltalk programs as well. This makes it possible to use concept definition slots in an active way to ensure the behavior of the software. Since part of the behavior can be factored out to the ConceptBase, this results in an improved malleability of the software. This is mainly due to the fact that changes to the software can be made at the concept level instead of at the code level.

In the following subsections we discuss CoBro-CML in detail. We start by explaining the integration with the standard Smalltalk namespace mechanism. Subsequently we discuss how concepts can be created and how code level entities are conceptified. Next we focus on the manipulation of concepts by slot accessors and setters. Also the lookup of slots through the parent chain is described.

### 5.2.1 Concept Namespace

Namespaces are used in Visualworks Smalltalk to disambiguate names. They provide a context where the object to which the name refers can be uniquely resolved. The standard Smalltalk namespace starts at the `Root` namespace, and stores the classes that make up the base system in the `Root.Smalltalk` namespace.

The dotted notation is used to describe a path to a binding for a particular name in the system. For example the full path to the `OrderedCollection` class is `Root.Smalltalk.Core`. This path combined with the name of the class gives a *fully qualified reference* (e.g., `Root.Smalltalk.Core.OrderedCollection`) that is unique in the system.

A namespace definition can import bindings of other namespaces. Doing so allows the programmer to refer to them as if they reside in the local namespace (e.g., `OrderedCollection` instead of `Root.Smalltalk.Core.OrderedCollection`).

To be able to refer to concepts we have created a dedicated `Concepts` namespace (e.g., `Root.Smalltalk.CoBro.Concepts`). Through this namespace, you can directly access concepts that are stored in the ConceptBase. As a result of namespace imports, the programmer can refer to concepts simply by using the prefix `Concepts`[3]. For example the binding for the `Wine` domain concept is looked up by evaluating the statement `Concepts.Wine`.

As we have explained in chapter 4, implementation entities have a conceptified counterpart. This is needed so that domain knowledge described as concepts can be hooked up to the implementation. The `Concepts` namespace allows us to distinguish between references to the implementation entity and references to the conceptified version. For example, if we want to access the concept associated to the `OrderedCollection` class, we can refer to it as `Concepts.OrderedCollection`.

### Extension of the Binding Lookup Mechanism

As we have described in section 5.1, concepts are not stored in the Smalltalk image but in an external relational database. Hence we also extended how Smalltalk resolves references into bindings. In figure 5.3 we present a flowchart that outlines this mechanism.

If the path of the reference does not contain the `Concepts` namespace then the reference refers to an implementation entity. Consequently the binding is resolved using the standard Smalltalk lookup mechanism. If the path does contain the `Concepts` namespace, then our extended lookup mechanism resolves the binding. There are two possibilities:

1. the reference refers to a domain concept, or
   (e.g., `Concepts.Wine`)

2. the reference refers to a conceptified version of an implementation entity
   (e.g., `Concepts.OrderedCollection`).

In the first case, we can directly retrieve the binding for the reference from the ConceptBase. The concept reference (i.e. the part of the reference after `Concepts`) acts as a fully qualified reference in the ConceptBase.

For the second case, we must first retrieve the fully qualified reference of the implementation entity from the Smalltalk image. This is necessary since, as a result of namespace imports, programmers are not required to provide the full path to an entity. Moreover, to avoid a naming conflict between normal concepts

[3]Since namespaces are dictionaries, an alternative way to access its contents is by sending the `at:` message (e.g., `Concepts at:#{Wine}`).

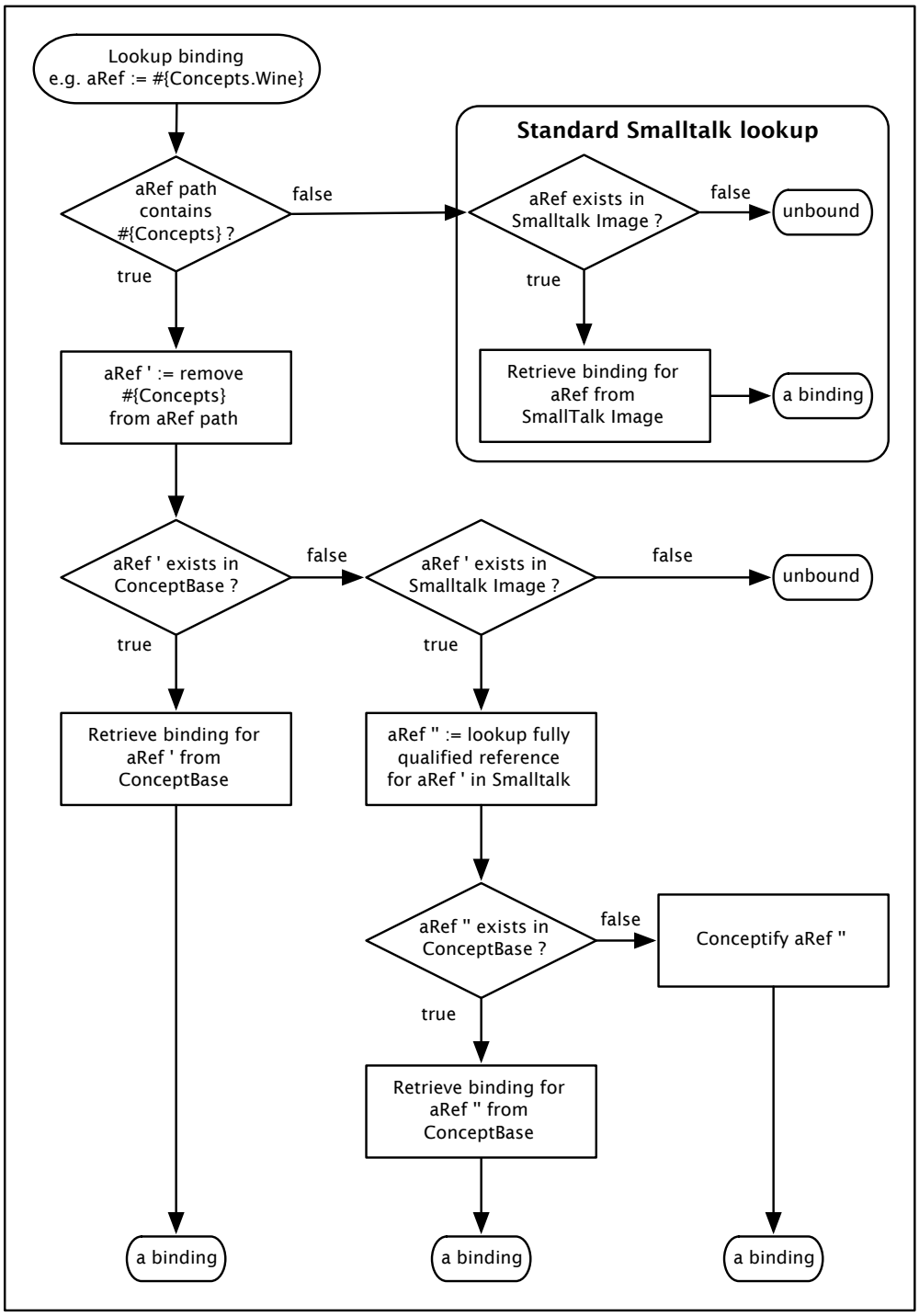



Figure 5.3: The CoBro extension of the standard Smalltalk binding lookup mechanism.

and implementation concepts, concepts that are the result of conceptification are stored in the ConceptBase using this fully qualified reference. For example the concept `Concepts.OrderedCollection` is stored with fully qualified reference '`Root.Smalltalk.Core.OrderedCollection`'. Note that if a domain concept is created manually with this exact reference, it will be automatically promoted to a conceptified code entity.

If the reference does not already exist in the ConceptBase, then we automatically generate a conceptified version of the implementation entity. This conceptification process is described in section 5.2.3.

### 5.2.2 Concept Creation

The creation of a concept can be done either:

- ex nihilo, or
- by using an existing concept definition, or
- by automatic conceptification, or
- by using the basic concept editor.

Ex nihilo creation of a concept corresponds to creating the concept's definition 'out of nothing'. It can be done by sending a `new:` message to the `Concepts` namespace with the fully qualified reference for the concept as argument. A corresponding `delete:` message exists for removing concepts from the ConceptBase. Note that CoBro contains a mnemonic generator to ensure the uniqueness of fully qualified references.

You can also create a concept by reusing the definition of an existing one. By sending the message `smalltalkDefinition` to an existing concept, you get a generated textual definition of that concept in CoBro-CML. By editing the values in this template you can create a concept that is similar to the existing one.

Creating a concept by conceptification is done by prefixing the reference (either fully qualified or not) of the code entity with '`Concepts.`'. How this reference is resolved was explained in the previous section.

Also the basic concept editor can be used to create a new or update an existing concept. This is explained in section 5.4.

Before we explain the conceptification process we show examples of the four ways to create a concept in CoBro-CML.

```
(Concepts new: #{HelloWorld})
       hasPreferredLabel: 'Hello World Concept' ;
       superconcept: Concepts.Concept ;
       comment: 'Testing concept creation' ;
       save.
```
⟶ ***ex nihilo*** *creation of a concept*

```
Concepts.HelloWorld smalltalkDefinition
```
⟶ *generates a template based on the* ***existing concept definition*** *of* `HelloWorld` *(i.e. the statements of the ex nihilo creation)*

```
Concepts.Root.Smalltalk.Core.OrderedCollection.
```
⟶ *returns the existing concept or* ***automatically conceptifies*** *the class* `OrderedCollection`

```
ConceptEditor open.
```
⟶ *Opens the* ***basic concept editor*** *to create a concept*

Note that a concept is only stored in the conceptbase if you explicitly tell the system to do so. This is done by sending the message `save` to the concept. We discuss concept slot manipulation, which allows you to add entries to a concept definition, in section 5.2.4. Depending on the core-ontology that is loaded, certain slots are mandatory when creating a concept. We discuss the default core ontology and its underpinnings in section 5.5. For a detailed discussion with examples we refer to chapter 6.

### 5.2.3 Conceptification Process

Conceptification takes place if you try to access a reference in the ConceptBase that corresponds to a code level entity. If conceptification is triggered then the resulting concept receives the fully qualified reference of the code level entity. As we have explained in the previous section, concepts are accessed through the `Concepts` namespace. This namespace is connected to a concept factory that takes care of building the concepts. The use of a factory makes it possible to add several mechanisms to construct concepts in a way that is transparent for the `Concepts` namespace. There are currently two factories active in the implementation of CoBro, one that takes care of domain concepts, and one that handles the conceptification of code level entities.

A definition of a conceptified code entity contains two parts: an extension and an intension. The notion of extensions and intensions of a concept were introduced in chapter 3, section 3.2.3.2. In summary, the extension part lists all the slot

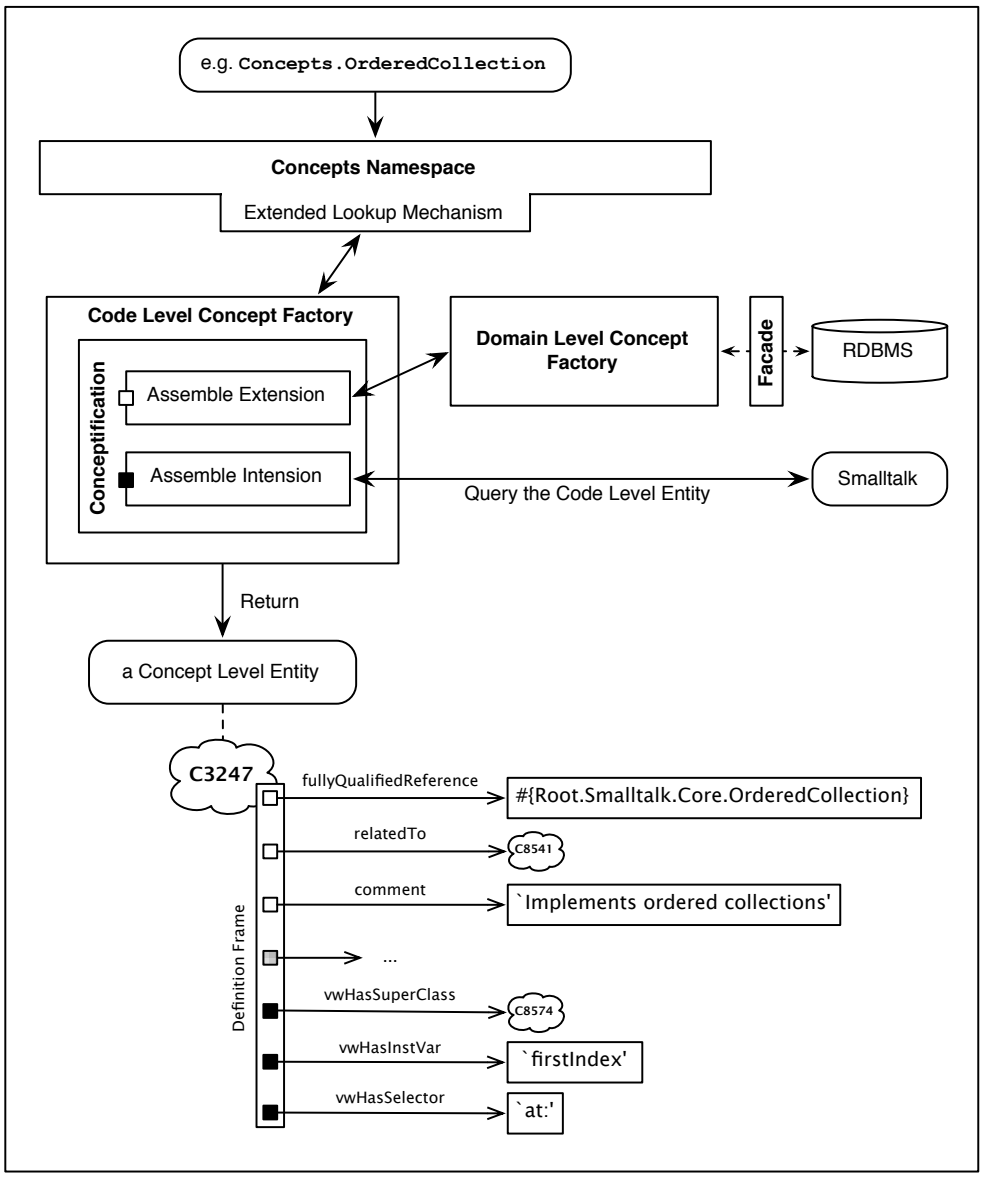



Figure 5.4: The conceptification process of code level entities

entries that belong to a concept. The intension part provides a prescription that is used to compute the slot entries.

The intensional part of a concept definition takes care of automatically transforming information about the code level entity into relevant slots at the concept level. As we have seen in chapter 4, a completely automatic mapping process between the code level and the concept level is unfeasible. This is mainly due to the inherent nature of domain knowledge as was described in chapter 3.

In figure 5.4 we illustrate how a code level entity is conceptified. The ex-

tended lookup mechanism will delegate the retrieval of a concept to either the code level concept-factory or the domain level concept-factory. In the case of the `Concepts.OrderedCollection` retrieval, it is resolved by the code level concept-factory. Assembling the definition consists of bringing together the extension (stored in the ConceptBase) and the intension (computed) of the concept. Computing the intension is done by querying the Smalltalk image through its meta level interface. Note that we use the convention to prefix the names of relations that point to a computed result with the letters '`vw`'.

A concept that is the result from conceptification is only made persistent in the ConceptBase if:

- an extensional definition part was defined, or
- another concept refers to the conceptified code level entity, or
- an explicit `save` message was send.

By default only the extension of a concept is stored in the ConceptBase. Hence the slots that are computed by interpreting the intension of the concept are not stored in the ConceptBase. Working with domain concepts or concepts that are the result of conceptification, is transparent to the programmer.

### Granularity of Conceptification

The first decision related to granularity is deciding *what* to represent at the concept level. Computing the intension of a concept requires access to the meta level of the base language. Introspection is used to get all the necessary information about the code level entity. For instance in order to compute the slots for the `OrderedCollection` class, we can ask the base language for its superclass, instance variables, method names, method bodies, or user interface specification. Note that the default conceptification process in the code level concept factory can be adapted so that a different intension is computed.

The next decision is *how* to represent these elements at the concept level. As we have discussed in chapter 4, entities in a concept network are either concepts or terminals. The main difference between both was that a concept definition contains slot entries, and a terminal definition contains a value. Hence only concepts can be related to concepts or terminals in the concept network. A terminal represents an element about which we do not want to know any extra information beyond its value.

The choice of whether to define a code element as a concept or as a terminal has to do with the ontological granularity that is required (see chapter 3) by the application for which CoBro is used.

The default granularity for conceptifying code entities in CoBro is at the class level. However, CoBro is implemented in a way that allows the adaptation

of this granularity. An example of such an adaptation would be to represent the methods of a class as concepts, which are currently represented as terminals by the default process. This would make it possible to couple domain knowledge to these conceptified methods as well. The changes that are required for this boil down to:

- setting up a fully qualified reference scheme that can be used to uniquely identify these entities, both at the code level and at the concept level, and
- adapting the conceptification process of the code level concept-factory so that it knows how to compute the intension.

### 5.2.4 Concept and Relation Manipulation

Similarly to manipulating objects in Smalltalk, concepts are manipulated through message sends. Hence accessing a slot of the definition frame of a concept is done by sending a message. The name of the message corresponds to the fully qualified reference of the relation that is used in the slot[4]. If the slot does not exist and it cannot be resolved through the concept parent chain then an exception is raised. The extended message lookup mechanism that takes care of this process is explained later in this section.

*Unary messages* in Smalltalk are messages with no arguments (e.g., `List new`). These are used in CoBro-CML to access the values of a slot. *Keyword messages* are messages with arguments (e.g., `myList add: 'Hello'`). These are used to add a slot to a definition frame. The message names in CoBro-CML correspond to the fully qualified reference of the relation. The relation reference can be used readily as a unary message to access slot values. If the reference is appended with a colon then it can be used as a keyword message to set the value for a slot (the value is an argument of the message).

Since we use the same message sending mechanism as Smalltalk, it is also possible to cascade messages. This is used to send two or more messages to the same concept, and is done by separating the messages with a semicolon. This way it is possible to avoid repeating the concept name for each message.

Relations themselves are concepts. So before a relation can be used, a corresponding concept should be created. We will first summarise the prerequisite slots, after which we will illustrate these with an example. The default *required* slots for a concept are:

- Exactly one `fullyQualifiedReference` slot
  *This is used to uniquely identify a concept.*

[4]In Smalltalk a message name may contain letters, numbers, and underscores. It may not begin with a number. Therefore the same restrictions apply to the fully qualified references of relations. We follow the same naming convention as Smalltalk, hence the references to concepts begin with an uppercase letter, and the references to relations begin with a lowercase letter.

- exactly one `hasPreferredLabel` slot
  *This is used for printing the concept.*

- one or more `superconcept` slots
  This is used to connect a concept to a superconcept in order to set up a parent chain.

- the `Concepts.Concept` concept in its parent chain
  This core concept indicates to CoBro that the current concept should be treated as a `Concept`. In addition it provides a number of default slots that are necessary for the functioning of the CoBro tools.

For a relation these requirements are extended with:

- exactly one `multiplicity` slot
  This is used to specify the multiplicity of the relationship.

- exactly one `allowedDestinations` slot
  This is used to specify the allowed destinations of the relationship.

- the `Concepts.Relationship` concept in its parent chain
  This core concept indicates to CoBro that the current concept should be treated as a `Relationship`. In addition it provides a number of default slots that are necessary for the functioning of the CoBro tools.

The `multiplicity` slot refers to the number of slots of this type that can be added to a concept. The `allowedDestinations` slot holds a collection of fully qualified references. These constrain the possible destinations of the relation. For instance the definition of the `comment` relation is:

```
(Concepts new:  #{comment})
       hasPreferredLabel:  'comment';
       superconcept:  Concepts.Relationship ;
       multiplicity:  '#(1 #n)' ;
       allowedDestinations:  '#(#{String})'.
```

Note that the argument of the `new:` message is the value of the fully qualified reference slot in the concept definition frame. In the example the destinations for a `comment` slot are restricted to strings (a terminal). We come back to this in section 5.5.

Depending on the characteristics of the relation, sending a unary message will either return a single value or a collection of values. For example the multiplicity of `hasPreferredLabel` is one-to-one, and the multiplicity of `comment` is one-to-many. This is shown by the following examples (see chapter 6 for more examples):

```
Concepts.HelloWorld comment.
```
⟶ *returns an ordered collection containing the comment slots for the* `HelloWorld` *concept:* `#('Testing concept creation')`

```
Concepts.HelloWorld comment: 'Test adding a slot'.
```
⟶ *adds an extra* `comment` *slot to the* `HelloWorld` *concept*

```
Concepts.HelloWorld hasPreferredLabel.
```
⟶ *returns a single value:* `'Hello World Concept'`

```
Concepts.HelloWorld hasPreferredLabel: 'Goodbye World'.
```
⟶ *the multiplicity of* `hasPreferredLabel` *is one-to-one, a slot already exists, so this throws an exception*

### Extension of the Message Lookup Mechanism

To be able to manipulate concepts with message sends (for which the method bodies are not defined in the class of the receiver), we have overridden the default Smalltalk message lookup mechanism for concepts. When a message in Smalltalk is sent, the corresponding method is located in the inheritance chain of the class of the receiver. If the method is found in the class of the receiver, it is executed. Otherwise the lookup will continue with the superclasses. If the method is not found and the root of the class hierarchy is reached, the `doesNotUnderstand:` message is sent to the receiver together with the initial message. The default behavior of the `doesNotUnderstand` message is to throw an exception. In figure 5.5 we present a flowchart that outlines our mechanism.

Basically three possibilities exist:

- the corresponding method is found in the Smalltalk class hierarchy
  e.g., `Concepts.HelloWorld extendedDisplayString`
  → ' (C6352) Hello World Concept'

- the corresponding method is not found in the Smalltalk class hierarchy, but it can be found as a relation in the ConceptBase
  e.g., `Concepts.HelloWorld hasPreferredLabel`
  → 'Hello World Concept'

- the corresponding method is not found in Smalltalk and in the ConceptBase
  e.g., `Concepts.HelloWorld unknownMessage`
  → exception is raised by doesNotUnderstand

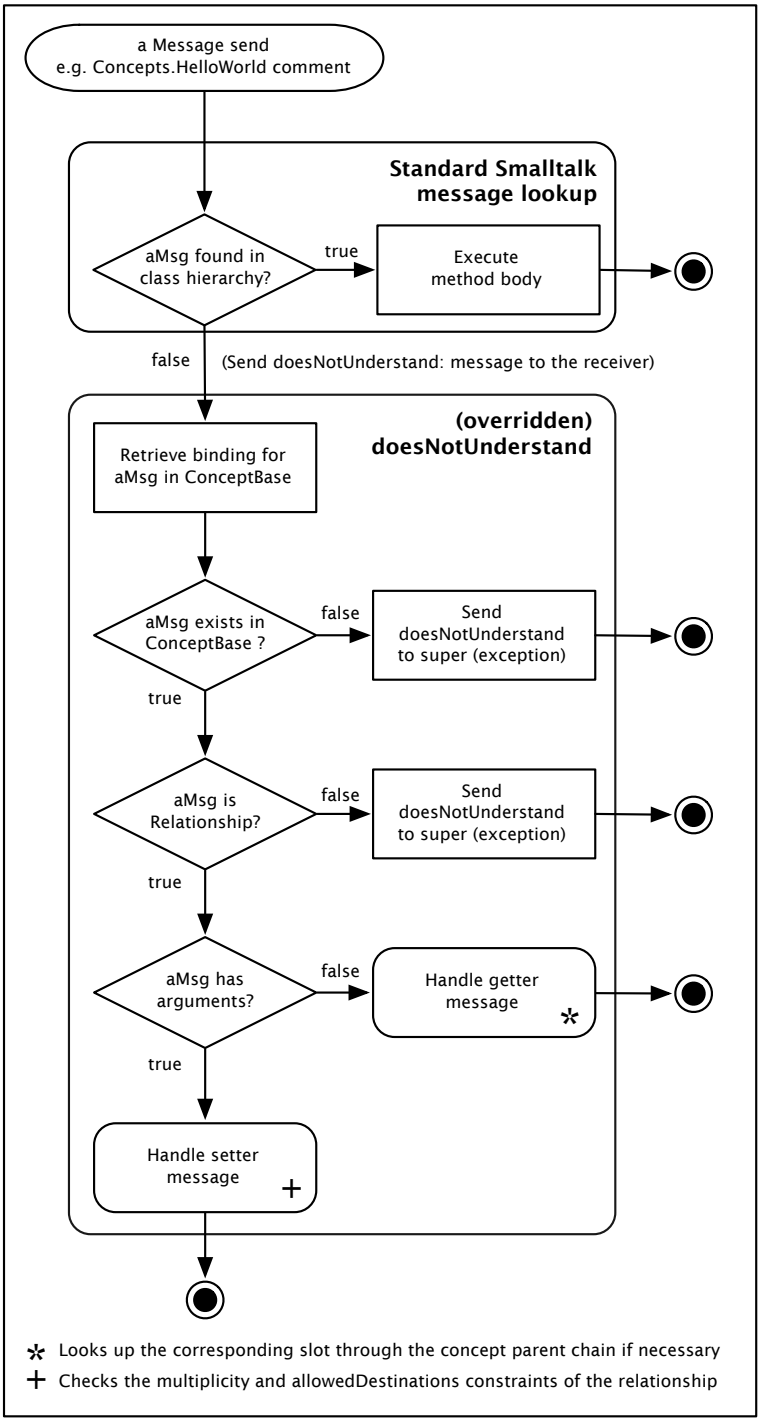



Figure 5.5: The CoBro extension of the standard Smalltalk message lookup mechanism.

To override the default handling of slot messages, you can add a method with the slot name in the *class* hierarchy (more specifically in the meta level interface of the `Concept` class).

If the relation that corresponds to the message is found, it is either handled as a setter (keyword message) or a getter (unary message) (figure 5.5). In the case of a setter, the characteristics of the relation are consulted to verify if the requested operation is permitted or not (i.e. `multiplicity` and `allowedDestinations`). In the case of a getter, the value for the slot is retrieved from the conceptbase. Before the value of the slot is returned, a value interpreter is applied to it. A value interpreter is a smalltalk block that is stored in the conceptbase. The default value interpreter simply returns the value stored. Value interpreters are explained in chapter 7, section 7.1.6.

In the following we describe how a slot is retrieved through the concept parent chain if it does not exist in the receiver definition frame.

### Lookup of Slots in the Concept Parent Chain

With the default core ontology installed, multiple `superconcept` slots can be added to the definition of a concept. A `superconcept` slot refers to a parent concept of the slot holder. Currently we have implemented a breadth-first lookup mechanism that traverses the parent chain if a slot is not found in the current concept. The choice of implementing a breadth-first lookup is a modeling decision. This lookup mechanism can easily be adapted in the meta level interface of the CoBro kernel. The reason for its existence is that the implementation of a number of CoBro tools depends on the availability of a number of concept slots. Certain information is identical for all the concepts, so it is stored only at the topmost level (and hence acts as a default slot). In section 5.3 we will show the use of such information in the implementation of CoBro-Nav. In figure 5.6 we illustrate the lookup mechanism.

As is shown, when a slot does not exist in the concept to which the message was sent, the slot lookup mechanism climbs up one level. At this level the slot is searched in the immediate parents of the concept from left to right. As a consequence the order of the parent slots in the definition frame of a concept is important. If the slot is not found on this level, the lookup continues at the next level. If the slot name is not found in the parent chain a `doesNotUnderstand` message is sent to the super class of the receiver's class (by default this throws an exception).

A number of helper messages for accessing the parent chain are provided in the meta level interface of a concept. The first two are `super` and `super:` which allow us to select either the first or a specific parent slot. The messages `allSuperconcepts` and `withAllSuperconcepts` facilitate collecting all the parent slots of a concept. Accessing the children that refer to a concept as their parent is done by sending the message `allSubconcepts`, or `withAllSubconcepts`. In

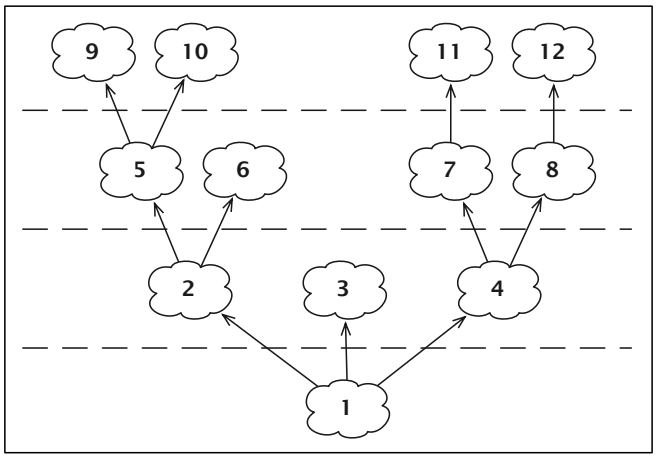



Figure 5.6: Default breadth-first lookup of slots in the concept parent chain.

chapter 6 section 6.1.4 we illustrate the use of these messages.

### Coupling Domain Level and Code Level Concepts

Using the mechanisms we described, it is possible to connect domain level concepts and code level concepts in a straightforward way. Connecting concepts is done by relating them to each other. This means adding a slot to a concept that relates it to another concept. The following possibilities exist:

- Domain level concept → Domain level concept
  e.g., `Concepts.HelloWorld isRelatedTo: Concepts.HelloStreet.`
- Domain level concept → Code level concept
  e.g., `Concepts.HelloWorld isRelatedTo: Concepts.OrderedCollection.`
- Code level concept → Domain level concept
  e.g., `Concepts.OrderedCollection isRelatedTo: Concepts.HelloStreet.`
- Code level concept → Code level concept
  e.g., `Concepts.List isRelatedTo: Concepts.OrderedCollection.`

The examples above require the existence of an `isRelatedTo:` relation with allowed destination `#(#{Concept})`. `HelloWorld` and `HelloStreet` are domain concepts. `OrderedCollection` and `List` are Smalltalk classes.

Since conceptification of code level concepts is transparent for the user, all these combinations are simply done by sending a keyword slot message. Note that besides relations between concepts, both domain level concepts and code level concepts can be related to terminal values as well (e.g., with a `comment` relation).

#### Active Use of Concepts in the Implementation

As a result of the symbiotic integration of CoBro within Smalltalk it is possible to write CoBro-CML statements everywhere in the programming environment where Smalltalk statements are allowed. Consequently you can write a method that relies on concepts to provide its behavior. This is not limited to consulting the ConceptBase, it is also possible to manipulate the concepts (e.g., creation of new slots) from within Smalltalk methods.

Part of the CoBro implementation relies on the existence of the concept level to provide its functionality. Consequently one could say that part of the behavior is pushed towards the concept level. This is similar to Dynamic Object Models (discussed in chapter 2) where part of the behavior is pushed towards the data. The main benefit obtained is a more malleable implementation of CoBro. This is because changes to the system can be done by manipulating concepts instead of adapting code.

In the following section we introduce the graphical concept network navigator. The implementation of the network navigator makes use of concepts. Besides describing its main characteristics, we will also highlight how it can be adapted through changes at the concept level.

## 5.3 Graphical Concept Network Navigator

Programmatic access to a concept network is beneficial for several reasons. Yet a graphical approach to navigate a concept network is more intuitive for exploration. This is why we have created the graphical concept network navigator (CoBro-Nav). The tool is based upon the Hotdraw framework [Bra95]. Hotdraw is a popular structured drawing framework that can be used to build graphical editors.

In this section we describe the main characteristics of CoBro-Nav and how it can be extended to support different types of nodes. The implementation of CoBro-Nav makes use of CoBro. As a consequence certain adaptations can be done at the concept level. Since visualisation is not the prime objective of this dissertation we will only briefly summarise its elements.

### Overview of Functionality

CoBro-Nav facilitates graphical browsing of the concept network stored in the ConceptBase. It can be opened on a collection of concepts, and starting from these you can navigate to the related concepts and terminals. In order to improve the interaction with the concept network, the nodes that represent terminals and concepts are built up as user interface components. Consequently manipulating nodes can be done using the same mechanisms used in standard user interfaces. Currently CoBro-Nav is only used to provide a view on the entities in the

ConceptBase. The main window through which all interaction is performed is shown in figure 5.7.

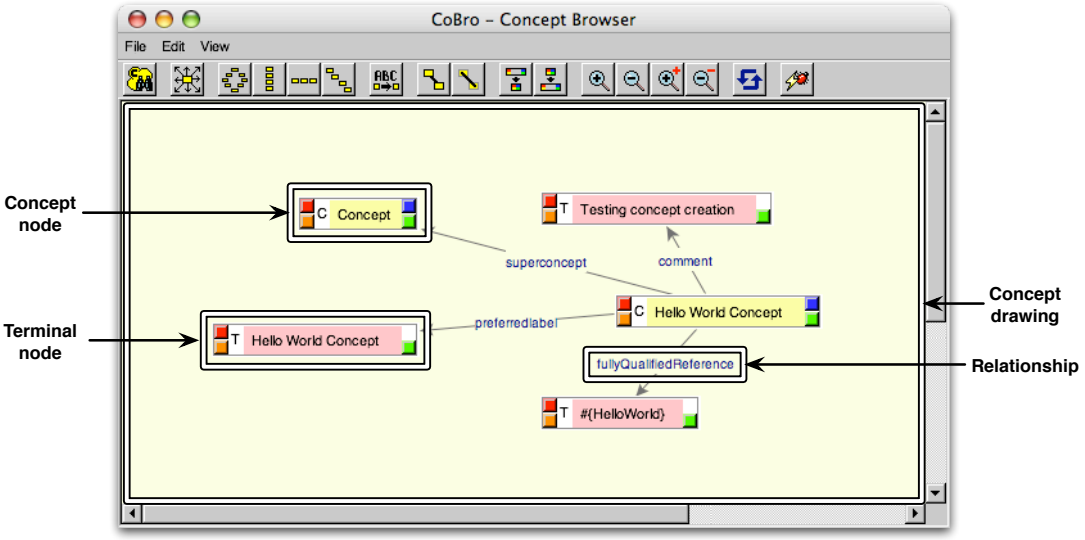



Figure 5.7: The CoBro-Nav graphical concept browser.

Analogously to a concept network, there are two types of nodes in CoBro-Nav: concepts and terminals. The main difference is that terminals do not have a definition frame that can be exploded. As described later in this section, it is possible to add different visualisations for both types. This is extremely useful for terminal nodes. In figure 5.8 we show the two main types of nodes and their minimised and expanded forms.

Each node in a concept drawing has a number of quick-action buttons, one on each corner. The corresponding actions are:

- Red (top-left): **Remove**
  Removes the node from the drawing. Note that the corresponding entity in the ConceptBase remains unaffected.

- Blue (top-right): **Explode**
  Explodes the definition frame of the concept node. Exploding a node means that for each slot, an arrow is drawn to the corresponding node. If the node is already shown in the drawing then the arrow is directed towards it. Otherwise a new node is added to the drawing.

- Orange (bottom-left): **Minimise / maximise**
  Minimises or maximises a node. This is useful if a large volume of concepts is shown and you want to make the drawing more concise.

- Green (bottom-right): **Expand / collapse**
  Expands a node to show its definition frame. For a concept node, all the slots of its definition are shown inside the node. Double-clicking a slot allows a selective explode (i.e. an arrow will be added to the drawing for the selected slot). For a terminal node, the corresponding value is shown. Depending on the type of terminal, this value is visualised differently (see later).

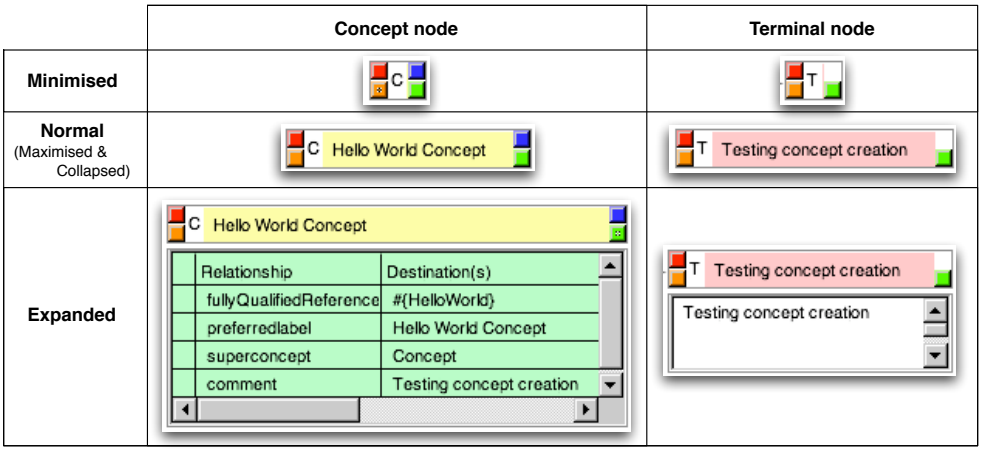



Figure 5.8: The available CoBro-Nav node types.

Different node layout strategies have been included in CoBro-Nav. It is possible to rearrange all the nodes in the drawing or a selection of nodes. Currently it supports circular, vertical, horizontal, and diagonal layout.

In figure 5.9 we present an overview of the toolbar and the corresponding operations. These operations are accessible through several context-sensitive menus as well. Depending on whether nodes in the drawing are selected, they will apply to all the nodes or to the selection.

Interaction with relations is done by opening a context-sensitive menu on the corresponding arrow. This way you can add the corresponding relation concept as a node to the drawing for further inspection. Other possible operations include browsing concepts that use the relation or that refer to the relation, for instance. Also the visualisation properties, such as line thickness and color of the arrow, can be manipulated. Note that there is support for visualising circular relations as well.

The concept drawings can be saved as a postscript file or exported to GraphViz format[5]. The latter is open source graph visualisation software that takes graph descriptions in a simple text-based format, and renders the graph into an optimal layout. We show the generated output for the `HelloWorld` example in

[5] `http://www.graphviz.org/`

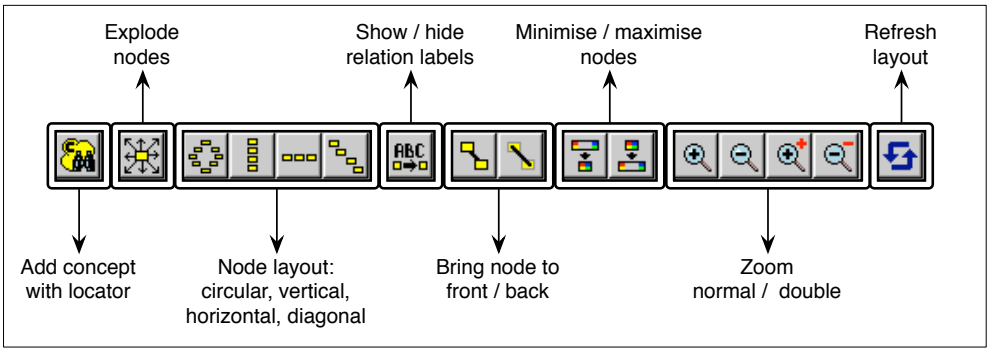



Figure 5.9: The CoBro-Nav toolbar

figure 5.10 (concepts are drawn as double-line boxes, terminals are drawn as single-line boxes).

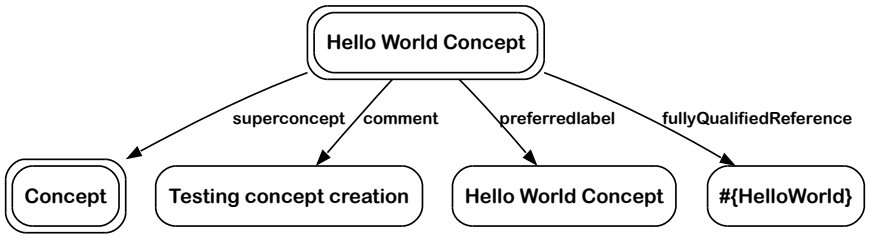



Figure 5.10: CoBro-Nav export to GraphViz.

We have also integrated CoBro-Nav in the Star Browser tool of Roel Wuyts[6]. In this way a concept network can be traversed in a tree-based fashion. One of the goals of the Star Browser is to allow developers to browse the Smalltalk environment and classify code level entities while browsing. Classification can be done by dragging and dropping the entities (e.g., classes, methods, ...) inside an extensional classification. The extension we made also supports classification of concepts in this way. The screenshot in figure 5.11 shows the tree-based navigation in Star Browser combined with the network-based navigation of CoBro-Nav.

## Extensibility of CoBro-Nav

In this section we will briefly touch the extensibility of CoBro-Nav. A comprehensive discussion can be found in chapter 7, section 7.1.5.

It is possible to extend CoBro-Nav with different visualisations of nodes. One of the experiments we performed with CoBro was in the context of separating user interface concerns [GD04, GD05]. For this purpose it was necessary

[6] `http://homepages.ulb.ac.be/~rowuyts/StarBrowser/`

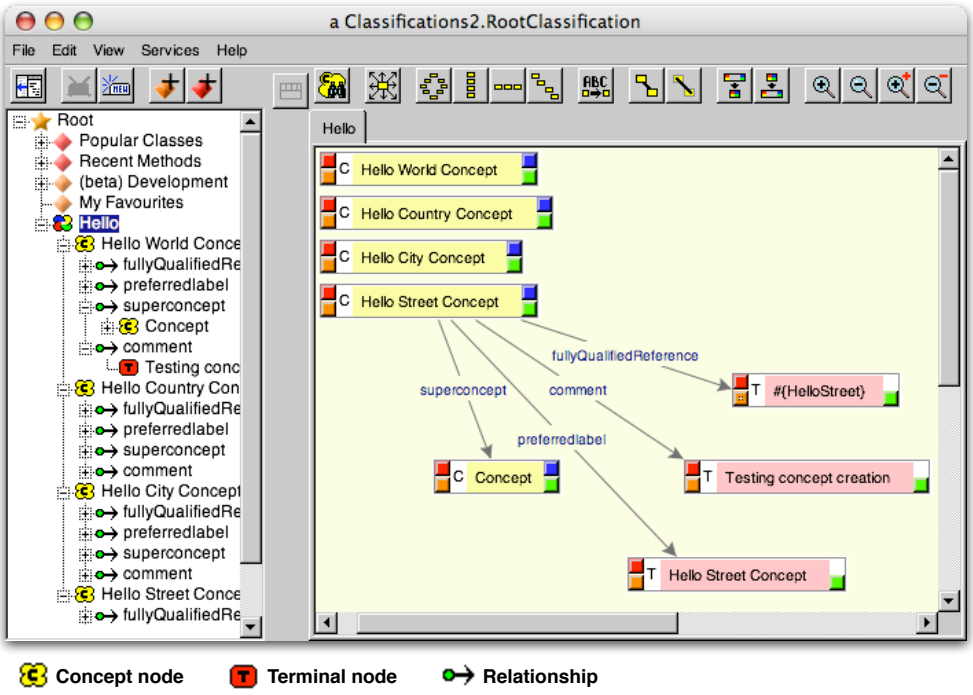



Figure 5.11: The extended Star Browser.

to be able to relate concepts to user interface specifications (a.k.a. a window spec in Smalltalk). We adapted the conceptification process for classes so that a `vwWindowspec` slot was added if a user interface specification existed. The specification itself is represented as a terminal in the concept network.

Using the CoBro-Nav tool to visualise such a terminal would result in one long string to be displayed (i.e. the textual representation of a window specification). So we added a different terminal node class to CoBro-Nav that would show the visual counterpart of this specification. Instead of writing explicit code for the decision to represent a terminal in the drawing as a standard terminal node or as a windowspec node, we added this information to the ConceptBase.

In doing so, CoBro-Nav can ask the terminal (or concept for that matter) its node representation class by accessing the corresponding slot in the ConceptBase. Note that up until now we only referred to the basic visualisations of a concept node and of a terminal node (i.e. the classes `CBViewConcept` and `CBViewTerminal`). In the previous section we already explained that slots are looked up through the parent chain in the concept network. Consequently we added this information as default slots (`gbConceptViewClass:`) to the concepts `Concept` and `Terminal`. We illustrate this in the following code excerpt:

```
Concepts.Concept
    gbConceptViewClass:  Concepts.CBViewConcept;
    save.

Concepts.Terminal
    gbConceptViewClass:  Concepts.CBViewTerminal;
    save.
```

This means that extending CoBro-Nav with new node visualisations boils down to:

- creating a new node class that is instantiated by CoBro-Nav to visualise the information (e.g., the class `CBViewTerminalWindowSpec`), and
- adding a `gbConceptViewClass` slot to the corresponding concept in the ConceptBase.

We show the window specification visualisation in figure 5.12. Note that this is a running user interface, so it is possible to interact with it if it has an associated application model.

Another terminal visualisation that was added in the same way to CoBro-Nav is to visualise images. This makes it possible to attach images as documentation to code and domain level entities. This illustrates the malleability of CoBro-Nav as a result of using CoBro in its implementation. We will illustrate the extension of CoBro-Nav with new node types in chapter 7.

## 5.4 Assorted Tools

In this section we will briefly introduce the other elements of CoBro that support a developer in interacting with concepts. In chapter 6 we also illustrate how these tools can be used from within CoBro-CML.

The CoBro launcher is the main window from which all the tools can be accessed. It is also the place where you can set up a connection to a ConceptBase and where you can initialize the system.

Also, as a result of the symbiotic integration with Smalltalk, one of the main ways to interact with the ConceptBase is by using a Smalltalk workspace in which you evaluate CoBro-CML statements interactively. Both are shown in the screenshot in figure 5.13.

Querying concepts interactively from within a workspace could result in a collection of concepts. If you want to manipulate this collection of concepts they can be sent to a concept bag window. From within a concept bag these concepts can be sent to the different tools of CoBro or they can be grouped together in

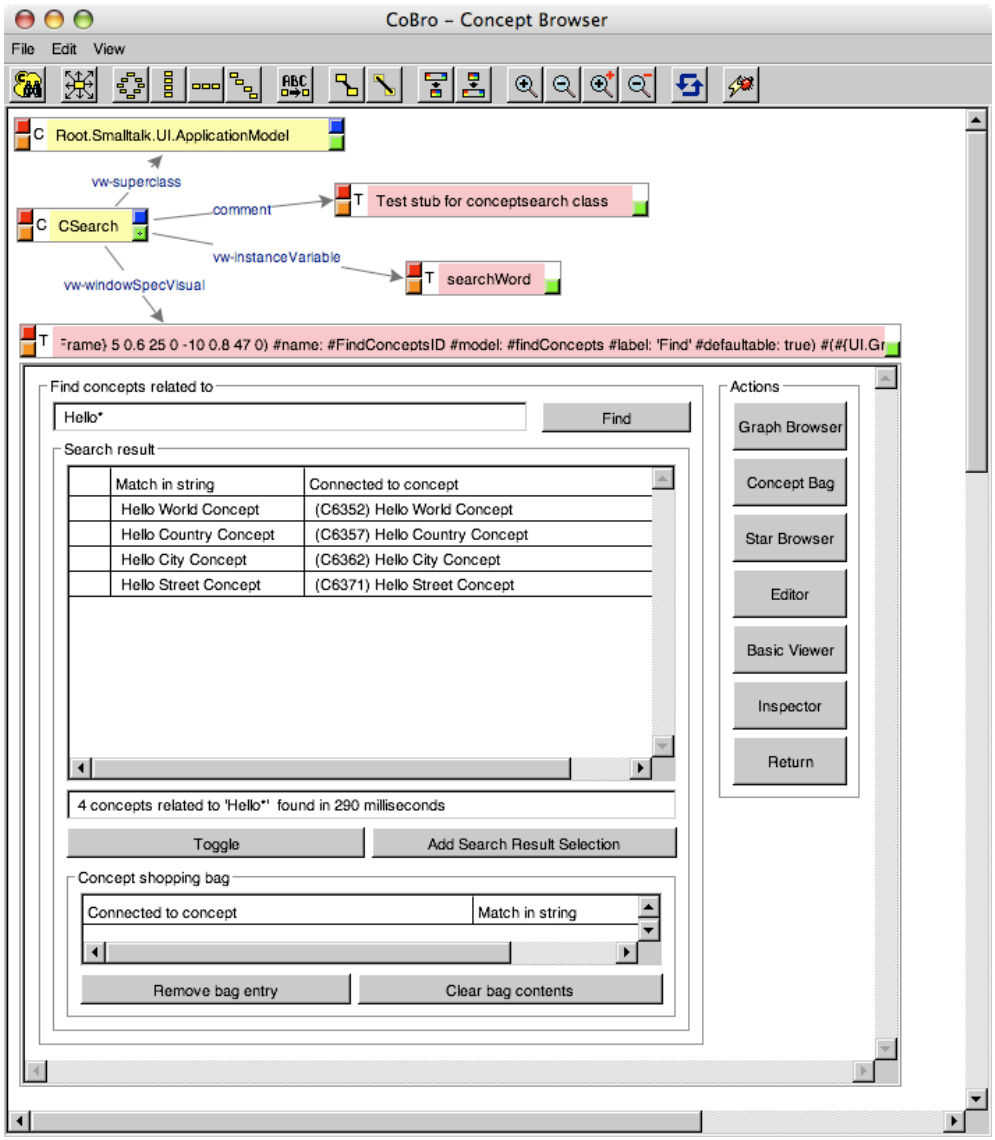



Figure 5.12: Visualising user interface specifications in CoBro-Nav.

the ConceptBase[7]. In figure 5.14 we show a concept bag window that contains a number of concepts.

Concepts can be manipulated programmatically by accessing and setting their slots. If more complex editing is required, it is often useful to use the basic concept editor. An example of this use is changing the order of the slots in a definition frame of a concept. This ordering is especially important for parent slots for instance, since they are handled from top to bottom during lookup. As

[7]As we will illustrate in chapter 6 grouping can be done by setting up a notion of group concepts in the ConceptBase.

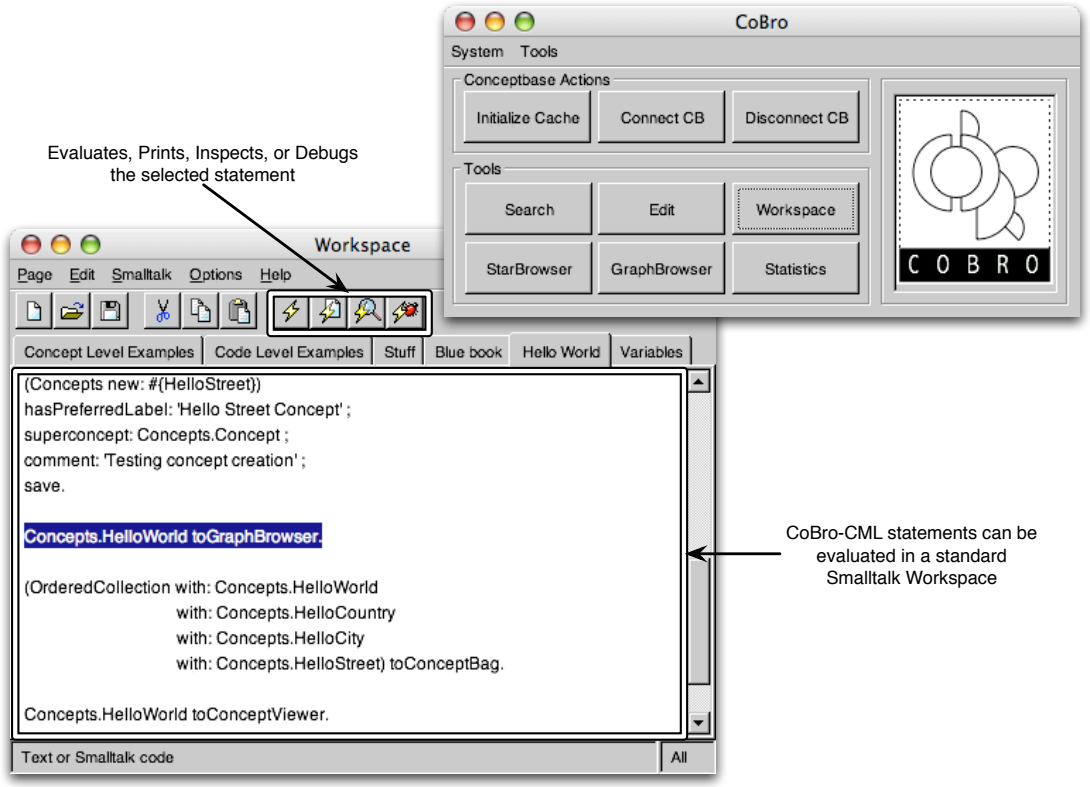



Figure 5.13: The CoBro launcher and a Smalltalk workspace.

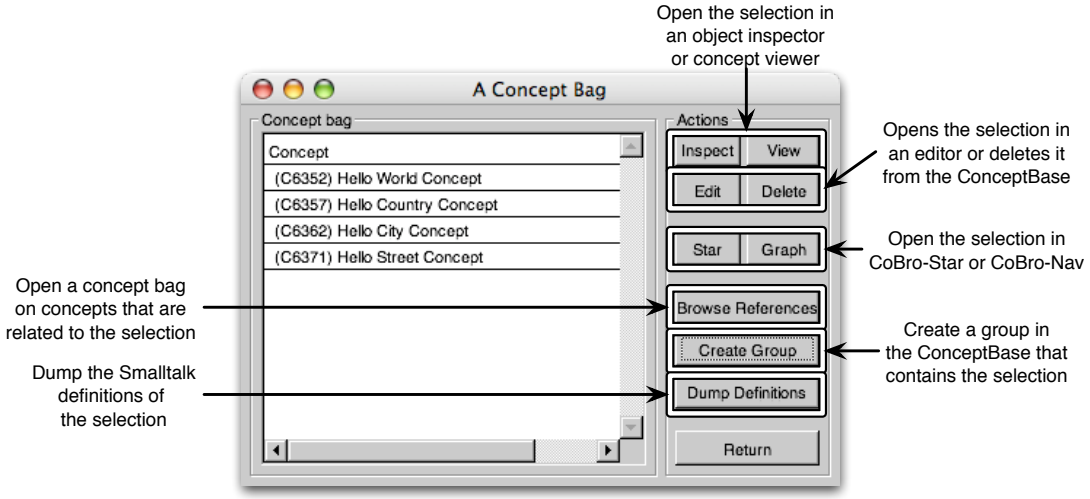



Figure 5.14: The CoBro concept bag.

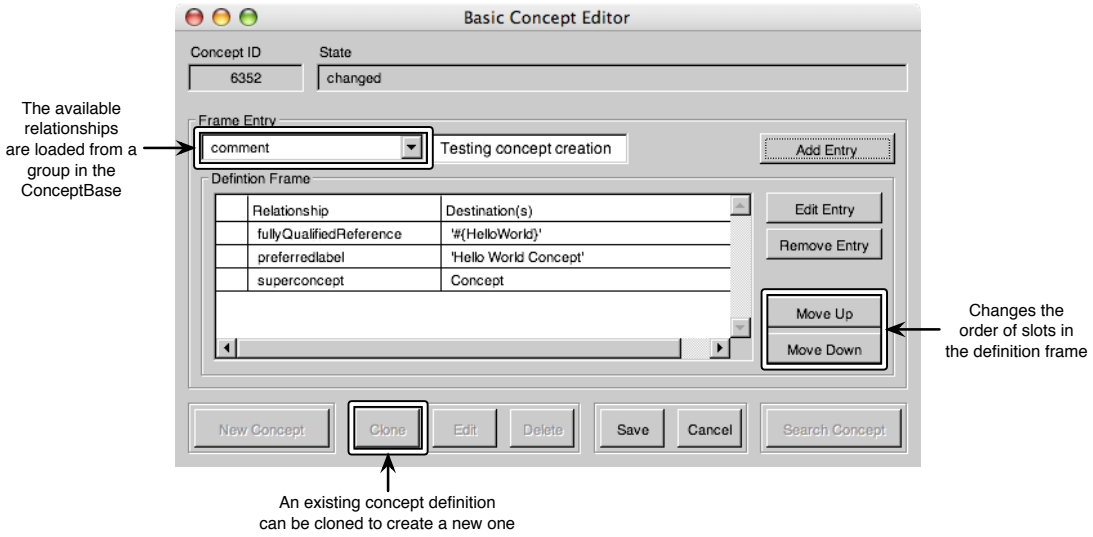



Figure 5.15: The basic concept editor.

a consequence the order of these slots determines the order in which the parent chain is traversed. Dedicated versions of this editor can be created to be more suited for a specific task. Figure 5.15 shows a screenshot of the basic editor. Note that this editor can also be used to adapt the extension of a conceptified code level entity.

Concepts are identified by their fully qualified reference. If this reference is not known, it can be located by examining the associated terminals (e.g., the comment slots). The concept locator allows us to search for concepts without knowing their fully qualified reference. It also has shopping bag functionality to collect all the concepts of your session. This is shown in figure 5.16[8].

Consulting the internal representation of concepts can be done through the basic concept viewer. Another way to view the internals would be to open a Smalltalk object inspector on a concept. We will illustrate this in chapter 6. Figure 5.17 shows a screenshot of the basic viewer.

The main tool for programming in VisualWorks Smalltalk is the Refactoring Browser. We integrated the CoBro tools in this browser to ease the manipulation of the concept level. Currently there are three CoBro extensions (a.k.a. code tools) for the system browser. The first is shown in figure 5.18 and is used to browse the concept intension and extension of the currently selected code entity in a tree-based fashion. It is also possible to send the conceptified code entity to the different CoBro tools. An evaluation pane is available in which CoBro-CML statements can be evaluated within the context of the conceptified class

[8]This is the same tool that was used in figure 5.12 to illustrate the window specification node of CoBro-Nav.

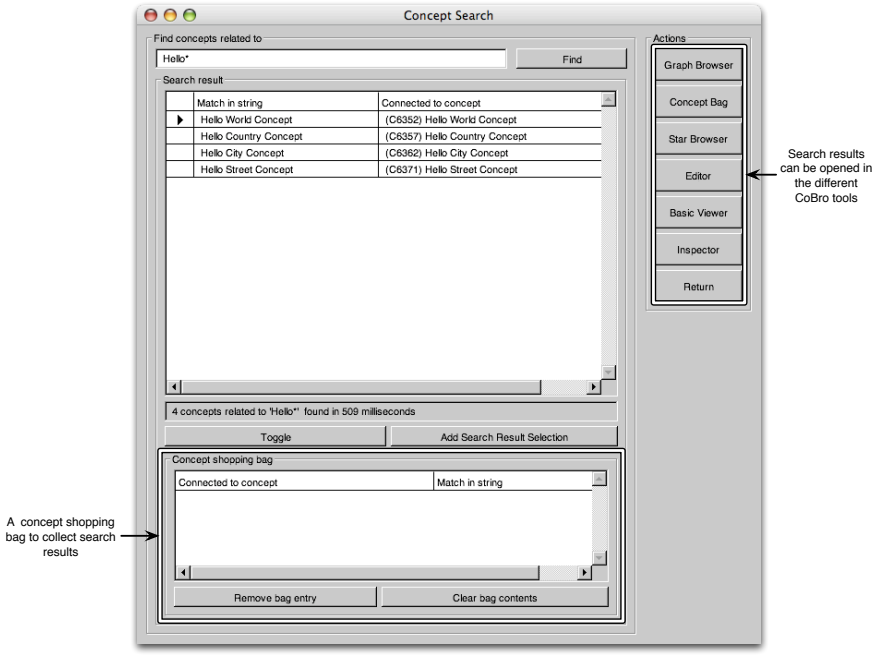



Figure 5.16: The concept locator.

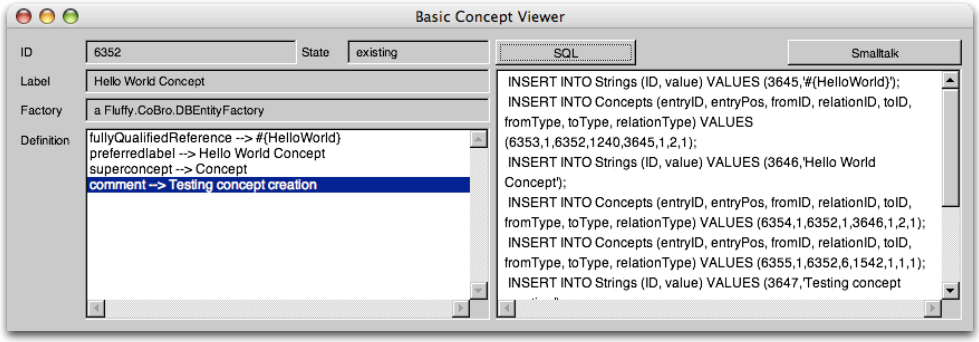



Figure 5.17: The basic concept viewer.

(`self` refers to the concept). As illustrated in the screenshot, the `relatedTo: Concepts.HelloStreet` slot was added this way.

The second code tool is similar to the first but is used for editing the definition frame of the selected code entity (figure 5.19). Slot setter messages can be send to the current selected concept. It is also possible to edit a slot, delete a slot, or move the position of a slot in the definition frame. The second code tool integrates CoBro-Nav in the Refactoring Browser. This is shown in figure 5.20, where we navigated to a number of concepts related to the selected class.

Note that both code tools depend on the granularity of conceptification (sec-

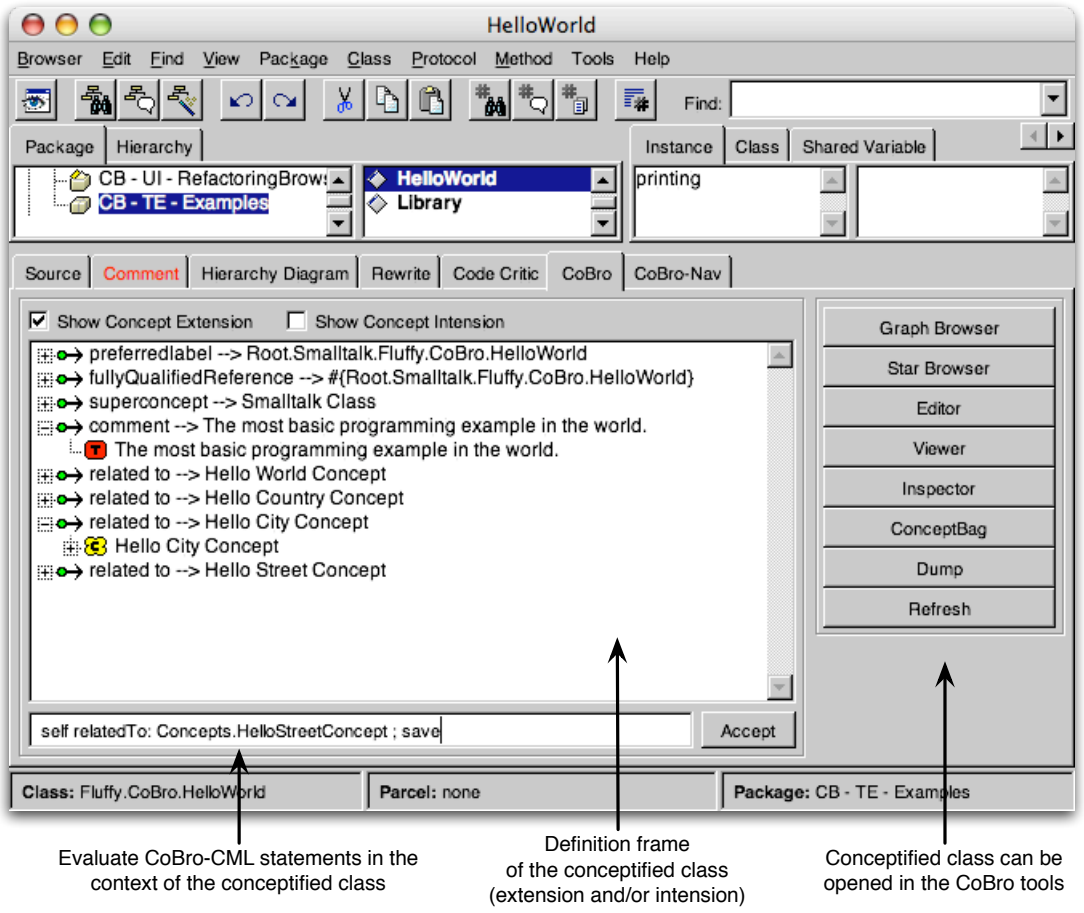



Figure 5.18: The CoBro code tool in the Smalltalk refactoring browser.

tion 5.2.3). Currently they are only available to the developer if a class is selected in the upper pane of the system browser. If methods are conceptified as well for instance, then the code tool is also available if the developer selects a method in the upper right pane of the browser.

## 5.5 Default Core Ontology

In section 5.2.4 we discussed a number of prerequisite slots for the creation of concepts and relations. For example concepts must have at least a fully qualified reference, a label, and a super concept. These prerequisites are set forth in the CoBro kernel. CoBro requires the existence of these slots to be able to function properly. For example without these slots CoBro would not be able to identify

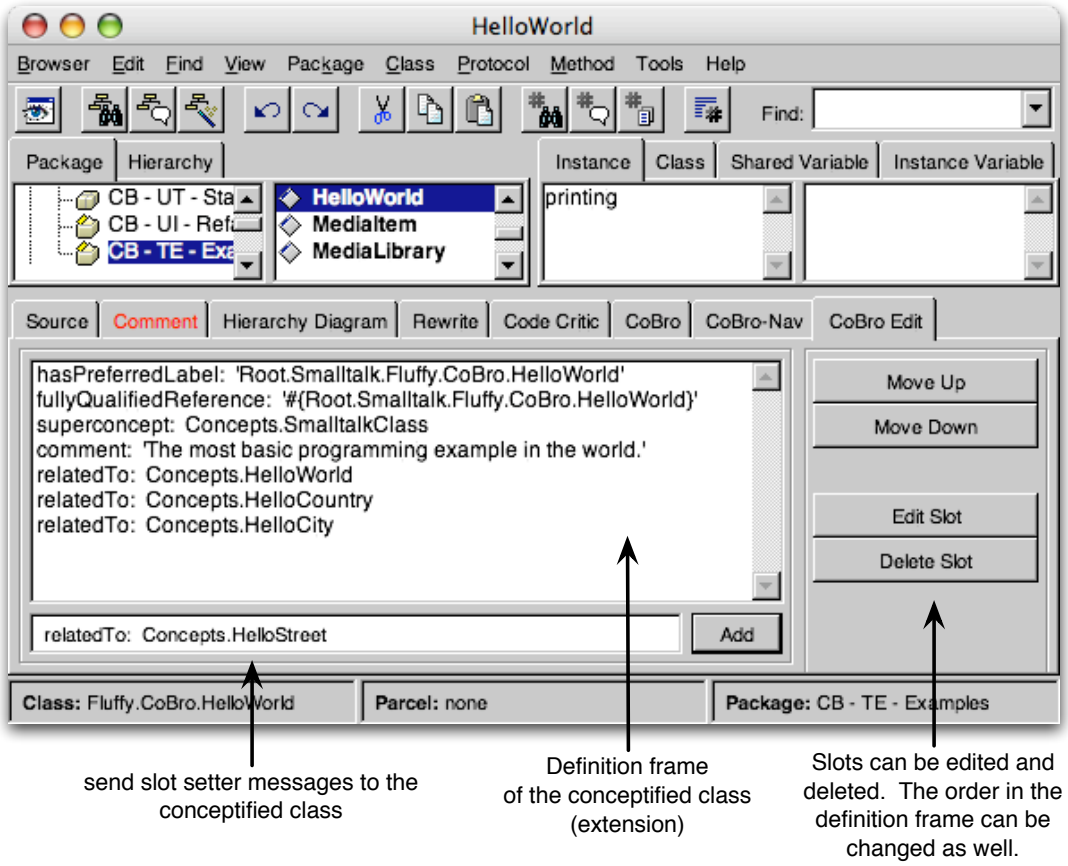



Figure 5.19: The CoBro edit tool in the Smalltalk refactoring browser.

and print the concepts[9]. Consequently it also requires the existence of relation concepts that can be used to set these slots. Hence both elements are part of the core ontology for the CoBro system. For greater flexibility we have tried to keep the prerequisites to a minimum.

Depending on the part of the CoBro environment, different prerequisites with respect to the ConceptBase are set. For example in section 5.3, CoBro-Nav expects the existence of a `gbConceptViewClass` slot to be able to visualise ConceptBase entities. Hence it requires the existence of a `gbConceptViewClass` relation concept in its core ontology. Note that such a prerequisite originates from refactoring the CoBro-Nav implementation so that it made use of concepts for

[9]Note that in essence the only hard requirement to define a concept is that it has a fully qualified reference. So it is possible to create concepts with just that one slot. If no label slot is defined then printing the concept will print 'error - unknown'. In the case that no super concept slot is provided, all the slots that are required by the CoBro tools should be added locally to the concept (e.g., the `gbConceptViewClass` slot for CoBro-Nav). Otherwise the tools are not able to handle the concept properly.

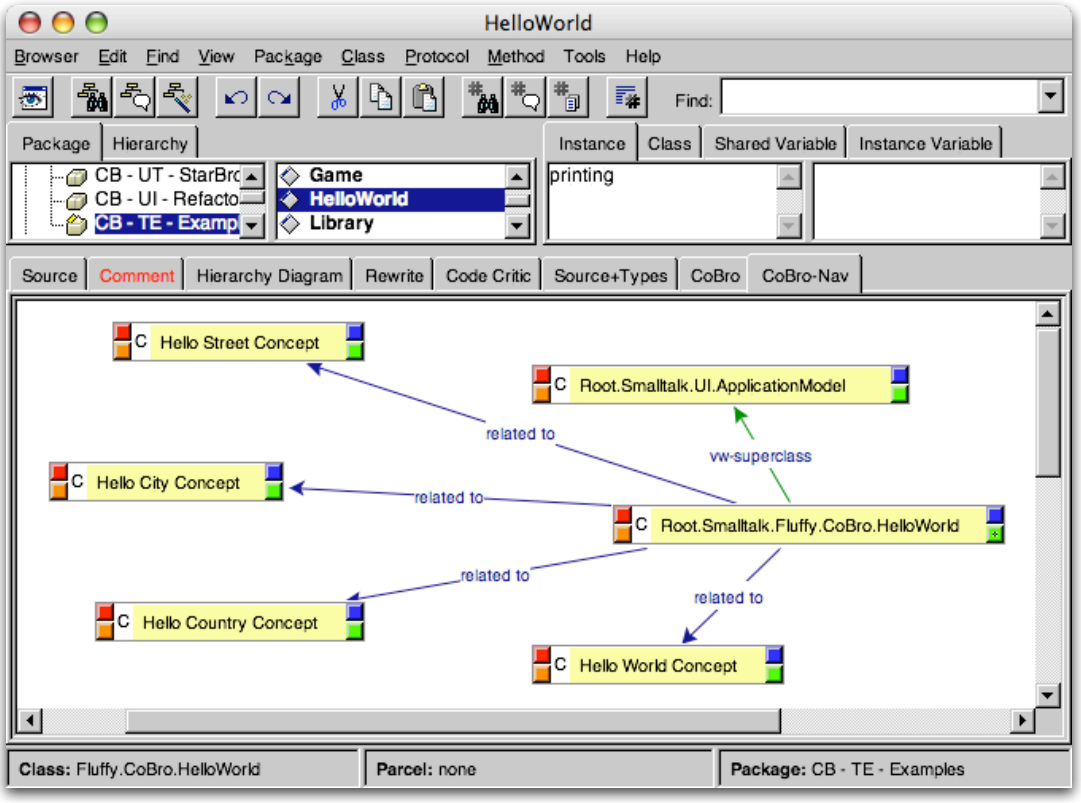



Figure 5.20: The CoBro-Nav code tool in the Smalltalk refactoring browser.

the purpose of malleability.

In figure 5.21 we show the upper level of the core ontology for the CoBro kernel. Note that the distinction between `Terminals` and `Concepts` is made explicit by modeling them as distinct concepts in the ConceptBase. Table 5.1 summarises elements of the core relations and concepts in the ConceptBase.

So in order to function properly, certain elements of the CoBro environment require the use and the existence of a number of core concepts. If other applications are built in a concept-centric fashion, then they will also require the existence of their own core ontology. Consequently we have foreseen two mechanisms that allow us to verify if the concept prerequisites are met.

The first mechanism should verify the existence of concepts in the ConceptBase. For this purpose we have incorporated a `containsConcept:` message that can be sent to the `Concepts` namespace. The use of this message is illustrated in chapter 6, section 6.2.4.

The second mechanism should verify the existence of slots in the concept definitions in the ConceptBase. This could be done by using the existing S-Unit testing framework in Smalltalk. An S-Unit test makes it possible to write down

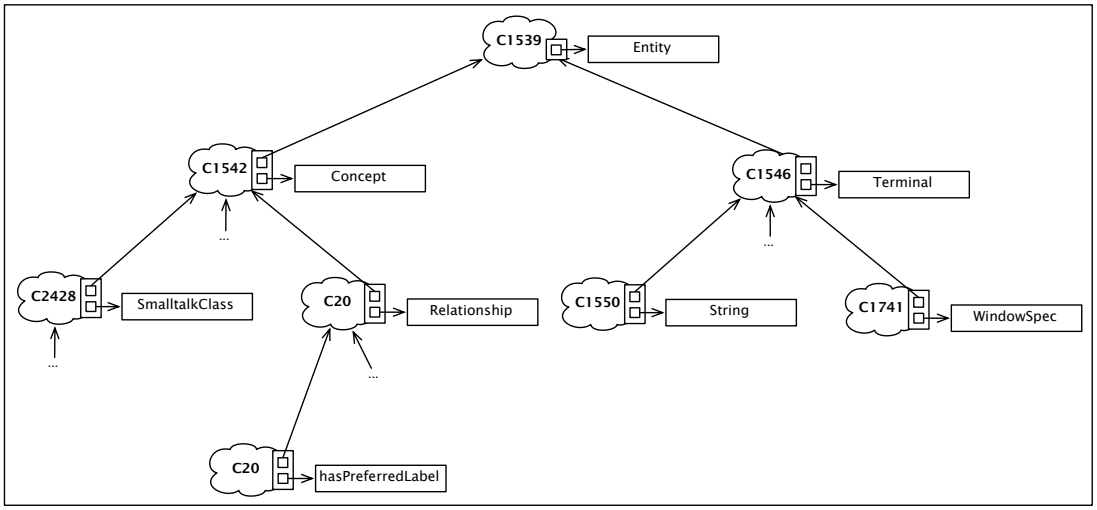



Figure 5.21: Part of the upper level of the CoBro core ontology.

a piece of code that verifies whether a part of the implementation satisfies a certain condition. We have set up a number of S-Unit tests that verify if the concepts in the ConceptBase have the required slots for the CoBro kernel. This also results in a good documentation of the core ontology and its usage for the different CoBro elements.

The elements of the different core ontologies can also be accessed as groups in the ConceptBase. One of these groups is used for instance by the CoBro kernel to cache certain core concepts to speed up the system (i.e. `EntitiesToCacheGroup`). Hence adding a concept to this group (e.g., by evaluating a CoBro-CML statement) will result in the concept to be cached.

Launching CoBro without its core ontology is not possible since CoBro-

| | |
|---|---|
| Defining concepts | `hasPreferredLabel`, `fullyQualifiedReference`, `superconcept`, `Concept`, `Terminal` |
| Defining relations | `allowedDestinations`, `multiplicity` |
| Conceptifying classes | `vwHasPreferredLabel`, `vwHasSuperclass`, `vwHasSelector`, `vwHasInstVar`, `vwHasClassVar`, `vwHasWindowSpecVisual`, `SmalltalkClass`, ... |
| Grouping | `isGroupFor`, `comment`, `Group` |
| Existing groups | `AllGroups`, `EntitiesToCacheGroup`, `SmalltalkEntitiesGroup`, `CoBroEntitiesGroup`, ... |

Table 5.1: Overview of selected core relations and concepts

CML makes use of these concepts to ensure its behavior. To avoid this chicken and egg problem, it is possible to dump core ontology concept definitions as SQL queries. Consequently these can be loaded into an empty ConceptBase, after which the CoBro system can be launched.

## 5.6 Summary

In this chapter we presented CoBro as a practical proof-of-concept implementation so as to validate our research. We introduced the overall architecture of CoBro and explained the different requirements that it should meet. We briefly summarise these requirements and how they are met by CoBro:

- **Explicit Concept Level Representation**
  Domain knowledge can be explicited in CoBro-CML by creating a concept network in the ConceptBase.

- **Coupled Concept and Code Level**
  Code level entities are deified into the concept level through an implicit conceptification process. This enables the coupling of both sides at the concept level.

- **Active Participation of the Concept Level**
  The concept level can be accessed and manipulated from within the implementation. Consequently it is possible to refactor code so that it depends on the concept level for its behavior. The participation of the concept level results in malleable software.

- **Transparency for the Developer**
  Interaction with objects and concepts is made transparent for the developer through symbiotic integration with the programming environment. This symbiotic integration is the main enabler for realising the active participation and coupling mentioned above.

We also motivated the use of Smalltalk as an implementation medium for our environment. This motivation was based on the interplay of the different meta and base levels within CoBro. Also the main motivations behind choosing a relational database to hold the concepts in the ConceptBase were discussed.

A prime element in CoBro is the concept manipulation language CoBro-CML. We presented a detailed explanation of its integration with Smalltalk and how it is used to manipulate concepts. The conceptification process that takes care of representing code level entities at the concept level was illustrated as well. This led to a discussion of the ontological granularity that is required. In essence this is a choice of what to represent at the concept level and how to represent it.

To support easy navigation in a concept network we have also described CoBro-Nav, the graphical network navigator we created. We have also illustrated how this tool is extensible since concepts play an active role in its implementation. Moreover we integrated CoBro-Nav in the Star Browser, which enables tree-based navigation.

The other tools that are part of the CoBro environment were briefly illustrated. This shows the kind of support a developer gets when interacting with CoBro.

We conclude with a discussion of the core ontologies that are used by CoBro in its implementation.

In the next chapter we present the CoBro environment in an example-based fashion. This should give an idea of how the elements introduced here are used in practice. Moreover it will show certain aspects of CoBro that did not fit in the discussion of the current chapter.

# Chapter 6

# CoBro by Example

The goal of this chapter is to give an idea of what interacting with the CoBro environment is like. This is done in an example-based fashion, where each example will highlight a particular aspect of the environment.

We begin by illustrating the manipulation of domain concepts in section 6.1. The main theme of the section is to demonstrate the use of CoBro to make domain knowledge *explicit*. Furthermore we illustrate the *symbiosis* with Smalltalk which makes manipulating concepts or objects transparent for the programmer. The examples show how to access, create, update, and delete concepts in CoBro-CML. We also zoom in on the interplay between CoBro-CML and the other tools of CoBro (e.g. CoBro-Nav).

In section 6.2 we focus on manipulating concepts that stem from the code level. The main theme of the section is to show the use of CoBro for *coupling* domain knowledge to code entities. Moreover we will illustrate how this domain knowledge can be used *actively* within the implementation. Similar to section 6.1 this is possible in a non-obtrusive way as a result of the *symbiosis* with Smalltalk. The examples will illustrate how code level concepts are created by conceptifying their corresponding code level entities. We will also illustrate how these code level concepts can be manipulated in CoBro-CML.

For each example we show what is returned by the system when the statements are evaluated. This is indicated by a '$\longrightarrow$', followed by the result. If the system responds to a statement by opening a window, then we will show this in a screenshot if it improves the understanding. All the examples in this chapter are included with the current CoBro distribution and can be evaluated in a standard Smalltalk workspace. A detailed discussion of advanced applications of CoBro is presented in chapter 7.

## 6.1 Domain Concept Examples

In this section we present a number of examples that show how a programmer can manipulate domain concepts. This illustrates the way in which domain knowledge can be made explicit by using CoBro-CML. How this domain knowledge can be coupled to the implementation, and how it can be used actively from within the implementation is shown in section 6.2.

The running example for this section consists of a small wine ontology for which we show the relevant subset in figure 6.1. The clouds represent concepts, the arrows represent relationships, and the rectangles represent terminal values. For example the `Wine` concept has an internal identification `C516`, an external identification (fully qualified reference) `#{Wine}`, a label that can be used for printing `'Wine'`, a number of comment slots, and a `superconcept` slot relating it to the `Concept` concept.

We first show how existing concepts can be retrieved from the ConceptBase, as well as a number of helper tools that can be called from within CoBro-CML. Next we illustrate how the slots in a concept definition can be manipulated by accessor and setter messages. We end this section with examples that illustrate the creation of a group-concept and a notion of projects in the ConceptBase.

### 6.1.1 Retrieving Concepts

In chapter 5 we introduced the `Concepts` namespace and our extended lookup mechanism that takes care of retrieving concepts from the ConceptBase. Since this forms the basis for the rest of this section, we first show how this namespace works in practice.

There are several ways to access a concept through the `Concepts` namespace. You can either use the dotted notation to indicate the path to the concept, or you can use the namespace as a dictionary by sending the `at:` message. The examples below will all return the `Wine` concept:

```
Root.Smalltalk.CoBro.Concepts.Wine.
Smalltalk.CoBro.Concepts.Wine.
CoBro.Concepts.Wine.
Concepts.Wine.
Concepts at:  #{Wine}.
⟶ a concept: Wine
```

The notation used for the concept key in the last example, is identical to the one Smalltalk uses to indicate a *fully qualified reference* of a binding. This is important since it allows the system to handle Smalltalk and concept references transparently. Verifying whether an entity is a concept or not can be done by

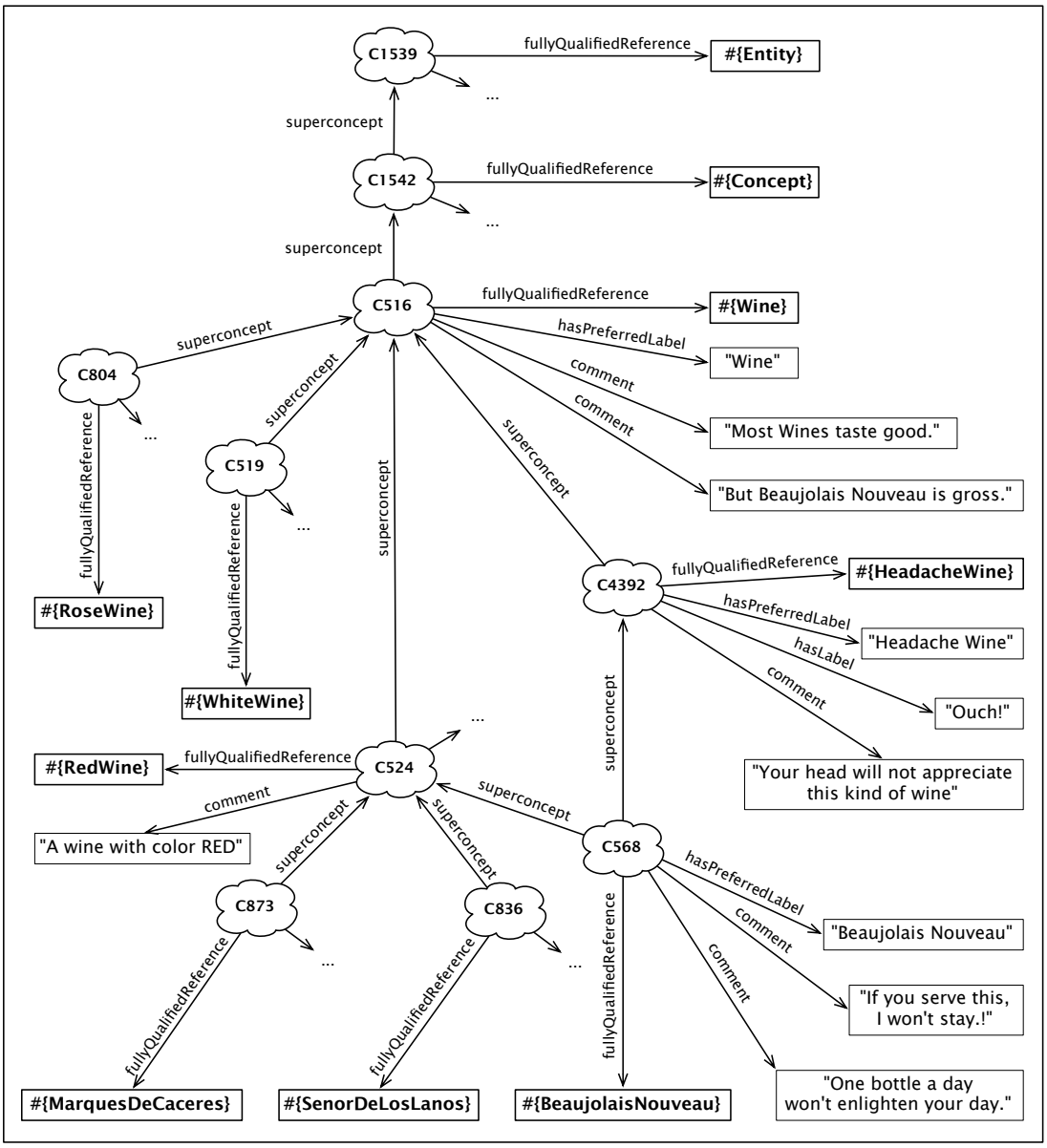



Figure 6.1: A subset of a simple wine ontology.

sending the `isConcept` message. As illustrated in the following example, even though `OrderedCollection` is not a concept, it has the same fully qualified reference as its concept counterpart `Concepts.OrderedCollection`[1]:

[1]Note that in Smalltalk the fully qualified reference of a class by default starts from `Root.Smalltalk`. We illustrate how to get the complete reference later in this chapter.

```
Concepts.Wine isConcept.
⟶ true
Concepts.Wine fullyQualifiedReference.
⟶ a binding reference: #{Wine}

OrderedCollection isConcept.
⟶ false
OrderedCollection fullyQualifiedReference.
⟶ a binding reference: #{Core.OrderedCollection}
Concepts.OrderedCollection isConcept.
⟶ true
Concepts.OrderedCollection fullyQualifiedReference.
⟶ a binding reference:
    #{Root.Smalltalk.Core.OrderedCollection}
```

The following illustrates what happens if you refer to a concept that does not exist in the ConceptBase, and that cannot be conceptified. Similar to Smalltalk, this will throw an exception which reports that it is an unbound identifier:

```
Concepts.TheConceptThatNeverExisted.
⟶ an exception: The identifier has no binding
```

To find out programmatically whether a concept exists or not, you can send the message `containsConcept:` or `containsAllConcepts:` to the concepts namespace. This is useful for code that depends on the existence of certain concepts[2]:

```
Concepts containsConcept:  #{TheConceptThatNeverExisted}.
⟶ false
Concepts containsAllConcepts:
    #(#{Wine} #{RedWine} #{WhiteWhine}).
⟶ true
```

Since CoBro-CML is implemented in symbiosis with Smalltalk, it can be mixed with and used from within Smalltalk programs. So if you do not want to refer directly to a concept, you can manipulate it through a variable. This is shown in the following where the concept `Wine` is retrieved from the ConceptBase and

[2]In Smalltalk, an instance of a collection can be created by enumerating the elements using the following notation: `aCollection := #(element1 ... element3)`.

stored in the variable `d1`[3]:

```
d1:= Concepts.Wine.
⟶ a variable d1 bound to: Concepts.Wine
```

Opening a Smalltalk object inspector on `d1` shows the internal details of the `Wine` concept[4]:

```
Concepts.Wine inspect.
⟶ a Smalltalk object inspector on: the Wine concept (see figure 6.2)
d1 inspect.
⟶ a Smalltalk object inspector on:
    d1 containing the Wine concept (see figure 6.2)
```

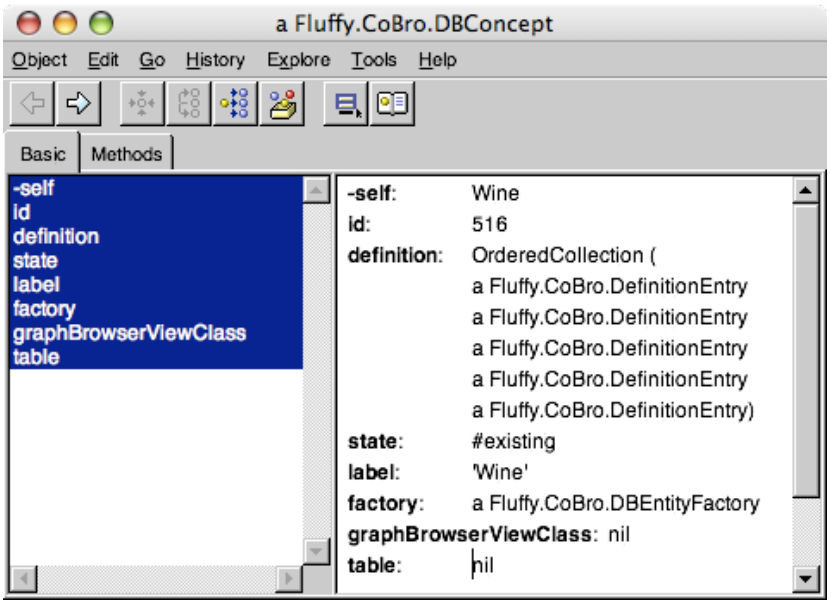



Figure 6.2: A Smalltalk object inspector on `Concepts.Wine`.

Using the inspector you can navigate to the collection that holds the definition frame of the concept[5]. Each slot that is added to the concept is represented by its own entry in this frame. Note that every definition entry consists of a relationship

[3]From now on when we refer to `d1`, we refer to the variable assigned to `Concepts.Wine`.

[4]If we refer to `d1` in the coming examples then we refer to `Concepts.Wine`.

[5]Note that a concept definition is only loaded on demand. To force loading the definition, evaluate `self definition` in the evaluation pane of the inspector.

concept that is associated with a collection of destinations (a collection to support n-ary relationships). This is shown in figure 6.3.

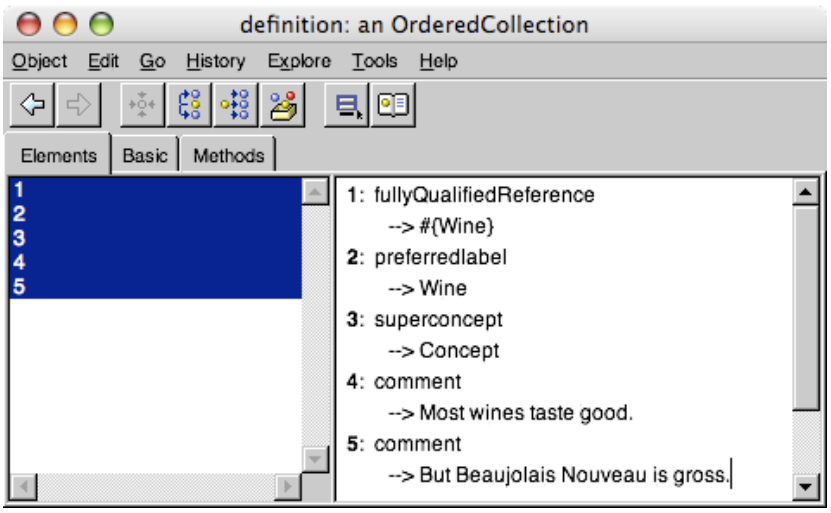



Figure 6.3: A Smalltalk object inspector on the definition frame of `Concepts.Wine`.

### 6.1.2 Helper Messages

Helper messages are useful for interacting with concepts during programming. They are also used by the different CoBro tools to provide certain parts of their functionality. Some of these messages will open a concept or a collection of concepts in one of the CoBro tools[6]:

```
d1 toConceptViewer.
```
⟶ *opens a basic concept viewer on the* **`Wine`** *concept*
```
d1 toConceptEditor.
```
⟶ *opens the* **`Wine`** *concept in the basic concept editor*
```
d1 toGraphBrowser.
```
⟶ *opens* CoBro-Nav *on the* **`Wine`** *concept for visual browsing*
```
d1 toStarBrowser.
```
⟶ *opens the extended StarBrowser on the* **`Wine`** *concept for hierarchical, textual, and visual browsing*
```
d1 toConceptBag.
```
⟶ *opens the* **`Wine`** *concept in a concept bag for further manipulation*

[6]For an explanation of the respective tools we refer to chapter 5.

Other helper messages can be used for instance to generate a textual representation of a concept. The following example shows only two of these. The first, `smalltalkDefinition`, will generate a string that corresponds to the definition of the concept in Smalltalk syntax. This is useful to generate a template if you want to define a concept that has many slots in common with an existing one (see section 6.1.5):

```
d1 smalltalkDefinition.
⟶ c := Concepts new: #{Wine}.
   c hasPreferredLabel: 'Wine'.
   c superconcept: Concepts.Concept.
   c comment: 'Most wines taste good.'.
   c comment: 'Only Beaujolais Nouveau is gross.'.
   c save.
```

A second helper message, `sqlDefinition`, generates the SQL representation that is used to store the concept in the ConceptBase. This is useful for bootstrapping the definition of elements in the core ontology:

```
d1 sqlDefinition.
⟶ INSERT INTO Strings (ID, value)
        VALUES (3484,'#{Wine}');
    INSERT INTO Concepts (entryID, entryPos, fromID,
        relationID, toID, fromType, toType, relationType)
        VALUES (6055,1,516,1240,3484,1,2,1);
    INSERT INTO Strings (ID, value)
        VALUES (1971,'Wine');
    INSERT INTO Concepts (entryID, entryPos, fromID,
        relationID, toID, fromType, toType, relationType)
        VALUES (6056,1,516,1,3485,1,2,1);
    INSERT INTO Concepts (entryID, entryPos, fromID,
        relationID, toID, fromType, toType, relationType)
        VALUES (6057,1,516,6,1542,1,1,1);
    INSERT INTO Strings (ID, value)
        VALUES (3486,'Most wines taste good.');
    INSERT INTO Concepts (entryID, entryPos, fromID,
        relationID, toID, fromType, toType, relationType)
        VALUES (6058,1,516,687,3486,1,2,1);
    INSERT INTO Strings (ID, value)
        VALUES (3487,'But Beaujolais Nouveau is gross.');
    INSERT INTO Concepts (entryID, entryPos, fromID,
        relationID, toID, fromType, toType, relationType)
        VALUES (6059,1,516,687,3487,1,2,1);
```

### 6.1.3 Querying Concepts by Accessing Slots

The slots in the definition frame of a concept can be accessed by sending messages to the concept. The message names that are used are the fully qualified references of the slot's relationship. For example accessing the preferredLabel slot is done as follows:

```
Concepts.Wine hasPreferredLabel.
⟶ a terminal for the string: 'Wine'
```

The following example illustrates what happens if you send a message to access a slot that does not exist in the concept. Similarly to Smalltalk, this will throw an exception that it is a new message:

```
Concepts.Wine theSlotThatNeverExisted.
⟶ an exception: theSlotThatNeverExisted is a new message
```

As we have seen in chapter 5, relationships themselves are represented by concepts. Consequently we can access the concept that represents a relationship, e.g. the `hasPreferredLabel` concept:

```
Concepts.HeadacheWine hasPreferredLabel.
⟶ a terminal for the string: 'Headache Wine'
Concepts.hasPreferredLabel.
⟶ a concept: hasPreferredLabel
```

Depending on the multiplicity that was defined for a certain relationship, a slot accessor message will return either:

- a single terminal representing the slot's value
- a collection of terminals containing the values for each occurrence of the slot

To find out whether a slot accessor will return a single terminal or a collection of terminals, you can access the `multiplicity` slot of the corresponding relationship concept. This is illustrated by the next example:

```
Concepts.fullyQualifiedReference multiplicity.
⟶ a collection representing the multiplicity: #(1 1)
d1 fullyQualifiedReference.
⟶ a terminal for the fully qualified reference: #{Wine}

Concepts.comment multiplicity.
⟶ an ordered collection representing the multiplicity: #(1 #n)
d1 comment.
⟶ an ordered collection with terminals for all comment slots:
    #('Most wines taste good.' 'Only Beaujolais Nouveau is gross.')
d1 comment first.
⟶ a terminal for the string: 'Most wines taste good.'
```

So if multiple slots of the same type are allowed, and accessing these returns a collection, then you can use normal Smalltalk collection behavior to iterate over the elements in the collection (e.g. `first`, `last`, `at: anIndex`, ...). The next example opens a concept in the graph browser which shows that it has two

comment slots (figure 6.4). Accessing the `comment` slot results in a collection of terminals. In the example we illustrate iterating over this collection by printing every comment in a Transcript window:

```
d2 := Concepts.BeaujolaisNouveau.
d2 toGraphBrowser.
⟶ opens d2 in CoBro-Nav (see figure 6.4)
d2 comment first.
⟶ a terminal for the string:
    'One bottle a day won't enlighten your day.'

d2 comment do:
        [:each | Transcript cr.
                 Transcript show:  each printString].
⟶ loops over the collection of comments and prints each
    in the Smalltalk Transcript window
```

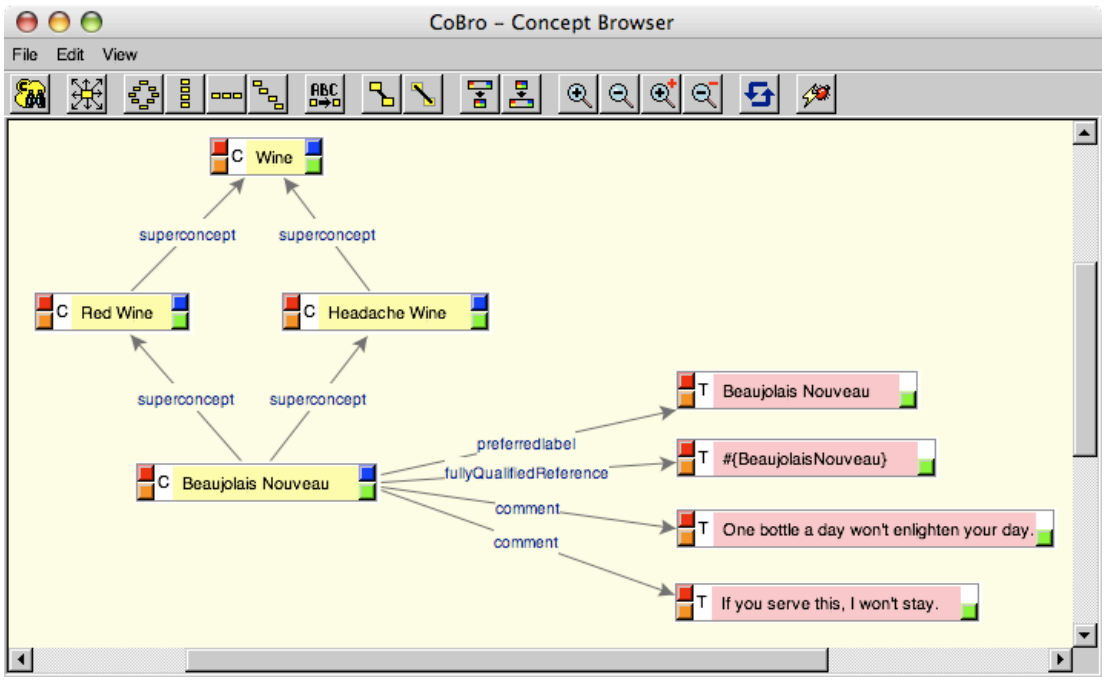



Figure 6.4: Exploring `Concepts.BeaujolaisNouveau` in the CoBro-Nav graph browser.

### 6.1.4 Lookup of Slots in the Parent Chain

As explained in chapter 5, the default behavior of accessors can be overridden in CoBro. The default core ontology includes a `superconcept` relationship-concept that indicates a parent/child relationship between concepts. The default interpretation of `superconcept` is that a concept can have multiple parents. As we discussed in chapter 5, section 5.2.4, The lookup mechanism is implemented as a breadth-first search. Hence it will first look within the concept itself after which it will go up one level and look within its parents from left to right. If the slot is not found at that level, it will repeat this process for the next level. Note that the meta-level interface of CoBro makes it possible to deviate from this default lookup mechanism as described in chapter 5, section 5.2.4. Hence you can easily override the current lookup or install your own relationship in the ConceptBase (e.g. `Concepts.myOwnSuperconcept`). The following examples illustrate the default `superconcept` accessor, as well as a number of helper messages, namely:

- `super`
  → selects the first superconcept encountered at this level
- `super:aRef`
  → selects the first superconcept named `aRef` at this level
- `allSuperconcepts`
  → returns a collection containing the superconcept chain
- `withAllSuperconcepts`
  →`allSuperConcepts` including the current concept
- `allSubconcepts`
  →returns a collection containing the subconcept chain
- `withAllSubconcepts`
  →`allSubConcepts` including the current concept

```
d2 := Concepts.BeaujolaisNouveau.
d2 superconcept.
⟶ an ordered collection containing the concepts
    RedWine and HeadacheWine

d2 superconcept collect:
          [:each | each hasPreferredLabel printString].
⟶ an ordered collection containing the labels of the concepts:
    #('Red Wine' 'Headache Wine')
```

The following example shows what happens if you want to access a slot that should be looked up through the parent chain. In this case `Concepts.BeaujolaisNouveau` has no `hasLabel` slot of its own. The breadth-first lookup will first look inside `Concepts.RedWine` after which the slot is found in `Concepts.HeadacheWine`:

```
d2 := Concepts.BeaujolaisNouveau.
d2 hasLabel first.
⟶a terminal for the string: 'Ouch!'
```

Selecting the first `superconcept` slot is done by sending the `super` message. If you want to select a particular superconcept, then you need to use the `super:` message by indicating the reference of the concept:

```
d2 := Concepts.BeaujolaisNouveau.
d2 super comment first.
⟶ a terminal for the string: 'A wine with color RED'

d2 super: #{Wine}.
⟶ Wine is not a direct parent so: nil

(d2 super: #{HeadacheWine}) comment first.
⟶ a terminal for the string:
    'Your head will not appreciate this kind of wine'
(d2 super: Concepts.HeadacheWine fullyQualifiedReference)
    comment first.
⟶ a terminal for the string:
    ' Your head will not appreciate this kind of wine'
(d2 superconcept at: 2) comment first.
⟶ a terminal for the string:
    'Your head will not appreciate this kind of wine'
```

It is often necessary to have access to the entire chain of parents or children starting at a particular concept. In the following we illustrate how this can be done. At the end of the example we open the resulting set of concepts in a concept bag for further manipulation:

```
d2 := Concepts.BeaujolaisNouveau.
d2 allSuperconcepts.
```
⟶ *an ordered collection containing the concepts*
*RedWine*, *HeadacheWine*, *Wine*, *Concept*, *Entity*

```
d2 withAllSuperconcepts toConceptBag.
```
⟶ *opens a concept bag on an ordered collection containing*
*BeaujolaisNouveau*, *RedWine*, *HeadacheWine*,
*Wine*, *Concept*, *Entity*

```
Concepts.RedWine allSubconcepts.
```
⟶ *an ordered collection containing the concepts*
*BeaujolaisNouveau*, *SenorDeLosLanos*,
*MarquesDeCaceres*

```
Concepts.RedWine withAllSubconcepts toConceptBag.
```
⟶ *opens a concept bag on an ordered collection containing*
*RedWine*, *BeaujolaisNouveau*, *SenorDeLosLanos*,
*MarquesDeCaceres* *(see figure 6.5)*

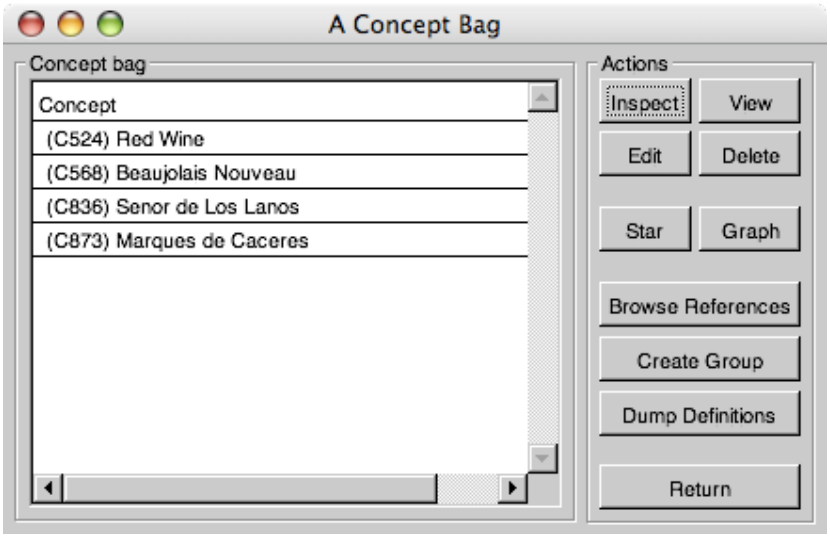



Figure 6.5: A concept bag holding the result from `Concepts.RedWine withAllSubconcepts`

### 6.1.5 Creating Concepts and Setting Slots

A concept is created by sending the `new:` message to the `Concepts` namespace. The argument to this message is the fully qualified reference that will be used to uniquely identify the concept. With the default core ontology loaded, a concept is expected to have:

- exactly one unique fully qualified reference (`fullyQualifiedReference:`)
- exactly one preferred label (`hasPreferredLabel:`)
- one or more super concepts (`superconcept:`)

The `save` message is used to make the concept persistent in the ConceptBase:

```
d3 := Concepts new:  #{ChateauMigraine}.
d3 hasPreferredLabel:  'Le Grand Chateau Migraine'.
d3 superconcept:  Concepts.HeadacheWine.
d3 comment:  'My name says it all'.
d3 save.
```
⟶ *a variable d3 bound to:* ***Concepts.ChateauMigraine***

```
d3 toStarBrowser
```
⟶ *opens the* ***ChateauMigraine*** *concept in the extended Star Browser (see figure 6.6)*

A concept can be deleted from the ConceptBase by sending the `delete:` message. If the concept is found and can be deleted, this returns `true`, otherwise it returns `false`:

```
Concepts delete:  Concepts.ChateauMigraine.
```
⟶ *deletes the concept* ***Concepts.ChateauMigraine: true***

Using Smalltalk's *cascading message sending* style, a concept can be created in a less verbose way. The semicolons in the example are used to cascade the messages:

```
d4 := (Concepts new:  #{ChateauMigraine})
      hasPreferredLabel:  'Le Grand Chateau Migraine' ;
      superconcept:  Concepts.HeadacheWine ;
      comment:  'My name says it all' ;
      save.
```
⟶ *a variable d4 bound to:* ***Concepts.ChateauMigraine***

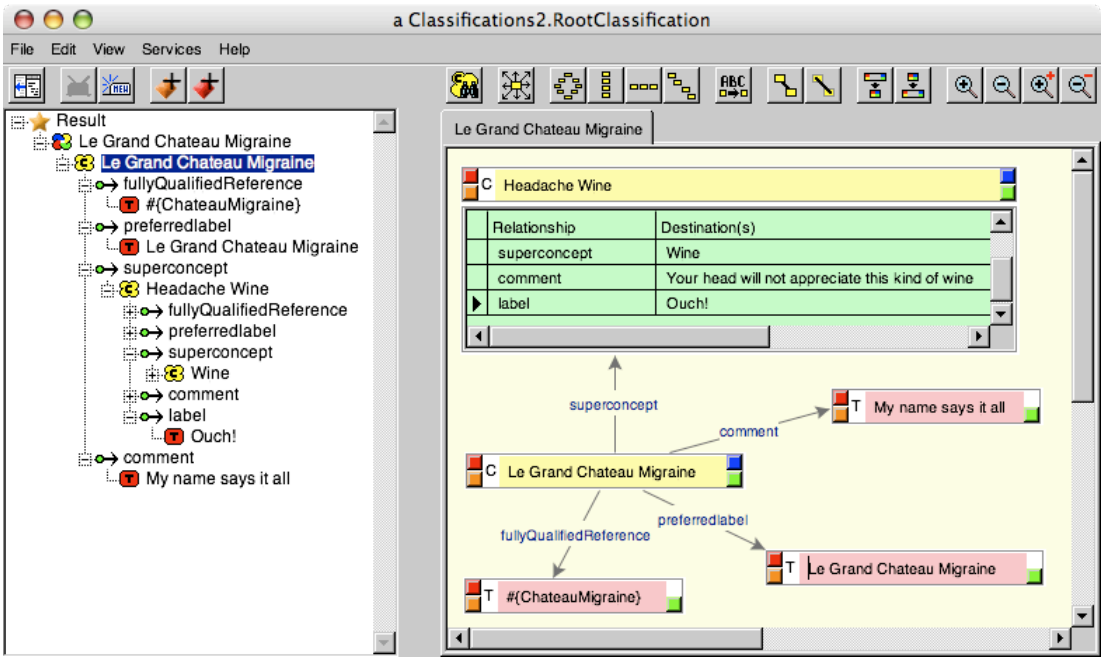



Figure 6.6: A Star Browser opened on the newly created `ChateauMigraine` concept.

The following example illustrates what happens if you attempt to create a concept that already exists:

```
Concepts.TheConceptThatExists.
⟶ a concept: Concepts.TheConceptThatExists

Concepts new: #{TheConceptThatExists}.
⟶ an exception and name suggestion:
    The fully qualified reference is already taken.
    Try using '#{TheConceptThatExists1}'
```

As shown, the system uses a mnemonic generator to suggest a name that is unique within the system.

Since relationships are also concepts, they can be created in the same way. As we have seen, the fully qualified reference of a relationship is used as the message name to access a slot. With the default core ontology loaded, a relationship is expected to have:

- exactly one unique fully qualified reference
- exactly one preferred label
- the `Concepts.Relationship` as one of its superconcepts
- exactly one `multiplicity:` slot
- exactly one `allowedDestinations:` slot

For instance the definition of the `comment` relationship looks like:

```
(Concepts new:  #{comment})
       hasPreferredLabel:  'comment';
       superconcept:  Concepts.Relationship ;
       multiplicity:  '#(1 #n)' ;
       allowedDestinations:  '#(#{String})' ;
       save.
```
⟶ *a concept:* ***Concepts.comment***

This definition can easily be generated by sending the `smalltalkDefinition` message to a concept. This way you can use an existing definition as a template for creating your own relationship for instance.

The `multiplicity:` in the previous example indicates how many occurrences of this slot are allowed within one concept. The `allowedDestinations:` contains a collection of concept references to which the relationship may point. In the above example, the `#{String}` concept refers to a `Terminal` concept. Such a destination type means that it will point to a terminal value (in this case a string).

The former examples showed how to create a concept from scratch. Adding a slot to an existing concept definition can be done by sending a message or by opening a concept editor. Note that the editor can be used to change the order of the slots in the definition frame:

```
Concepts.ChateauMigraine
       comment:  'Is it the chateau or the migraine
                  that is grand?'  ;
       save.
```
⟶ *adds a* ***comment*** *slot to* ***ChateauMigraine***

```
Concepts.ChateauMigraine toConceptEditor
```
⟶ *opens the* ***ChateauMigraine*** *concept in the basic concept editor (see figure 6.7)*

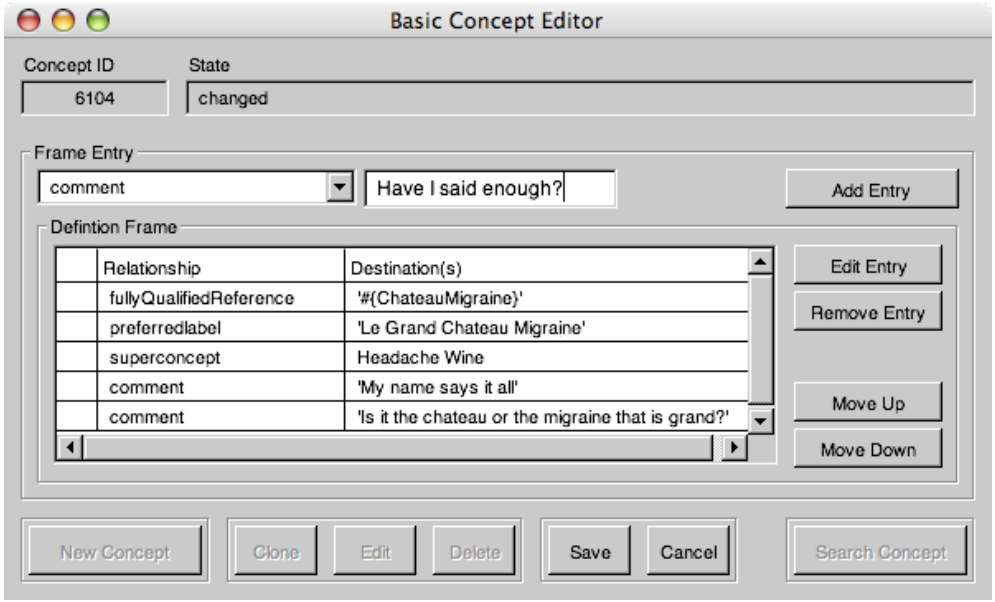



Figure 6.7: A concept editor opened on the `ChateauMigraine` concept

### 6.1.6 A Notion of Groups and Projects

In practice it happens a lot that there is a set of concepts that you need to work with frequently, or that simply belong together. Therefore it would be useful to have some kind of mechanism that allows you to group them. Instead of having some special kind of concept, this is achieved by defining a group concept.

The way to go is to start by defining a `Group` concept and an `isGroupFor:` relationship. In the following examples we will show the corresponding statements:

```
(Concepts new:  #{Group})
       hasPreferredLabel:  'Group' ;
       superconcept:  Concepts.Concept ;
       save.
⟶ a concept: Concepts.Group

(Concepts new:  #{isGroupFor})
       hasPreferredLabel:  'isGroupFor' ;
       superconcept:  Concepts.Relationship ;
       multiplicity:  '#(1 #n)' ;
       allowedDestinations:  '#(#{Concept})' ;
       save.
⟶ a concept: Concepts.isGroupFor
```

Now that a `Group` concept and an `isGroupFor` relationship is created, we can use them as follows to create a group to hold some favorite wines:

```
(Concepts new:  #{MyFavoriteWines})
       hasPreferredLabel:  'Favorite wines' ;
       superconcept:  Concepts.Group ;
       comment:  'These are all the wines I like' ;
       isGroupFor:  Concepts.SenorDeLosLanos ;
       isGroupFor:  Concepts.MarquesDeCaceres ;
       isGroupFor:  Concepts.Concepts.RoseWine ;
       save.
⟶ a concept: Concepts.MyFavoriteWines

Concepts.MyFavoriteWines isGroupFor toConceptBag
⟶ opens a concept bag with the concepts
    SenorDeLosLanos, MarquesDeCaceres, RoseWine
```

Browsing all groups in the system or creating a group that holds them all, can be done by:

```
Concepts.Group allSubconcepts toConceptBag
```
⟶ *opens a concept bag on all the group concepts currently in the system*

```
g := (Concepts new:  #{AllGroups})
       hasPreferredLabel:  'All Groups' ;
       superconcept:  Concepts.Group ;
       comment:  'Holds all the groups at this
                  point in time' ;
       save.

Concepts.Group allSubconcepts do:
       [:each | g isGroupFor:  each].

g save.
```
⟶ *a concept* ***AllGroups*** *referring to all the groups in the system at this point in time*

Another way to indicate that concepts belong together is to use the dotted notation we introduced earlier for namespaces. Concept names can thus be organised similarly to the way namespaces in Smalltalk are organised. This provides an easy way to disambiguate concepts with the same name between projects (instead of having to append the names with numbers for instance):

```
(Concepts new:  #{MyProjectX.SubTaskA.Employee})
       hasPreferredLabel:  'Employee' ;
       superconcept:  Concepts.Concept ;
       comment:  'An Employee concept in the context of
                  Project X - Subtask A' ;
       save.

(Concepts new:  #{MyProjectX.SubTaskB.Employee})
       hasPreferredLabel:  'Employee' ;
       superconcept:  Concepts.Concept ;
       comment:  'An Employee concept in the context of
                  Project X - Subtask B' ;
       save.
```

⟶ *the creation of two distinct* ***Employee*** *concepts, each with a different dotted name*

As might be expected, you need to take care that the dotted concept names do not overlap with the possible paths in the Smalltalk environment. If such an overlap does occur, an exception will be raised that indicates that the name is already taken.

## 6.2 Implementation Concept Examples

The previous section targeted examples that illustrated the manipulation of domain level concepts. In this section we focus on a number of examples that manipulate concepts that stem from the code level. The manipulation of either domain or code level concepts is transparent for the programmer. This means that all the domain concept examples can be applied to code level concepts as well.

We start by showing how a code level concept is retrieved through the `Concepts` namespace and what is returned as a result of the implicit (default) conceptification process. Next we show the different ways in which domain concepts can be *coupled* to the code level concepts and vice versa. Consequently we present examples that illustrate the *active* use of domain concepts in the implementation. We end the section by showing what happens if we visualise a code level concept in CoBro-Nav, when the corresponding code entity has a user interface defined.

### 6.2.1 Retrieving Concepts

In section 6.1.1 we illustrated the use of the `Concepts` namespace. Retrieving concepts that stem from the code level is done through the same namespace. The only difference lies in the origin of the fully qualified reference that is preceded with '`Concepts.`'. Whereas for domain concepts we could use our own references to uniquely identify them, we now have to use the fully qualified reference of the code entity in the Smalltalk environment. Again, as a result of namespace imports, the dotted name can be abbreviated if necessary. The following examples will all return the conceptified version of the `Root.Smalltalk.Core.Number` class:

```
Concepts.Root.Smalltalk.Core.Number.
Concepts.Smalltalk.Core.Number.
Concepts.Core.Number.
Concepts.Number.

Concepts at:  #{Root.Smalltalk.Core.Number}.
```

⟶ *a concept with fully qualified reference* `#{Root.Smalltalk.Core.Number}`, *representing the conceptified* `Number` *class*

Currently we have implemented the system in a way that reserves all the existing class names as fully qualified references. Hence trying to create a domain concept with the name `Root.Smalltalk.Core.Number` will result in an exception indicating that it is already taken. Nevertheless, this behavior can be easily overridden so that this does become possible if needed. Note that the fully qualified reference that is stored in the ConceptBase always contains the full path to the code entity[7]:

```
Concepts.Number isConcept.
⟶ true
Concepts.Number fullyQualifiedReference.
⟶ a binding reference: #{Root.Smalltalk.Core.Number}

Number isConcept.
⟶ false
Number fullyQualifiedReference.
⟶ a binding reference: #{Core.Number}
Number fullyQualifiedReferenceFrom:  nil.
⟶ a binding reference: #{Root.Smalltalk.Core.Number}
Number fullyQualifiedReferenceFrom:  Root.Smalltalk.Core.
⟶ a binding reference: #{Number}
```

As we have already stated in section 6.1.1, the fact that the Smalltalk and the concept references are implemented identically allows the programmer to manipulate them transparently. This is shown in the following where we open a Smalltalk system browser on the binding value for the reference:

```
Number fullyQualifiedReference value browse.

Concepts.Number fullyQualifiedReference value browse.
⟶ opens a system browser on the class Number
    (see figure 6.8)
```

It is important to note that conceptified entities are not saved by default. An explicit `save` message is needed to store the extensional part of its definition. This is because conceptification consists of adding a number of slots that are defined by an intensional definition. Since these slots are computed, they are not stored

[7]By default, Smalltalk looks up references starting at `Root.Smalltalk`. To get an environment independent unique name that includes this part, you should send the basic message `fullyQualifiedReferenceFrom:  nil`.

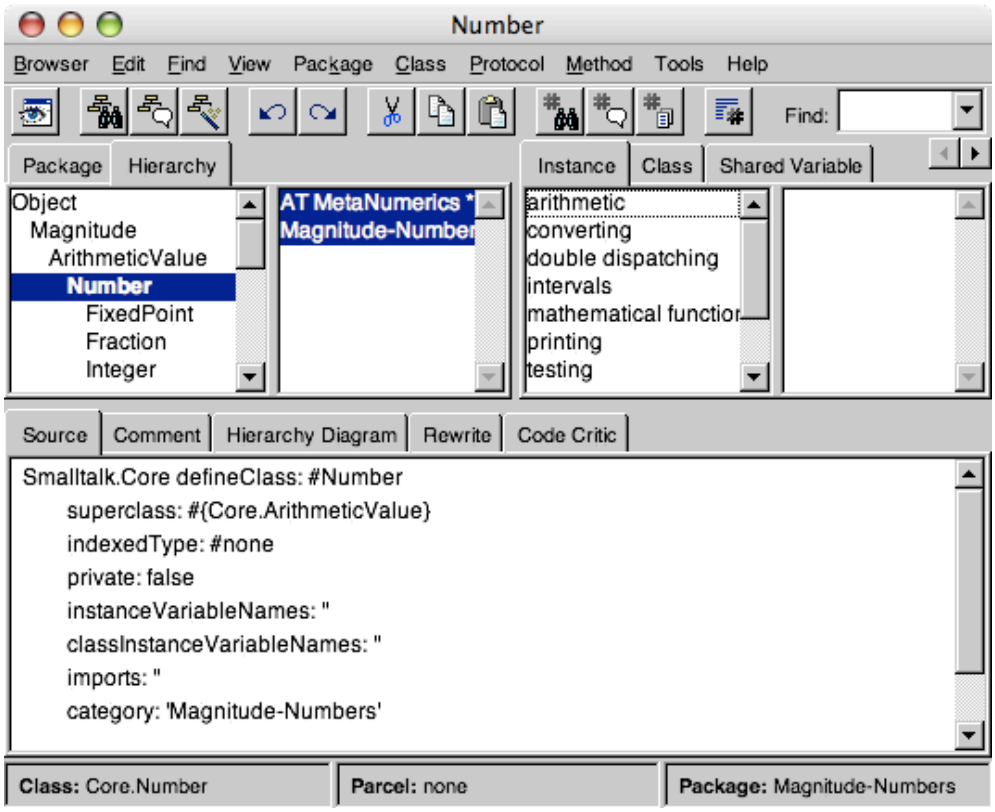



Figure 6.8: A Smalltalk refactoring browser on `Root.Smalltalk.Core.Number)`.

explicitly in the ConceptBase. Moreover, when saving a conceptified concept, only the extensional part of its definition is stored. This can be seen by opening such a concept in the basic concept editor, since it will only show the extensional slots.

### 6.2.2 Querying Concepts by Accessing Slots

The slot accessing mechanism has already been described in section 6.1.3. Since there is no difference between a domain level concept and a conceptified concept, we will just show a number examples that illustrate the slots that are available after conceptification:

```
d1:= Concepts.Number.

d1 fullyQualifiedReference.
⟶ a binding reference: #{Root.Smalltalk.Core.Number}

d1 hasPreferredLabel.
⟶ a terminal for the string: 'Root.Smalltalk.Core.Number'

d1 toStarBrowser.
⟶ opens the Root.Smalltalk.Core.Number concept in
    the extended Star Browser (see figure 6.9)
```

As is shown in figure 6.9, the `Number` concept has the same structure as a domain concept. The relationship names that are prefixed with '`vw`' are generated from the implementation by the conceptification process. Depending on the concept granularity that was defined in the CoBro kernel, these will either refer to concepts or terminals. If a class is conceptified then it will have a `superconcept` link to `Concepts.SmalltalkClass`. This indicates that part of the concept is computed. The `vw-superclass` slot that is generated refers to the (conceptified) superclass of `Number`, namely `ArithmeticValue`. These references are only computed on demand. In the following we demonstrate querying the superconcept and superclass chain:

```
Concepts.Number superconcept.
⟶ an ordered collection containing SmalltalkClass
Concepts.Number super.
⟶ a concept: SmalltalkClass
Concepts.Number allSuperconcepts.
⟶ an ordered collection containing
    SmalltalkClass, Concept, Entity
```

```
Concepts.Number vwHasSuperclass.
⟶ a concept: Root.Smalltalk.Core.ArithmeticValue
Concepts.Number vwHasSuperclass
                vwHasSuperclass
                vwHasSuperclass.
⟶ a concept: Root.Smalltalk.Core.Object
```

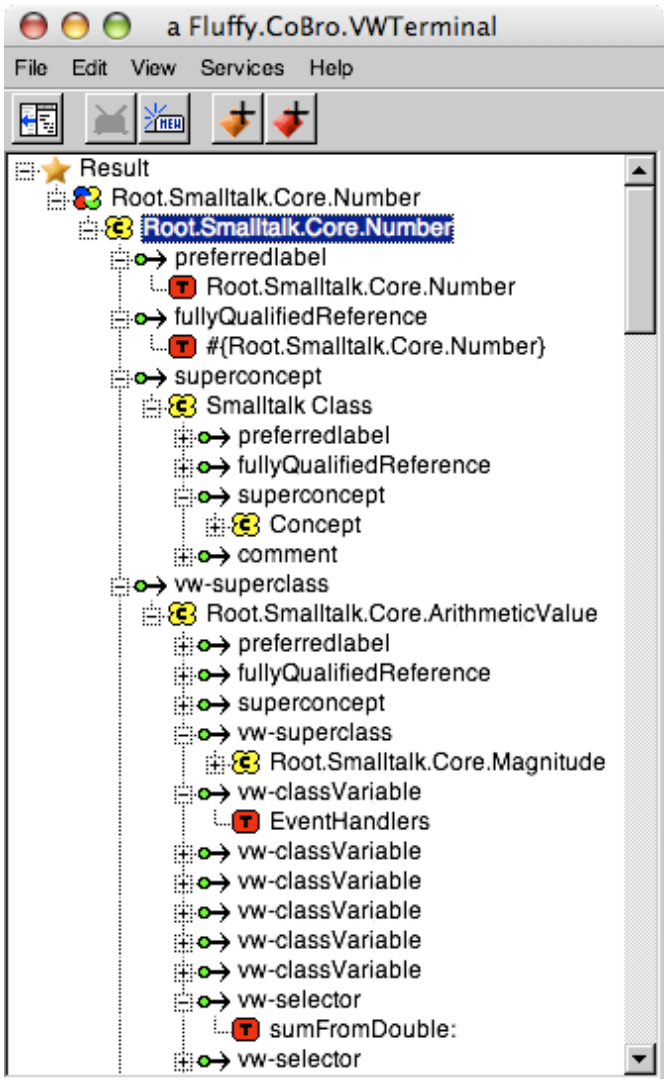



Figure 6.9: An extended Star Browser on `Concepts.Number`

```
Concepts.Number allSuperclasses.
⟶ an ordered collection containing
    Root.Smalltalk.Core.ArithmeticValue,
    Root.Smalltalk.Core.Magnitude,
    Root.Smalltalk.Core.Object
Concepts.Number withAllSuperclasses.
⟶ an ordered collection containing
    Root.Smalltalk.Core.Number,
    Root.Smalltalk.Core.ArithmeticValue,
    Root.Smalltalk.Core.Magnitude,
    Root.Smalltalk.Core.Object
```

The latter two (helper) messages were added to the meta-level interface of the CoBro kernel. This is because they do not follow the default behavior of slot accessor messages (by default they return a slot value or a collection of values). As a consequence you need to override the default behavior. This is similar to the way `superconcept` was added to the meta-level interface.

### 6.2.3 Coupling Domain and Concept Level Concepts

Up to this point we have described how to get a concept version of a class. In the following examples we will show that it is possible to relate it to domain level concepts, code level concepts, or plain annotations such as comments. The first example illustrates how to set up a `relatedTo` relationship that is consequently used to hook up the conceptified `Number` class in a concept network:

```
(Concepts new:  #{relatedTo})
       hasPreferredLabel:  'related to' ;
       superconcept:  Concepts.Relationship ;
       multiplicity:  '#(1 #n)' ;
       allowedDestinations:  '#(#{Concept})' ;
       comment:  'A generic association' ;
       save.
```
⟶ *a concept:* ***Concepts.relatedTo***

```
Concepts.Number
       comment:  'I think this class is used
                  for counting' ;
       relatedTo:  Concepts.Wine ;
       relatedTo:  Concepts.OrderedCollection ;
       save.
Concepts.Number toGraphBrowser.
```
⟶ *opens* ***Concepts.Number*** *in* CoBro-Nav *(see figure 6.10)*

Of course it is also possible to connect a domain level concept to a a code level concept as well. Adding slots to a code level entity can be done by using the integrated editor tool in the refactoring browser. A slot can be added by typing the slot message, e.g. `comment:  'The CoBro launcher tool'`, and pressing the add-button. What happens is that the message is evaluated in the context of the conceptified class entity. In screenshot 6.11 we illustrate the addition of

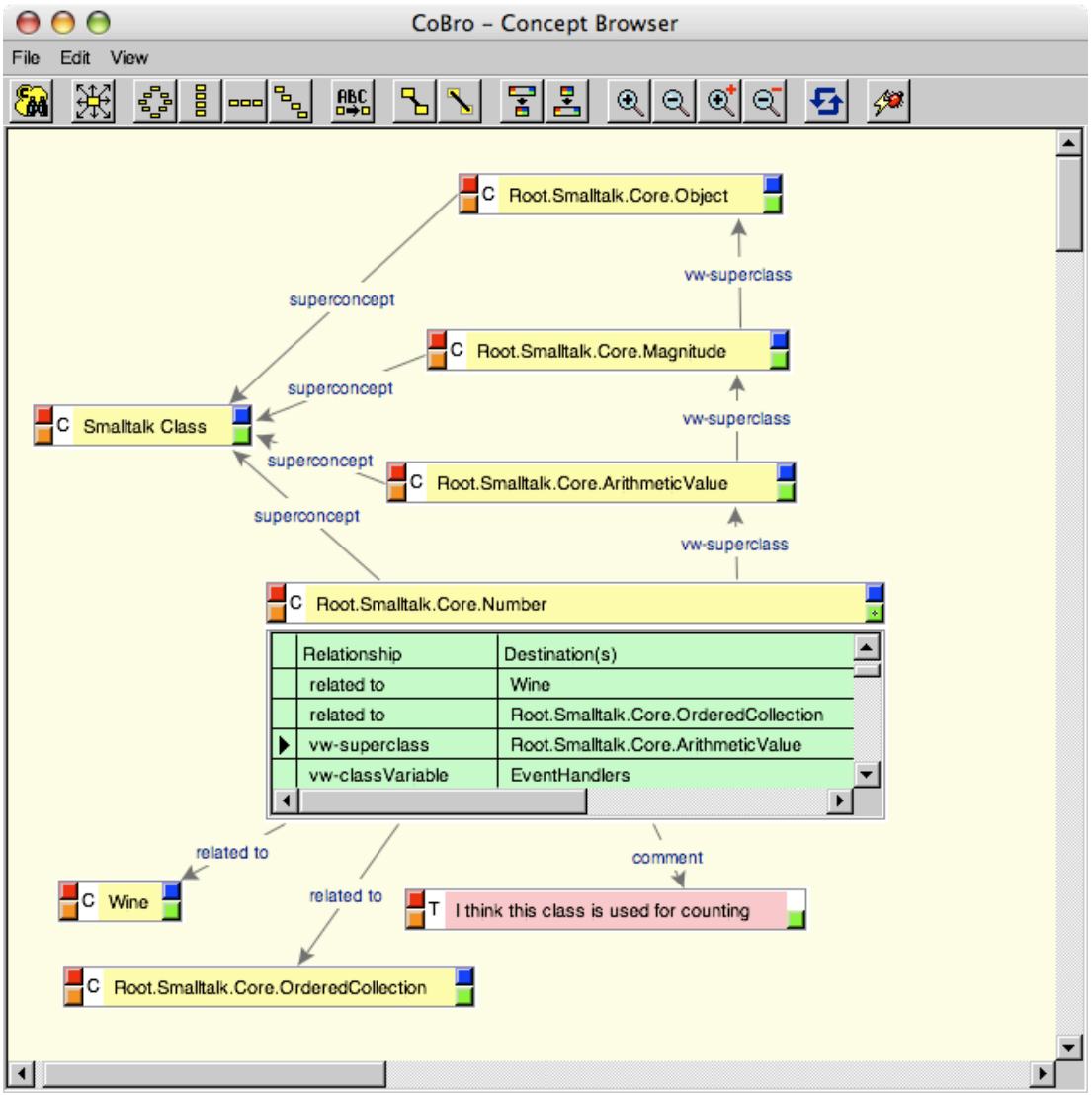



Figure 6.10: CoBro-Nav on the hooked up `Number` class

an `image` slot to the concept representation of the class. The value for the slot is the filename of the png-image. Screenshot 6.12 shows the corresponding concept network in the CoBro-Nav tool that is also integrated in the refactoring browser. We discuss the extension of CoBro-Nav to support images in chapter 7.

### 6.2.4 Active Use of Concepts in the Implementation

Another important aspect we need to show is how concepts can be used actively from within the implementation. We have already illustrated how concepts can be manipulated using CoBro-CML. These statements can be used in Smalltalk

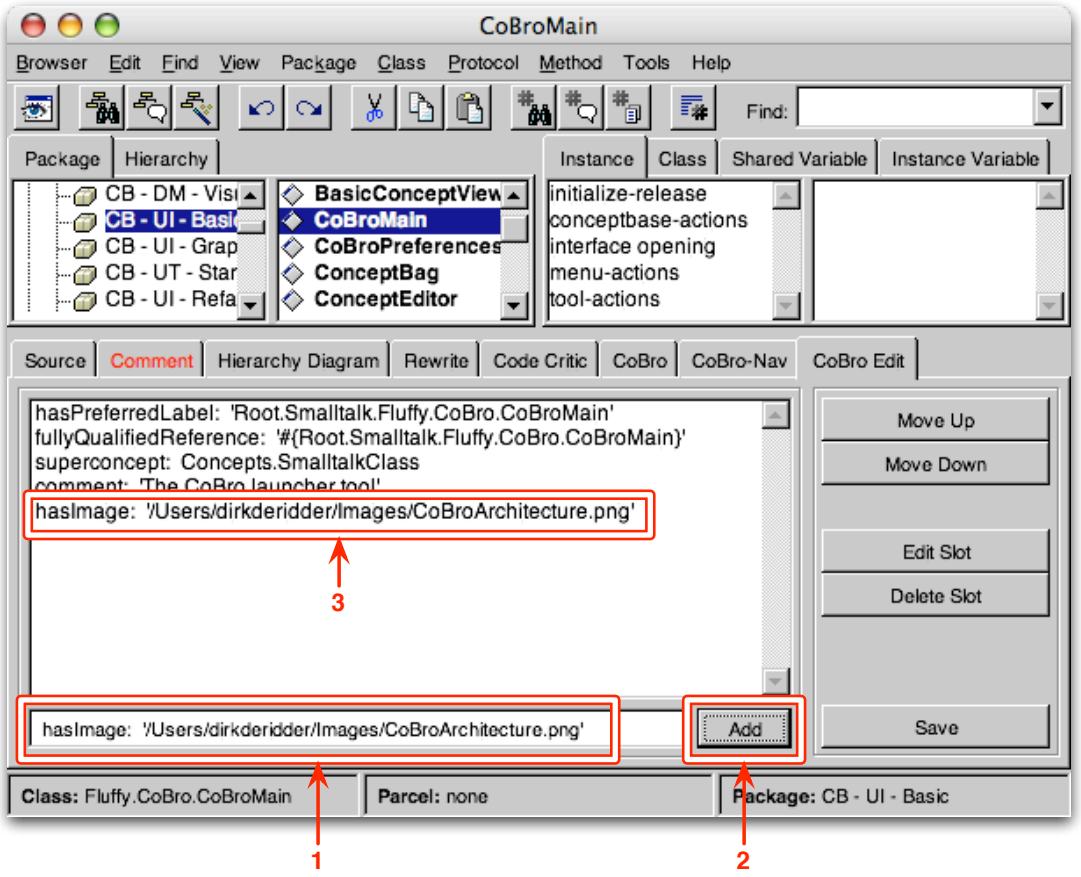



Figure 6.11: Adding an image to the `CoBroMain` class concept with the refactoring browser edit tool.

method bodies as well. As a matter of fact, certain parts of the CoBro environment were implemented this way. In the following examples we will show a number of method bodies that use elements from the ConceptBase to fulfill their task[8].

The first example illustrates a helper method on the `ConceptNameSpace` that dumps the definition of a collection of concepts. This is useful to export concepts from the ConceptBase, for instance:

[8]In the examples we use `ClassX>>messageA` to indicate an instance side method, and `ClassX class>>messageA` to indicate a class side method

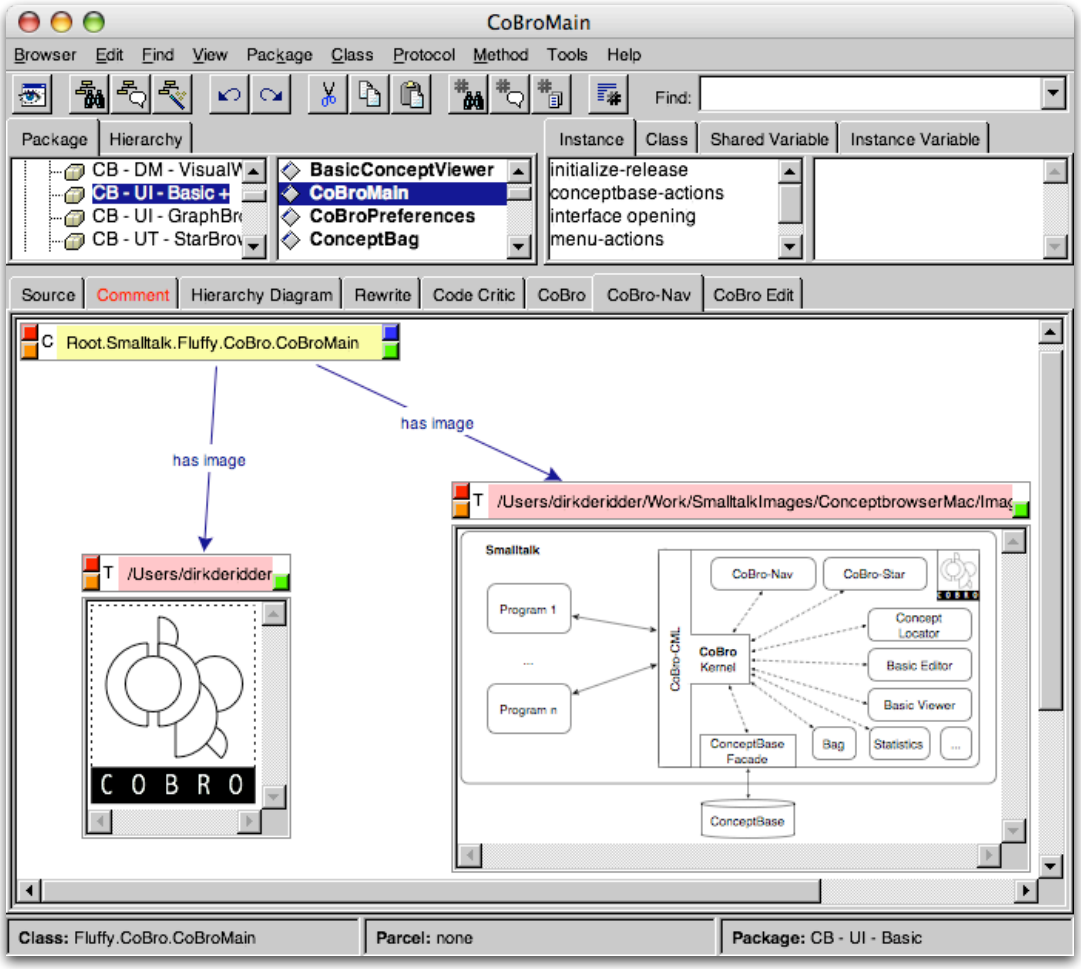



Figure 6.12: Visualising the `CoBroMain` class concept in the refactoring browser CoBro-Nav tool.

```
ConceptNameSpace>>dumpConcepts:  conceptCollection
    | result |
    result := conceptCollection collect:
                [:each | each smalltalkDefinition].
    result := (result fold:
                [:a :b | a, '\\', b]) withCRs.

    (Workbook new replaceAllTextPagesWith:
                (WorkspacePage labeled:  'Concept Dump'
                                with:  result)) open.
```

⟶ *This method will open a Smalltalk workbook on the definitions of a collection of concepts. For example:*

```
    Concepts dumpConcepts:
                Concepts.MyFavoriteWines isGroupFor
```

In the following we show the implementation of the actual method that caches a set of (core) concepts to speed up the CoBro kernel. If you want CoBro to cache your own concepts, simply add them to the group concept `EntitiesToCacheGroup` by sending the message `isGroupFor:`[9]:

```
ConceptNameSpace>>cacheCoreConcepts
    | conceptsToCache |
    conceptsToCache :=
           Fluffy.Concepts.EntitiesToCacheGroup isGroupFor.
    conceptsToCache do:
           [:each |
                each definition.  "disable lazy loading"
                self at:  each fullyQualifiedReference
                     put:  each].

⟶ This method will cache the concepts in the group
    EntitiesToCacheGroup
```

Another example in which the domain knowledge stored in the ConceptBase is used is shown by the `manages` method of the class `Manager`. The assumption made by the domain model is that a manager cannot manage more than five employees. So instead of hardcoding this information in the code level, we escape to the concept level to find it:

```
Concepts containsAllConcepts:  #(#{Manager} #{TeamMember}
                                  #{manages}).
⟶ true
Concepts.Manager manages.
⟶ a concept: Concepts.TeamMember
Concepts.manages multiplicity.
⟶ an ordered collection representing the multiplicity: #(1 5)
```

[9]Note that for performance reasons, the definition of a concept is only loaded if it is accessed. So it is necessary to load the definition before the concept is stored in the cache.

```
Manager>>manages:  aTeamMember
    self manages size <
                (Concepts.manages multiplicity at:  2)
         ifTrue:  [self manages add:  aTeamMember]
         ifFalse:  [Dialog warn:
                              'This manager cannot
                               manage extra team members !']
```

⟶ *This method will complain if you add a team member and the instance already manages too many people*

The implementation above clearly assumes the existence of certain concepts in order to function properly. As was illustrated you can test the existence by using the `containsAllConcepts` message. Another way is to have a concept-setup and concept-teardown message on the class side. Such methods will take care of setting up the necessary concepts. An excerpt of both methods for the manager example is shown below:

```
Manager class>>setup
    (Concepts new:  #{Manager})
       hasPreferredLabel:  'Manager' ;
       superconcept:  Concepts.Concept ;
       comment:  'A manager manages team members.'  ;
       save.
...
    (Concepts new:  #{manages})
       hasPreferredLabel:  'manages' ;
       superconcept:  Concepts.Relationship ;
       multiplicity:  '#(1 5)' ;
       allowedDestinations:  '#(#{TeamMember})' ;
       save.
    ...
    Concepts.Manager manages:  Concepts.TeamMember ; save.
    ...
```

```
Manager class>>teardown
    Concepts delete:  Concepts.Manager.
    Concepts delete:  Concepts.TeamMember.
    ...
    Concepts delete:  Concepts.manages.
```

As we have shown in the previous section, it is possible to connect a class to another class at the concept level. In order to be able to switch to the concept level you can use the `asConcept` message. The `asConcept` message allows you to write a statement that looks up the concept version of a class. This provides access to a path at the concept level that is not available at the code level. For instance in the `setup` method of the `Manager` class we connect the `Manager` class to the existing `Manager` concept as follows:

```
Manager class>>setup
...
    self asConcept implements:  Concepts.Manager ; save.
...
```

⟶ *adds an* ***`implements`*** *slot to the*
***`Root.Smalltalk.Playground.Manager`*** *concept.*
*The destination of the slot is the* ***`Manager`*** *concept (see figure 6.13)*

The corresponding concept network is shown in figure 6.13.

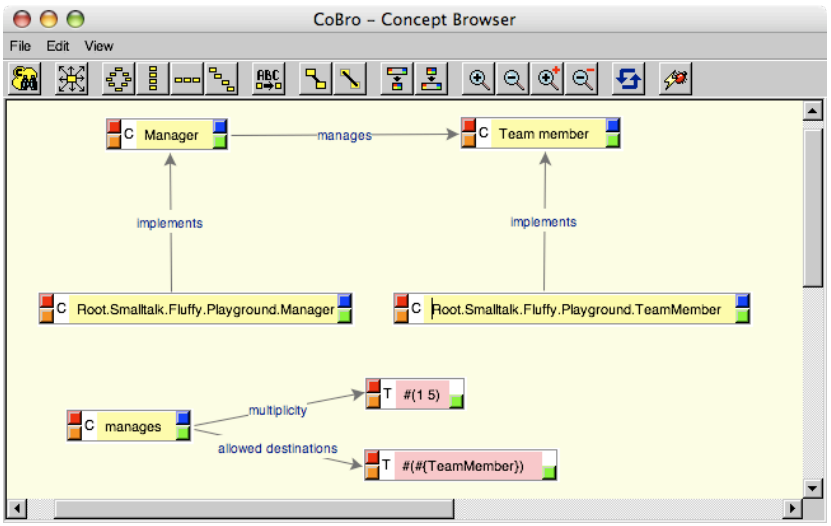



Figure 6.13: CoBro-Nav illustrating the manager example

## 6.3 Summary

In this chapter we have illustrated the interaction with CoBro in an example-based fashion. In the first part we limited the discussion to concepts that come from the domain level. The examples we covered clearly illustrated how to manipulate domain concepts in CoBro-CML in a variety of ways. This demonstrated that domain knowledge can indeed be made *explicit* in CoBro.

The second part focused on concepts that stem from the code level. In essence this showed how conceptified code level entities can be connected to the domain knowledge and vice versa. This illustrated that the domain knowledge and the implementation can also be *coupled*.

By implementing CoBro in symbiosis with Smalltalk we made it possible to manipulate objects and concepts in an identical way. This was shown throughout the different examples where we manipulated concepts from a standard Smalltalk workspace, or from within method bodies. This illustrated that manipulating concepts can be done *transparently* for the programmer.

Moreover these examples in the second part showed an interaction between standard Smalltalk code and the concepts in the ConceptBase. This illustrated that the concept level can participate *actively* in the implementation.

We have also demonstrated the use of different tools and helper messages that support the programmer in working with the concept level. These were all built to support a highly interactive way of manipulating the system similar to the way you program in the Smalltalk environment.

In the following chapter we will demonstrate a number of advanced applications of CoBro.

# Chapter 7

# CoBro Applied

The discussion in this chapter is divided into two main sections. In the first part we illustrate a number of applications of CoBro in practice. We base the discussion on a small Media Library case study that we use to highlight the different aspects introduced in the previous chapters. We will show how CoBro is used to set up a malleable implementation that is documented by the explicit concept level. In addition this demonstrates the active interplay between entities at the code level and entities at the concept level. We also illustrate how CoBro can be extended to support a developer when writing malleable code. This also illustrates how small adaptations to the ConceptBase enable the use of CoBro within other contexts than originally intended.

The second part focuses on our work performed in the context of research projects in collaboration with industry. It provides insights in how the concept-centric environment was conceived over the years and how CoBro builds upon the experience gained by its (several) predecessors.

## 7.1 CoBro: The Power of Malleability

The running example for this section is a media library system (`MediaLibrary`) that can be used to manage a collection of media items (`MediaItem`). The different media items that are supported by the library are games (`Game`), digital video discs (`DVD`), compact discs (`CD`), and books (`Book`). Depending on the configuration of the library, the available media item choices can vary (`itemChoices`). For each of the media items, a corresponding user interface is available that enables a user to record information for an item (`windowSpec`). The class diagram for the media library example, together with the user interfaces for the media items, is shown in figure 7.1.

The main user interface of the media library contains a dropdown list from which a user can select a media item type to add. Depending on what was selected

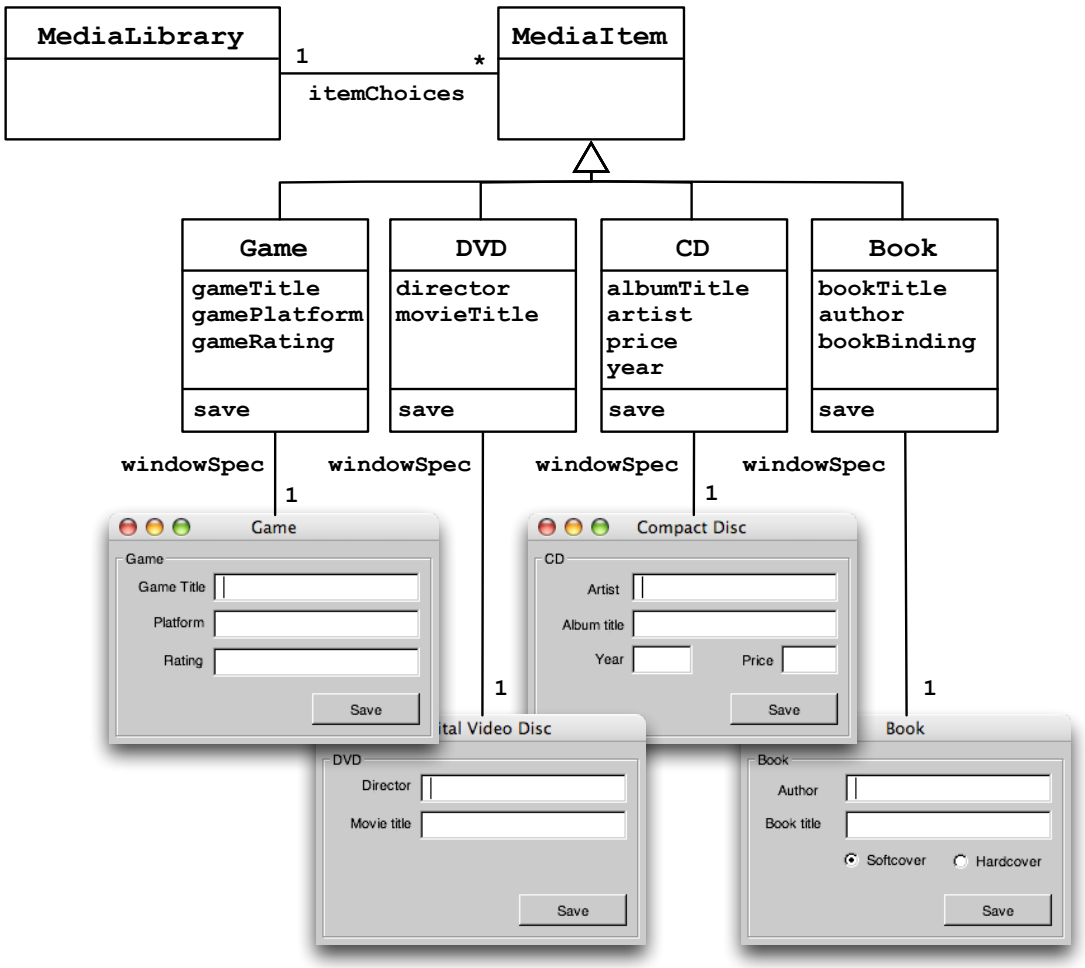



Figure 7.1: Media Library class diagram.

in the list, we open the corresponding user interface within a subcanvas of the main window. This is illustrated by the screenshot in figure 7.2.

In the following section we illustrate a basic implementation that makes no use of CoBro. Next we evolve this implementation into a malleable one by creating configurations at the concept level. We use this extension to illustrate the interplay between the code level and the concept level. Subsequently we extend the implementation with a notion of library users that require different user interfaces according to their level of expertise. This illustrates the active participation of the concept level during the execution of the system. Moreover it also shows how domain knowledge is made explicit at the concept level. The last addition to the library example sets up a validation mechanism. This can be used by developers to verify whether a configuration definition contains no conflicts. We end with two sections where we discuss how the CoBro environment can be

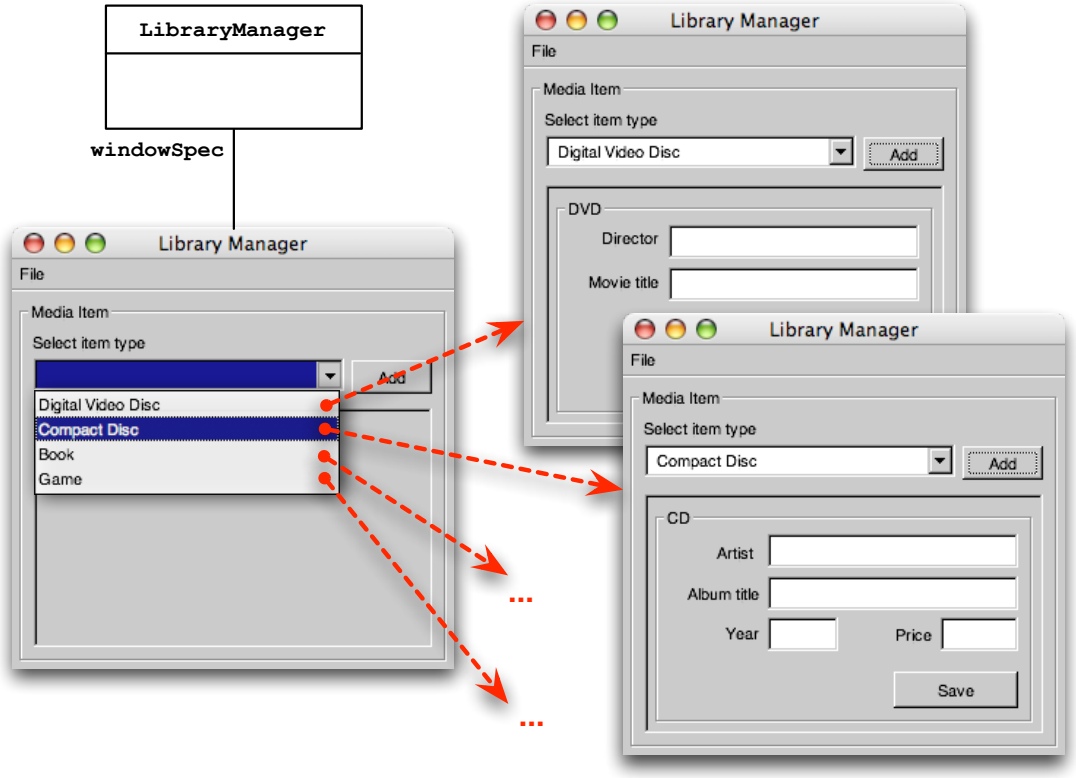



Figure 7.2: Media Library main window.

extended in different ways.

### 7.1.1 Media Library: Standard Implementation

In this section we discuss a standard implementation of the media library system. In the next section we refactor this implementation so it makes use of the concept level. As we will see, this results in a malleable implementation that is customisable at the concept level.

There are several valid implementations that can provide the behavior for the media library application. One approach is to store the classnames for the `itemChoices` in a list. Subsequently this list can be used to provide the choices that are available to a user. This is illustrated by the following method body:

```
initializeItemChoices
    self itemChoices:  List new.
    self addChoice:  'Game'
         value:  Game.
    self addChoice:  'Digital Video Disc'
         value:  DVD.
    self addChoice:  'Compact Disc'
         value:CD.
    self addChoice:  'Book'
         value:Book.
```

Additionally we also need to take care of opening the user interface for the selected media item. Opening the user interface for the selected item type requires three steps which are triggered when a user clicks the add-button. First of all we ask the corresponding media item class for its `windowSpec`. Secondly we create an instance of the media item which serves as the application model. Thirdly, we install the combination of both in the subcanvas of the main library window. These three steps are illustrated in the body of the `addItem` method:

```
addItem
    | app spec |
    spec := self selectedItem
                        interfaceSpecFor:  #windowSpec.
    app := self selectedItem new.
    (self widgetAt:  #subcanvas) client:  app spec:  spec.
```

At this point the media library is able to switch the user interface according to the selected media item type. Changing the media item choices that are available to a user is done by editing the code of the `initializeItemChoices` method.

Now suppose that we want to sell the media library system to different customers. For each customer we need to change the available choices according to the desired configuration. One way to achieve this is to include a conditional statement in the `initializeItemChoices` method. The conditional checks the current customer configuration and sets up one of the predefined lists:

```
initializeItemChoices
    self itemChoices:  List new.

    self mediaConfiguration = #MediaConfigurationA
     ifTrue:  [
        self addChoice:  'Game'
             value:  Game.
        self addChoice:  'Digital Video Disc'
             value:  DVD.
        self addChoice:  'Compact Disc'
             value:CD.
        self addChoice:  'Book'
             value:Book ].

    self mediaConfiguration = #MediaConfigurationB
     ifTrue:  [
        self addChoice:  'Digital Video Disc'
             value:  DVD.
        self addChoice:  'Book'
             value:Book ].
```

Even though this is a feasible implementation, it does not scale to a large number of possible configurations. First of all the conditional would become very difficult to maintain due to its length. Secondly if we need to change for example the label of a particular media item (indicated by the `key:`), then we have to update each occurrence in the `itemChoices` method.

Therefore we will refactor the implementation to make the available `itemChoices` no longer hardcoded in the implementation. There are several ways to refactor the code into a more generic implementation that provides support for different configurations. For example an abstract factory design pattern can be implemented that takes care of the different configurations. Even though such a solution is perfectly acceptable, as a developer you will need to invest time in setting up the necessary infrastructure to accommodate this. Moreover the resulting malleable implementation will contain less explicit code that is more difficult to understand.

In the following we discuss how this can be done by representing the configuration information at the concept level. Besides illustrating the use of the up/down mechanism between the code level and the concept level, this also shows the improved malleability of the implementation. In addition the malleable implementation is documented in the associated concept network.

### 7.1.2 Media Library: Configurations at the Concept Level

In this section we refactor the basic implementation of the media library. We will make the implementation more generic so as to accomodate different configurations.

In order to be able to represent different configurations at the concept level we need to create a number of relations and concepts to accomodate such a configuration. This boils down to creating concepts for the domain layer of the ontology. After this step we can use the concepts in the domain layer to describe concepts in the application layer of the ontology.

First we create a `mediaConfiguration` relation that is used to connect the `MediaLibrary` class to a configuration at the concept level. Next we create a `supportsMediaItems` relation that is used to connect a configuration to the media items it supports:

```
(Concepts new: #{mediaConfiguration})
    hasPreferredLabel: 'media configuration';
    superconcept: Concepts.Relationship;
    multiplicity: '#(1 1)';
    allowedDestinations: '#(#{Concept})';
    save.

(Concepts new: #{supportsMediaItems})
    hasPreferredLabel: 'supports media items';
    superconcept: Concepts.Relationship;
    multiplicity: '#(1 n)';
    allowedDestinations: '#(#{SmalltalkClass})';
    save.
```

Subsequently we can use both relations to create a `MediaConfigurationA`, and a `MediaConfigurationB` concept. These concepts respectively represent the configurations for customer A and customer B. Customer A has access to `DVD`s, `CD`s, `Book`s and `Game`s. Customer B has access to `DVD`s and `Book`s. The corresponding CoBro-CML statements for this are:

```
(Concepts new: #{MediaConfigurationA})
    hasPreferredLabel: 'Media Configuration A';
    superconcept: Concepts.Concept;
    comment: 'Configuration for customer A.';
    supportsMediaItems: Concepts.DVD;
    supportsMediaItems: Concepts.CD;
    supportsMediaItems: Concepts.Book;
    supportsMediaItems: Concepts.Game;
    save.

(Concepts new: #{MediaConfigurationB})
    hasPreferredLabel: 'Media Configuration B';
    superconcept: Concepts.Concept;
    comment: 'Configuration for customer B.';
    supportsMediaItems: Concepts.DVD;
    supportsMediaItems: Concepts.Book;
    save.
```

Note that the CoBro environment provides concept editors that can be used to define these concepts instead of manually writing the CoBro-CML definitions.

The alternative labels that are used to present the media items in the drop-down list are attached to the conceptified media item classes. The following shows this, as well as the final step where the `MediaLibrary` class is connected to a particular configuration concept:

```
(Concepts new: #{prettyLabel})
    hasPreferredLabel: 'pretty label';
    superconcept: Concepts. Relationship;
    multiplicity: '#(1 1)';
    allowedDestinations: '#(#{String})';
    save.

Concepts.DVD prettyLabel: 'Digital Video Disc'; save.
Concepts.Book prettyLabel: 'Book'; save.
Concepts.CD prettyLabel: 'Compact Disc'; save.
Concepts.Game prettyLabel: 'Game'; save.

Concepts.MediaLibrary
    mediaConfiguration: Concepts.MediaConfigurationA;
    save.
```

This concludes the setup of the concept level to support media item configurations. As shown by these examples we did not need to set up our own code-level infrastructure to support the meta-level for the configurations (e.g. by implementing a number of analysis and design patterns). The work presented up to this point was limited to 'configuring' the meta-level (in this case the concept level).

Changing the supported media items of a configuration is done by editing the corresponding configuration concept. New configurations can be added at will. Changing the configuration of the media library is done by connecting it to a different configuration concept.

Now that we have the necessary elements at the concept level, we can refactor the `initializeItemChoices` method so it looks up the configuration at the concept level (CoBro-CML shown in red):

```
initializeItemChoices
    self itemChoices:  List new.
    self class
           asConcept
           mediaConfiguration
           supportsMediaItems
             do:  [:each | self addChoice:  each prettyLabel
                                  value:  each ]
```

An abstract schema of the corresponding interplay between the code level and the concept level is given in figure 7.3. As shown, the class is 'upped' to the concept level with the statement `self class asConcept` (1). Next we consult the `mediaConfiguration` slot, which returns the `MediaConfigurationA` concept (2). Consequently we can ask the `supportsMediaItems` slot of this concept, which returns a collection of conceptified classes (i.e. the ones we added earlier to the configuration concept) (3). Each of the media item concepts are subsequently added to the available choices in the dropdown list. The `prettyLabel` slots we attached earlier to these conceptified classes are used to present the choices in the dropdown list.

The only thing left to do is to adapt the `addItem` method. As shown in the following code, the only difference with the previous version is that we send the message `asSmalltalk` to the selected item. This 'downs' the conceptified media item class to its corresponding code level representation (4). As a result we can ask it for the necessary information in order to install the user interface in the main window's subcanvas. This way we obtain a code level connection between for example the `MediaLibrary` class and the `Game` class (5) (CoBro-CML shown

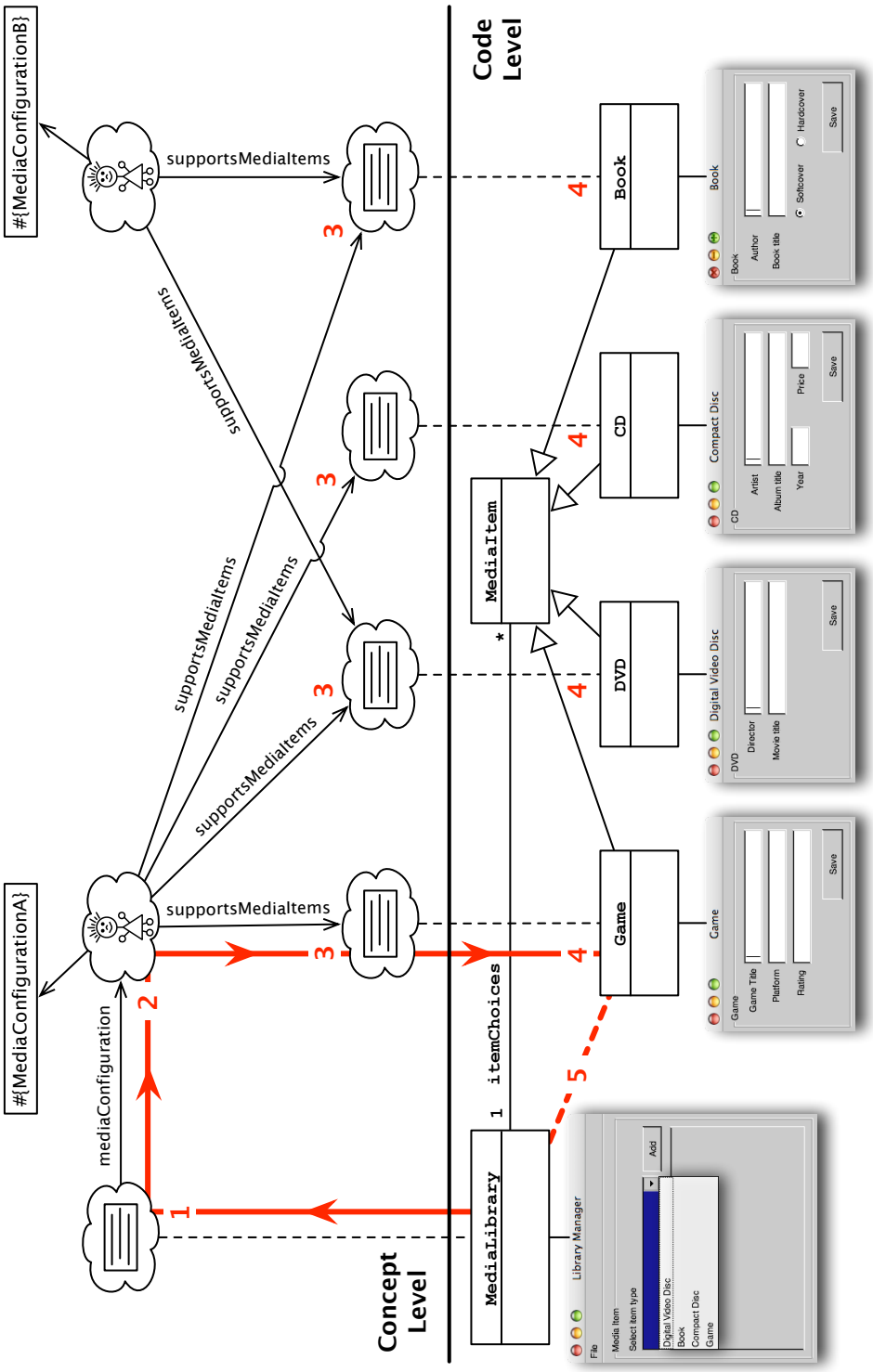



Figure 7.3: Media Library with concept level configuration.

in red):

```
addItem
    | app spec |
    spec := self selectedItem asSmalltalk
                interfaceSpecFor:  #windowSpec.
    app := self selectedItem asSmalltalk new.

    (self widgetAt:  #subcanvas) client:  app spec:  spec.
```

In figure 7.4 we show a screenshot of the refactoring browser in which we have explored the live concept network that corresponds to the abstract schema in figure 7.3.

In the next section we will show how we can evolve the implementation in order to accommodate different user interfaces according to the current user of the library system. We do so by invoking the concept level in order to provide the necessary information for the code level. This improves the malleability of the corresponding implementation. Even though the resulting implementation contains less explicit code, the domain knowledge is still available to the developer in the explicit concept level.

### 7.1.3 Media Library: Adding a Notion of Users

In this section we extend the media library with a notion of media library users (`MediaLibraryUser`). The information we want to record about a user is limited to a name and age. The goal is to be able to show different media item user interfaces according to the current user's level of expertise, which is based on his age.

We foresee two kinds of users: a novice user and an expert user. We model these users at the concept level with the concepts `NoviceUser` and `ExpertUser`. A screenshot of the novice user interface (`specNoviceUser`) and expert user interface (`specAdvancedUser`) for the book media item is shown in figure 7.5.

The level of expertise for a user is based on the age category he or she is in. We provide the following age categories at the concept level: `Children` (age 0 - 14), `Youth` (age 14 - 24), `Adults` (age 25 - 64), and `Seniors` (age 65 - 150). By representing the age categories at the concept level, they can be customised for each customer. The corresponding concept definitions are shown by the following statements:

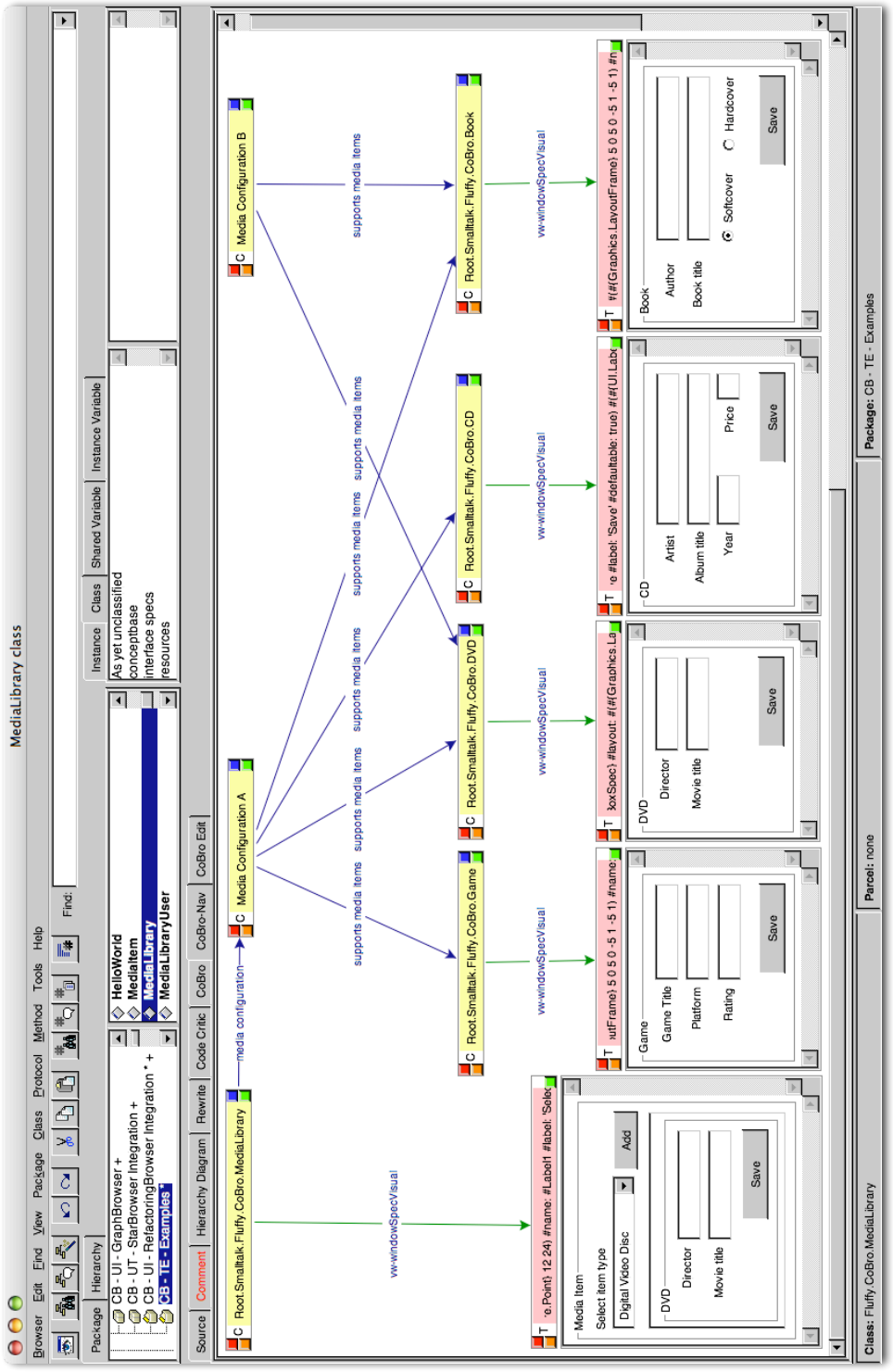



Figure 7.4: Media Library configuration in a Smalltalk Browser

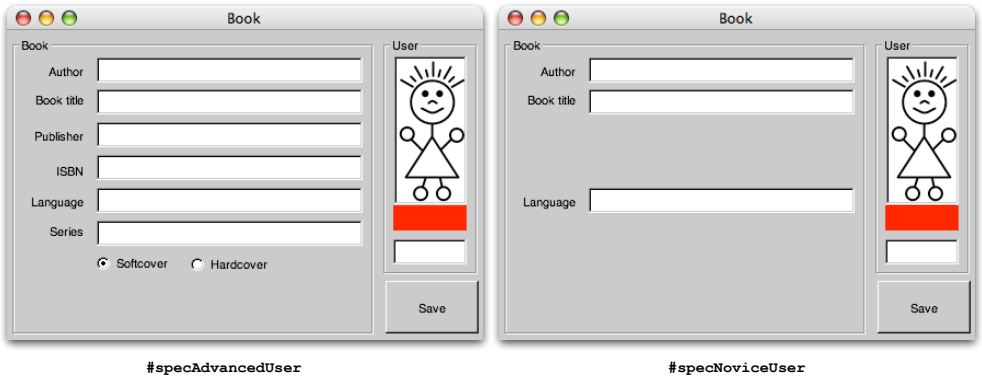



Figure 7.5: The novice and expert user interface for books.

```
(Concepts new: #{AgeCategory})
    hasPreferredLabel: 'Age category';
    superconcept: Concepts.Concept;
    hasLabel: 'Life cycle grouping';
    comment: 'Used to distinguish between categories
              of people according to their age.';
    save.

(Concepts new: #{ageRange})
    hasPreferredLabel: 'age range';
    superconcept: Concepts. Relationship;
    comment: 'e.g. #(1 20) indicates an age range
              from 1 to 20 years';
    multiplicity: '#(1 1)';
    allowedDestinations: '#(#{SmalltalkValue})';
    save.

(Concepts new: #{Children})
    hasPreferredLabel: 'Children age category';
    superconcept: Concepts.AgeCategory;
    ageRange: '#(0 14)';
    save.

...
```

Calculating the age category for a user is done by consulting the age ranges of the categories at the concept level (`Concepts.AgeCategory allSubconcepts`). If the user's age is in the `ageRange` of a category then the corresponding category

concept is returned (i.e. `Children`, `Youth`, `Adults`, or `Seniors`). The corresponding method body is written as follows (CoBro-CML shown in red):

```
ageCategory
    ^ Concepts.AgeCategory allSubconcepts
        detect:
            [:each |
            (self age >= each ageRange first) &
            (self age <= each ageRange second)].
```

Next we define the method body that returns the level of expertise for a user based on the age category he or she is in. For the current customer we classify `Children` and `Youth` as novice users, and `Adults` and `Seniors` as expert users. The corresponding method body is (CoBro-CML shown in red):

```
expertiseLevel
    ^ (self ageCategory is:  Concepts.Children) |
        (self ageCategory is:  Concepts.Youth)
            ifTrue:  [Concepts.NoviceUser]
            ifFalse:  [Concepts.ExpertUser].
```

In the following code excerpt we illustrate the use of both messages for two different users:

```
userA := MediaLibraryUser name:  'Wim' age:  12.
userB := MediaLibraryUser name:  'Linda' age:  32.

userA ageCategory.
⟶ Concepts.Children
userB ageCategory.
⟶ Concepts.Adults

userA expertiseLevel.
⟶ Concepts.NoviceUser
userB expertiseLevel.
⟶ Concepts.ExpertUser
```

Now that we have implemented the necessary elements for library users, it is time to set up the infrastructure for showing the correct user interface. First of all we

need to extend the `MediaLibrary` so that it knows the current user. Next we need to annotate the different media item classes with a slot that returns the name of the particular window specification for a kind of user. Consequently we can install this in the subcanvas of the main window. In order to set up a slot that holds a piece of Smalltalk code, you need to set the `allowedDestinations` of the relation to `SmalltalkValue`. In doing so, you can evaluate the block by sending the `value:` message with the current user as an argument. For example if the implementation of the `Book` class provides a user interface specification for an expert user (`#specAdvancedUser`) and a novice user (`#specNoviceUser`), then the slot annotation for the `Book` class is:

```
Concepts.Book
    specNameForUser:
         '[:user |
          (user expertiseLevel is:  Concepts.ExpertUser)
              ifTrue:  [#specAdvancedUser]
              ifFalse:  [#specNoviceUser]]';
save.
```

The same can be done for the other media item classes if they support different window specifications as well. Otherwise it suffices to add the following slot, which contains the default behavior, to the `MediaItem` class:

```
Concepts.MediaItem
    specNameForUser:
         '[:user | #windowSpec]';
save.
```

The final step consists of updating the `addItem` method that takes care of installing the user interface for the selected media item in the subcanvas of the main window. In addition to setting the current user for the instantiated application model, the hard coded reference to `#windowSpec` is replaced by a statement that evaluates the `specNameForUser` slot (changes shown in blue).

```
addItem
    | app spec |
    spec := self selectedItem asSmalltalk
                interfaceSpecFor:
                  (self selectedItem specNameForUser
                   value:  self currentUser).
    app := self selectedItem asSmalltalk new.
    app currentUser:  self currentUser.
    (self widgetAt:  #subcanvas) client:  app spec:  spec.
```

In figure 7.6 we present the library main window that shows different user interfaces for a book according to the current user of the system.

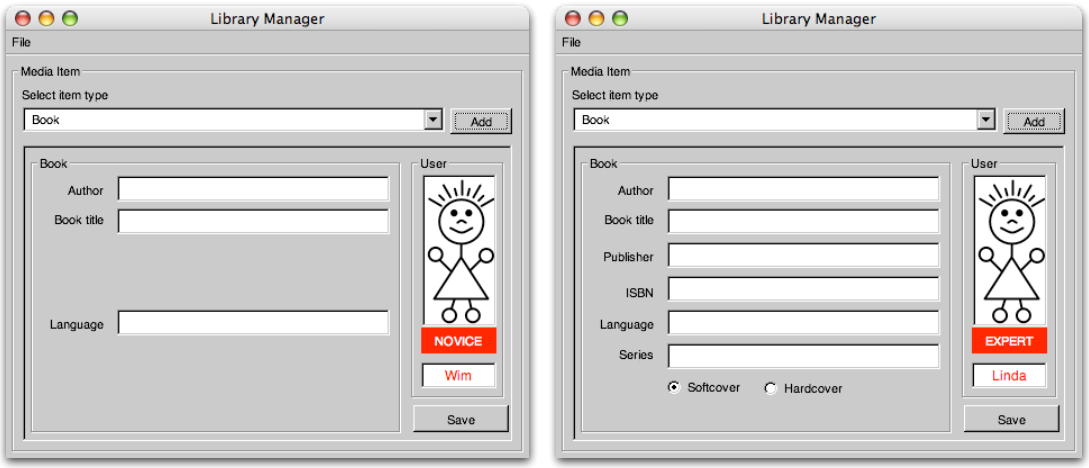
Changing the age ranges of the different age categories can be done by editing the corresponding concepts. Consequently it becomes possible to adapt the categorisation of users at the concept level.

The media library implementation can be made more generic by refactoring the `ageCategory` and `expertiseLevel` methods. Currently these methods are shared by all customers of the media library system. This means that if one customer wants to present a novice user interface to `Senior` users (whereas they now get an expert user interface), we need to update the corresponding method. This results again for example in the introduction of a conditional for each customer[1]. In such case, it is better to include the corresponding code as a Smalltalk block that is stored as a slot (e.g. `expertiseLevelDecision`) in the corresponding `MediaConfiguration` concept:

```
Concepts.MediaConfigurationA expertiseLevelDecision:
    '[:aUser | (aUser ageCategory is:  Concepts.Children) |
               (aUser ageCategory is:  Concepts.Youth)
                  ifTrue:  [Concepts.NoviceUser]
                  ifFalse:  [Concepts.ExpertUser]]';
    save.
```

The corresponding method body is consequently reduced to consulting the concept level and evaluating the block. As shown in the code excerpt below, this code is less explicit than the original version (i.e. the actual decision is no longer visible at the code level). Yet since we implemented it in a concept-centric environment, what has become implicit in the code as a result of a malleable implementation, can still be consulted interactively at the concept level. For example the

[1] Another implementation strategy would be to use subclassing and method overriding.




`MediaLibraryUser name: 'Wim' age: 12.` `MediaLibraryUser name: 'Linda' age: 32.`

Figure 7.6: Different user interfaces according to user expertise.

`expertiseLevel` method is reduced to the following (CoBro-CML shown in red):

```
expertiseLevel
    ^ MediaLibrary asConcept mediaConfiguration
        expertiseLevelDecision value:  self.
```

In the next section we illustrate how a validation mechanism can be set up. This enables a developer to verify the correctness of a configuration for a particular customer.

### 7.1.4 Media Library: Configuration Validation

Of course a developer wants to be able to verify that a media configuration only contains media items that are a subclass of `MediaItem`. This can be achieved by setting up a validation mechanism which is stored at the concept level. The reason to invoke the concept level for this is to make it possible to write different configuration validators for each kind of configuration. In addition this results in an active documentation of how the code should be used (or in this case how it should be instantiated for a particular customer).

In order to implement the validation mechanism we first need to create a concept that points to a `ConfigurationValidator`. The configuration validator is a terminal value that holds a Smalltalk block. This block accepts a particular configuration concept as its argument, and returns either an empty collection or

a collection that contains the conflicting elements. The corresponding relation definition is:

```
(Concepts new: #{configurationValidator})
    hasPreferredLabel: 'configuration validator';
    superconcept: Concepts.Relationship;
    multiplicity: '#(1 1)';
    allowedDestinations: '#(#{SmalltalkValue})';
    save.
```

The `configurationValidator` slot can be individually added to each configuration concept, or to a common superconcept. In the following CoBro-CML statements we have chosen to add a common superconcept that holds the slot and makes it available to all subconcepts. In this case, in order to have a valid configuration, we only require it to support media items that are a subclass of the `MediaItem` class:

```
(Concepts new: #{MediaConfiguration})
    hasPreferredLabel: 'Media Configuration';
    superconcept: Concepts.Concept;
    configurationValidator:
          '[:config | config supportsMediaItems reject:
                    [:each|each asSmalltalk
                              inheritsFrom: MediaItem]]';
    save.
```

Consequently we can use the validation slot to validate a configuration we have assembled. For example, verifying whether `MediaConfigurationA` is a correct configuration is achieved by the following statement:

```
Concepts.MediaConfiguration configurationValidator
          value: Concepts.MediaConfigurationA
```
⟶ *An empty collection*

If the configuration supported media items that didn't inherit from the `MediaItem` class, then these would have been returned in a collection.

The (default) `configurationValidator` slot can be overridden by adding it to the configuration that you want to validate with another Smalltalk block. Note

that the configuration slot acts as an explicit documentation of what is expected from a configuration by the code level.

In the next section we examine the extensibility of CoBro-Nav to support new node visualisations. We will use this flexibility to add a dedicated node that supports the validation of media configurations.

### 7.1.5 CoBro-Nav: Adding a New Node Visualisations

In this section we explore how CoBro-Nav can be extended to visualise concepts with new kinds of nodes. A node is a visual representation of a concept or terminal in the concept network drawn by CoBro-Nav.

Before we add a node for media configurations, we will describe the general principle. We illustrate this with the addition of a dedicated node that visualises the concepts that are the result of conceptification. This way we can provide support for a developer when navigating to code entities through the concept level. For example in the media library case it is useful if you can open the class of a supported media item from within CoBro-Nav. We also add support to the node to restrict the slots shown in the concept node to either the intension or the extension of the concept.

As already indicated in chapter 5, section 5.3, adding a new visualisation boils down to:

- the creation of a node class that can be instantiated by CoBro-Nav, and
- the addition of a `gbConceptViewClass` slot to the corresponding concept.

In figure 7.7 we present the extended user interface so as to represent a conceptified class. As shown it contains checkboxes for filtering the slots according to the intension or extension of the concept. For this purpose we reuse the existing functionality provided by the standard concept node to show the definition frame. All we need to do is filter the elements according to the value of the checkboxes. The new visualisation also supports a button that opens the corresponding class in a Smalltalk browser. This is implemented by opening the refactoring browser on the 'downed' concept. The user interface for the node is installed on the `CBViewConceptSmalltalk` class, which is also used as the application model.

Since by default all conceptified entities are represented as subconcepts of the `SmalltalkClass` concept, all that is left to do is add a `gbConceptViewClass` slot to it:

```
Concepts.SmalltalkClass
    gbConceptViewClass:  Concepts.CBViewConceptSmalltalk;
    save.
```

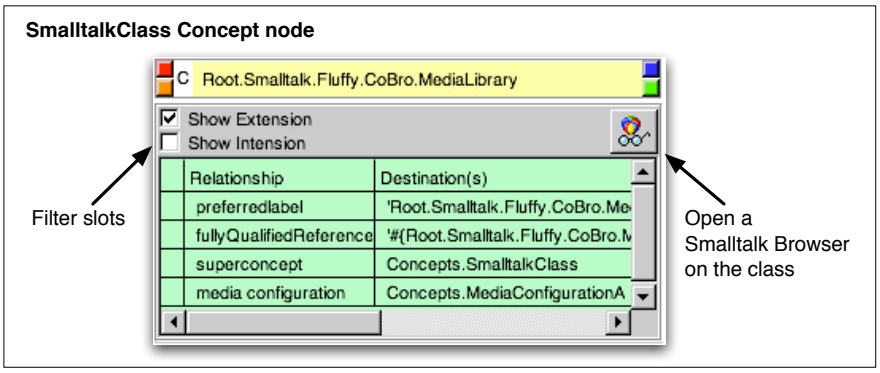



Figure 7.7: A dedicated CoBro-Nav node visualisation for deified classes.

This overrides the default `gbConceptViewClass` slot that is provided at the top level `Entity` concept.

The concept level is consequently used in the implementation of CoBro-Nav. As shown by the following code excerpt from the CoBro-Nav implementation, we instantiate the correct node according to the ConceptBase entity that needs to be visualised:

```
...
entityView := self cbEntity gbConceptViewClass
                                    asSmalltalk new.
...
```

This concludes the discussion of adding a special node for visualising a concept. Adding a dedicated node for terminals is done in a similar fashion. Yet an additional slot is required for terminals since these can be browsed *as a concept* or *as a value* in CoBro-Nav. Therefore terminals require an extra slot to indicate the class by which a terminal *value* should be visualised (`gbTerminalViewClass`). For example the `Image` terminal concept has both a `gbConceptViewClass`, and a `gbTerminalViewClass` slot. The former is used to display the concept. The latter refers to the way terminal values of this type are to be represented. The difference between both slots is shown in figure 7.8.

In a similar fashion other elements of CoBro-Nav can be customised according to the ConceptBase entity that should be represented. For example the default color and width of the arrow that represents a relation can be altered at the concept level. The slots responsible for this are `gbDefaultLineColor` and `gbDefaultLineWidth`. For example the `superconcept` relation overrides the default slots provided in the `Relationship` concept (i.e. a thick red line is used). This is illustrated by the following CoBro-CML statements.

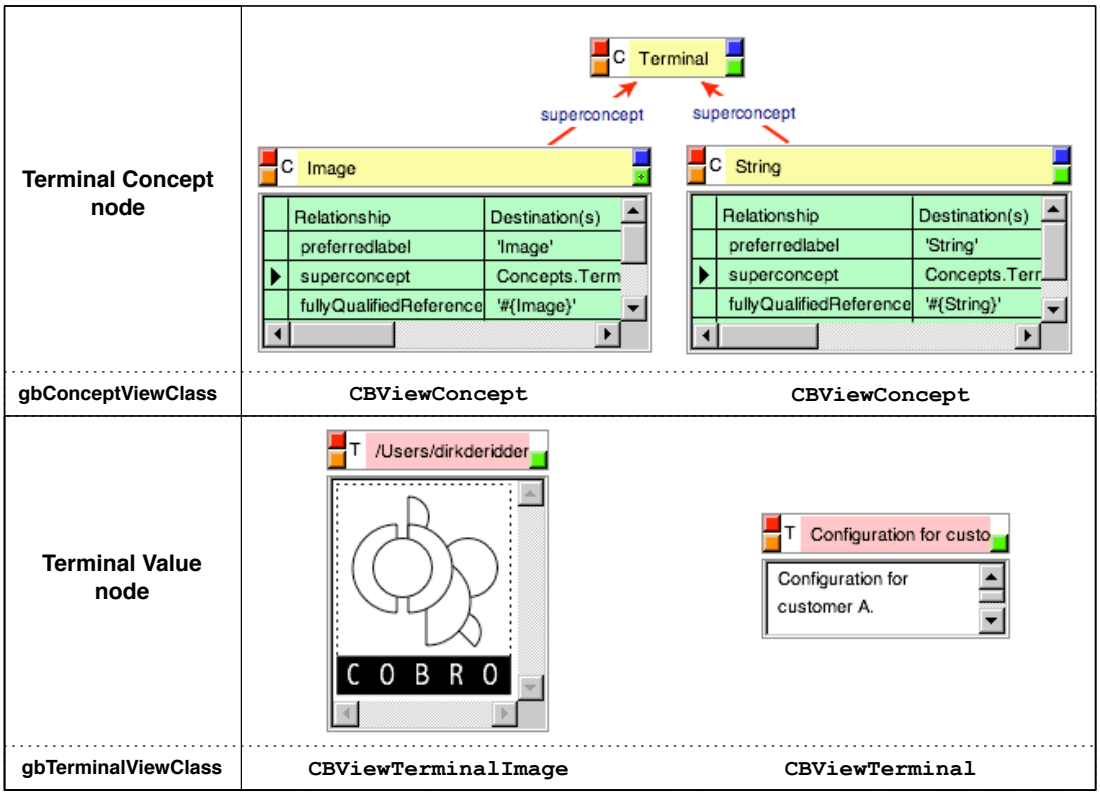



Figure 7.8: Nodes for a terminal concept and a terminal value.

```
Concepts.Relationship
    ...
    gbDefaultLineColor:  'ColorValue navy';
    gbDefaultLineWidth:  '1'.

Concepts.superconcept
    ...
    gbDefaultLineColor:  'ColorValue red';
    gbDefaultLineWidth:  '2'.
```

Now that we have shown the general principle of extending CoBro-Nav we will add a dedicated media library node (`CBViewMediaConfiguration`). Again, we take the existing node visualisation for concepts as a basis for the user interface. A button is added that can be pressed to validate the current configuration.

For the implementation of the method that is called when the button is pressed, we reuse the configuration validator infrastructure we introduced in the

previous section. If a configuration is valid, then we show a dialog box that says so. If a configuration contains conflicts, then we show a dialog box together with the conflicting elements in a concept bag. The corresponding method body is:

```
validateConfiguration
    | result |
    result := self cbEntity configurationValidator
                  value:  self cbEntity.
    result isEmpty
      ifTrue:
        [Dialog warn:  'This configuration is valid.']
      ifFalse:
        [result toConceptBag.
         Dialog warn:  'Invalid configuration'.]
```

Next we add the `gbConceptViewClass` slot to the `MediaConfiguration` concept:

```
Concepts.MediaConfiguration
    gbConceptViewClass:  Concepts.CBViewMediaConfiguration;
    save.
```

In figure 7.9 we show the node in action. We have opened it on a media configuration that contains a conflicting element. As shown in the node, the `Number` class was added as a supported media item. Since the configuration validator expects the supported media items to be subclasses of `MediaItem`, it rejects this element. As a consequence the `Number` concept is opened in a concept bag for further inspection.

In the following section we discuss value interpreters, which are used to interpret the value of a terminal. This makes it possible to compute the value of a terminal each time it is accessed.

### 7.1.6 CoBro: Adding Terminal Value Interpreters

In order to facilitate the integration of CoBro with other environments, we provide a notion of value interpreters. Value interpreters are Smalltalk blocks that are used to interpret a terminal value. Consequently they are stored as slots in the terminal concept which they apply to. In the following we show a number of already existing value interpreters:

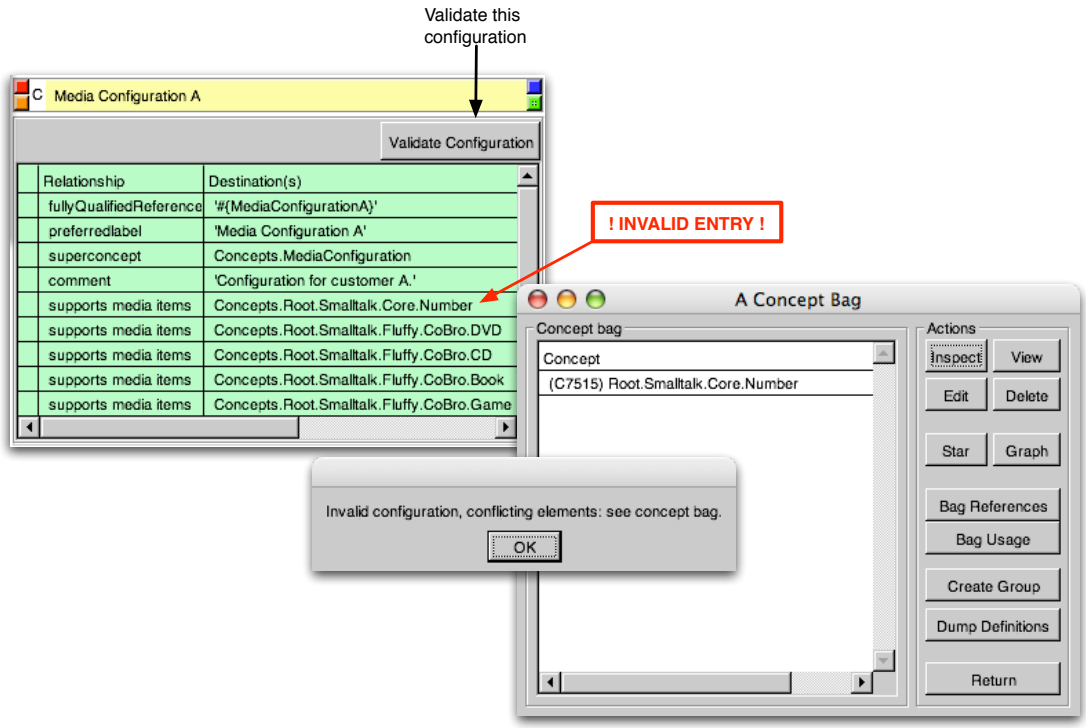



Figure 7.9: A dedicated node for media library configurations.

```
Concepts.Terminal cbValueInterpreter
⟶'[:aTerminal | aTerminal definition ]'
```
*Returns the value of the terminal as provided in its definition.*

```
Concepts.SmalltalkValue cbValueInterpreter
⟶'[:aTerminal | Compiler evaluate:
                        aTerminal definition ]'
```
*Returns the Smalltalk value for the terminal.*

```
Concepts.Image cbValueInterpreter
⟶'[:aTerminal | (ImageReader fromFile:
                        aTerminal definition) image ]'
```
*Returns an image for the value of the terminal as provided.*

The meta level interface of CoBro applies the `cbValueInterpreter` slot to the terminal value of a relation destination. So for example if you set the allowed destination for the `hasImage` relation to `#{Image}`, then the corresponding value interpreter is applied to the destination when it is returned. This is illustrated in

the following:

```
Concepts.HelloWorld hasImage: 'myWorldPicture.png' ; save.
Concepts.HelloWorld hasImage first.
```

⟶*Returns an image object for the file* **'myWorldPicture.png'**, *which is the result of applying the* **cbValueInterpreter** *of the* **Image** *concept on the terminal containing the string* **'myWorldPicture.png'**.

In the media library example this could be used to represent the destination of `ageRange` as a special kind of terminal. Currently the allowed destination for this relation was defined as `#{SmalltalkValue}`. As a consequence, providing the terminal value `'#(0 14)'` is returned as a Smalltalk array (since the Smalltalk compiler evaluates this string to an array). Hence it can be manipulated with the messages provided by the `Array` class (e.g. `first`, `at:`). Another option would be to introduce a dedicated terminal to represent age ranges. This allows us to provide a `cbValueInterpreter`, which converts the terminal value into an instance of an `AgeRange` class:

```
Concepts.AgeRange cbValueInterpreter:
    '[:aTerminal | AgeRange new: aTerminal definition ]'
    save.
```

The above illustrates that it is possible to relate concepts to terminals whose value is computed each time it is accessed. This could be used to represent the conceptification process at the concept level for example. In essence this boils down to storing the Smalltalk conceptification prescription at the concept level. Moreover this makes it possible to integrate CoBro with other environments so it relates to terminals for which the values are provided by these environments. For example a terminal representing a Prolog query whose value is computed by invoking the Prolog engine. Invoking the Prolog engine with the terminal value is realised by implementing a value interpreter for that particular kind of terminal.

## 7.2 The Origins of CoBro

In this section we discuss how different elements of the concept-centric environment were conceived and put to the test in projects with industry. We present

the lessons learned and relate this to the current version of our concept-centric environment.

### 7.2.1 SoFa

SoFa is the acronym used for the project titled 'Component development and product assembly in a software factory: organisation, process and tools'. It was an IWT funded research project in collaboration with MediaGeniX and EDS Belgium.

As already indicated by the acronym, the main topic of investigation were software factories (a.k.a. software product-lines). A software factory is a system that is intended to automate part of the software development lifecycle. Broadly speaking it divides the application to be built into commonalities and variabilities. The commonalities are the elements of the software that can be reused for several customers. The variabilities indicate the variation parts that are needed to customise the software towards the needs of the specific customer. The software products that share this set of commonalities and predicted variabilities are referred to as product families (e.g. financial applications). A software factory thus refers to the fact that a developer should only specify the variabilities of the product to be built, whereas the factory takes care of assembling the fully customised application according to these variabilities. An example family-based software development process that can be used to set up a software product-line is FAST [WL99].

The work package relevant in the context of this dissertation looked into the difficulties encountered by software companies to manage their domain-specific business expertise (e.g. about the domain of finance or broadcast management). The ultimate goal was to find a way to represent the variabilities in terms of business concepts, and to have these automatically translated into the required adaptations of the implementation. A less ambitious goal was set forth however where a first step was to support a business modeler in capturing the domain expertise of the software company. The next step was to use this explicit model of (internal) domain knowledge during the elicitation of (external) domain knowledge from a customer. This way the developer had a point of reference to which he could map the domain knowledge provided by the customer [WDVP00]. As a consequence, he could for instance indicate conflicting or synonymous terminology for the internal and external domain concepts.

For this purpose we built an ontology tool in Visual Basic in which the domain knowledge could be made explicit [Der02b, Der02a]. The internal representation was network-based and was used to relate the domain concepts to other domain concepts and terminals (e.g. images, unstructured text). Browsing the domain knowledge was done in a hyperlinked fashion. A screenshot of the tool is shown in figure 7.10. The window in the back is the main concept browser, the window in the front is one of the several concept editors. The left-hand pane of the concept

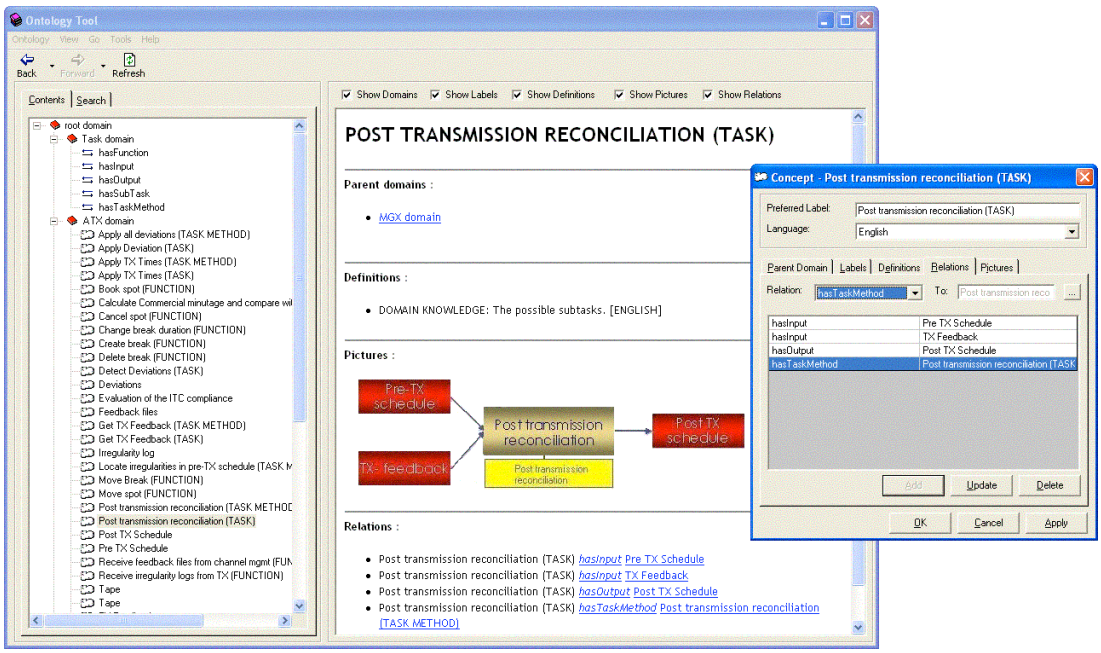



Figure 7.10: The SoFa Ontology Browser

browser supports navigating through the domain concepts in a tree-based fashion. As is shown the `Post Transmission Reconciliation` concept is selected, and its corresponding definition is rendered as HTML in the right-hand pane of the browser. The relationships, as well as the destinations in this HTML view on the concept are represented with hyperlinks to the corresponding concepts in the concept network. Within SoFa, this tool was proposed as a way to:

- avoid terminological confusion between the different stakeholders,
- guide interviews by business modelers and customers so as to couple the terminology of the customer to the terminology used within the software development company (and implicitly the implementation),
- use the existing domain knowledge as a reusable asset across project boundaries,
- use the domain knowledge as a starting point for identifying the variabilities and commonalities of the software product-line.

One of the results was a side-effect of making the domain knowledge explicit in the hyperlinked concept browser. We identified separate concepts (i.e. different words were used) that actually referred to the same thing. As it occurred, after looking up the corresponding implementation by hand, we found that they were implemented as separate but almost identical subsystems. As a result of this

observation both implementations were integrated, which eliminated the subsequent maintenance overhead. This was a first indication that it is important to be able to couple domain concepts to an implementation [DWL00]. Moreover, if a business modeler assembles the necessary delta's with respect to the existing domain knowledge, it is easier to identify which elements of the implementation probably require adaptation as well.

The tool that was built in the context of this project was not generic at all. For instance the meta model that defined the concept representation and the relations was hard-coded. As a consequence it was extremely difficult to extend the concept browser to accomodate new kinds of relations (e.g. one that allows the specification of the allowed destinations), previously unsupported terminals (e.g. a user interface specification), and corresponding browser visualisations (e.g. changing the rendered html-layout of the concept definition). Moreover, even though we identified the coupling towards the implementation as extremely important, this was not possible within the SoFa Browser.

The lessons learned during SoFa that are important for this dissertation can be summarised as follows:

- Domain knowledge is an important asset in software development, yet a lot of it remains implicit. There is a real need to be able to make it explicit.
- Terminology is often ambiguous since the same term can have different meanings in different contexts (i.e. because they are represented as terminals). Therefore it is necessary to have an underlying notion of concepts, which makes it possible to define the terminology by relating it in a concept network.
- Maintaining a link between domain knowledge and an implementation is a valuable tool for developers.
- A domain knowledge environment requires a highly flexible implementation so it can be easily customised and extended towards different uses.

### 7.2.2 eVRT MPEG

As indicated by the name, the context of the eVRT MPEG project was multimedia content. It was performed in collaboration with VRT, VUB, and IMEC. [2]

The central theme of the project was to investigate and develop the necessary technology to set up an enterprise-wide content management system in the context of a public radio and television broadcaster. The content stored in the system ranged from old archive material to material that was still in the process of being created [PNVB+03].

[2] This work was performed together with Peter Soetens.

The work-package of interest for this dissertation concentrated on meta-data management for a large multimedia archive. The multimedia archive consisted of over 500,000 digital archive documents with a growth of approximately 30,000 documents each year. The annotations of these documents were grounded in a thesaurus that contained 229,860 lead terms. The search engine of the archive made use of the lead terms and the relations between them to guide user searches. The use of the archive itself ranged from a work of reference for preparing newscasts to providing interesting material for game shows and documentaries. It is still used on a daily basis and is considered as a main asset for the broadcaster.

The constant pressure to accommodate new demanding search operations lead to uses of the thesaurus infrastructure that were never anticipated at the time of its conception. Even though the software had evolved, the underlying representation had not. It still was a thesaurus with a limited set of predefined relationships that could not be extended. Not surprisingly we found that this resulted in a number of 'inconsistencies' in the data as well as a number of creative abuses of the existing tools. Hence there was a major interest in investigating new ways of organizing and managing the archives' meta-data by revaluating the existing thesaurus.

For this purpose we built an experimental ontology-aware environment in which we could explore different ways to navigate through the content of the archive. The content for the ontology was given by converting the thesaurus into a prototype based concept network. Given the high number of concepts that were defined in the concept network, we encountered a number of scalability issues with respect to user interface components and the persistency layer. Our implementation was used to explore different ways to address these issues. The CoBro persistency layer is partially based on the work done at that time.

Also, during the analysis of the existing thesaurus we found that there were several conceptual lapses and inconsistencies [DS03]. These were mainly the result of its inadequacy as a medium to support the daily operation of the programme makers. Broadly speaking, the problems we identified were related to:

- Lead term ambiguity
  To resolve ambiguity, terms were suffixed with extra information. Yet these suffixes themselves were often ambiguous.

- Taxonomy
  As a result of the small number of available relations in the thesaurus, the existing relations were often overloaded to serve different purposes.

Similarly to our experience during the SoFa project this clearly indicated the need for an underlying notion of concepts. Moreover it became clear that the available relations should not be fixed beforehand. Again this results in the observation that a concept environment which is used to maintain such an asset should be as extensible as possible.

### 7.2.3 Advanced Media

The Advanced Media Project took place in collaboration with VRT, Vrije Universiteit Brussel, IMEC, and Universiteit Gent. The project covered a wide range of research objectives for which the main themes were:

- Software automation and knowledge engineering (SaKe)
- Semantic Web Services for Media Adaptation and Adaptation of Scalable Multimedia Documents (Castalot)
- Compression through Audiovisual Analysis (Cavia)

The theme that is important in the context of this dissertation was software automation and knowledge engineering. In the following we will restrict the discussion to research performed in this workpackage. The context of the workpackage was interactive media software development. In this project we proposed to install an iMedia Software Generation System, which combined elements of the research performed by Thomas Cleenewerck (Linglet Transformation System [Cle03]), Johan Brichau (Generative Logic Meta Programming [Bri05]), and myself (CoBro) [CDBD04, DCBD05]. It was during this project that the current version of CoBro was implemented. Hence the main characteristics of the concept-enviroment were made concrete during this project.

#### Interactive Media

Media broadcasters have observed that media consumers are no longer satisfied with the traditional situation in which the consumer had to 'sit back, relax and consume the media'. Hence they are extending their media product range with a more interactive form, i.e. interactive media. An iMedia product is created by integrating the traditional content component (audio, video, text, ...) with a behavioural component (software, website, ...) that is responsible for providing the interactive experience. Note that interactivity in this context should be interpreted more widely than the mere addition of a feedback channel. Examples of such behavioural components are online games that follow a TV show's story-line, interactive quizzes and online virtual communities for kids. All of these are highly 'integrated' with the content component (i.e. the corresponding TV show). Developing this kind of behavioural component not only requires an advanced set of software development tools, but also requires a different view on software development in general. This is easily motivated when looking at the specific characteristics of iMedia software development.

#### Interactive Media Software Development

First of all, iMedia software development takes place in an environment in which *extremely strict deadlines* constrain the development process. The most strict

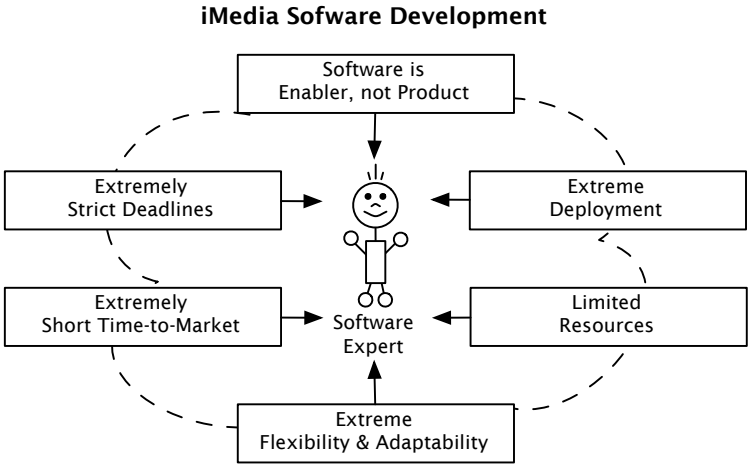



Figure 7.11: Main characteristics of iMedia software development.

deadline that manifests itself is the time at which a certain media production is put on air. Since media broadcast companies want to provide a total and integrated media experience, it is necessary that the content part as well as the interactive part are presented as a whole at that time. As broadcasting occurs in realtime, missing the broadcast deadline consequently renders the software completely useless. This is also related to the fact that broadcasting companies find themselves in a highly competitive market. Therefore it is of utmost importance to maintain high standards in order to obtain high viewer ratings and customer lock-in. In short this means that it is not acceptable to interrupt broadcasts, alter programme schedules or provide the software at a later time (e.g. at a later episode of a TV show) simply because it isn't ready yet.

Another characteristic of iMedia, which is very much related to the previous, is the *extremely short time-to-market situation* one is confronted with in this business context. Especially in the case of developing the (supporting) software component this aspect has a dual nature. First, software for media plays an *enabler-role* instead of being the product. Consequently this limits the number of justifiable person-months for development. It is clear that, unlike a game development company, a broadcaster cannot justify spending several man-years for developing an iMedia game. Second, the general media situation is one in which many events and decisions occur in real-time and on *extremely short notice*. This means that the necessary software components must allow fast assembly, configuration, and adaptation. In part this can be attributed to the creative aspect of media production in general. Last minute changes are no exception, since a lot of product features are crystallised as development moves onwards.

This implies that more evolution-power should go to the media producer, thus whenever possible bypassing the involvement of a software expert.

The last characteristic we will discuss is the current need for multi-platform and multi-channel publication of media. The former means that the iMedia have to be deployed on a plethora of devices (PDA, mobile phones, desktop computers). Each of these devices has its own set of specific capabilities which are clearly not overcome by using a cross-platform virtual machine. Examples of these are the available display size and memory for instance. The multi-channel aspect of media has to do with bringing the same content (and in the case of iMedia also the same behaviour) to different target audiences and media (e.g. TV, website). It is clear that this rather extreme deployment requirement is difficult to fulfil within the strict deadline / short time-to-market context.

In Figure 7.11 we summarise the main characteristics of the context of iMedia software development. This kind of business environment is an E-type system in the extreme. CoBro was proposed in part to meet the extremely strict deadlines, short time-to-market, flexibility and adaptability.

## 7.3 Summary

In this chapter we presented a number of applications of CoBro in practice. We illustrated this by refactoring a media library system to make use of an explicit concept level. This results in a malleable implementation that makes it more flexible towards future evolution efforts. Even though this results in a more generic implementation, all the information is still available to the developers at the explicit concept level. The resulting implementation depends heavily on the concept level to provide the functionality of the application. This illustrates the interplay between the code level and the concept level. Moreover it also shows how relationships available at the concept level can be used to establish a link between code level entities.

In addition to the application of CoBro to the media library software we also illustrated the extensibility of the CoBro environment. This is achieved by applying CoBro to itself. As a consequence it becomes possible to alter CoBro by applying changes at the concept level. Consequently a more malleable version of CoBro was obtained by refactoring it to make use of CoBro.

We also discussed the origins of CoBro, which lie in projects performed in collaboration with industry. We briefly introduced the SoFa, eVRT MPEG, and Advanced Media projects. The experiences obtained during each of these projects contributed to the formulation of this dissertation.

# Chapter 8

# Conclusion

In this dissertation we proposed the symbiotic integration of a programming environment with an explicit concept level that is coupled to the code level. We have demonstrated that this concept-centric environment amplifies the malleability of software through active concept level participation. Moreover we have shown that it supports program comprehension in the context of agile evolution.

In this chapter we present the conclusions of this dissertation. We begin by summarising the work presented, after which we discuss the main contributions. We end with a presentation of future research directions.

## 8.1 Summary

Current software development practices are put under a continuous strain to keep a software implementation in sync with the needs of the business environment it supports. Consequently *software evolution* is no longer an occasional activity, but a dire necessity that occurs on a daily basis. This implies a high degree of agility to accommodate the changes conceived in the business domain of the software.

A major amount of time during the evolution of software is dedicated to program comprehension. Developers first need to understand an implementation before they can alter it. For this they rely heavily upon their own domain expertise, but in practice a lot of time is 'wasted' on rediscovering the domain knowledge of the original developers. The main reason why this domain knowledge needs to be rediscovered is because the majority of knowledge about the software system and the associated business domain is not captured in an explicit form. Most of it remains *implicit* in the brains of the original developers or is lost over time. Also, even when the domain knowledge is available in an explicit form, it is often *detached* from the implementation. Nevertheless, developers need a link from the domain knowledge to the implementation. Establishing this coupling is far from trivial. Moreover it is unfortunate if you have to do it retroactively since there probably existed a clear link between both at some point in time. In addi-

tion, developers apply implementation techniques that make an implementation more *malleable* to anticipate future evolution efforts. A downside of such 'generic' implementations is that they result in less explicit code. This makes code comprehension an even more time consuming process, since a lot of assumptions become implicitly available to the developer. Implicit means that he or she needs to discover the assumptions that have become hidden in the data used to configure such systems at run time. Hence there is a tradeoff between malleability and simplicity. Besides this, creating a malleable version of an implementation introduces an overhead to the developer. Each time a developer adapts code to achieve malleability, he or she needs to implement a corresponding meta-infrastructure that governs the operational level of the system. This meta-infrastructure refers to the part of the implementation that governs the configuration of the instances that live in the operational level.

To overcome the problem of implicit domain knowledge we proposed a *concept-centric environment*. In this environment, domain knowledge is made *explicit* and is *coupled* to the corresponding code entities. As a consequence, developers can make explicit what they know about the software. In summary this is done in a frame-based representation where concepts are defined by relating them to other concepts or terminals in a concept network. The concept network itself is stored in a ConceptBase which is used as an ontology for both the developers and the implementation. Besides making explicit what they know, developers can also consult the domain knowledge that was made explicit by other developers. Combined with an explicit link between the domain knowledge (represented at a concept level) and the implementation (represented at the code level), this reduces the effort needed for code comprehension. This was done with an automatic conceptification process that represents code level entities at the concept level. Subsequently these code level concepts can be connected to other entities with the same mechanisms used for domain concepts.

However this is only a partial solution to avoid that domain knowledge becomes implicit in the context of agile evolution. There is no short term benefit for a developer who spends time on writing down domain knowledge relevant to the implementation. This is because it is used as a *passive* source of documentation. As a consequence the domain knowledge is still neglected in favor of writing code since the latter has a clear benefit of delivering the system in time. Moreover, spending time at the concept level implies that there is less time available to spend at the code level. We characterised this situation as an *antibiosis* between the concept and the code level. Therefore we added an important requirement to the concept-centric environment, namely that the concept level must participate *actively* in providing the functionality of the system. This activation of the concept level implies that concepts are invoked at runtime to provide the functionality of the system. Thus spending time at the concept level no longer has a negative impact on delivering the software in time.

To ensure that developers no longer perceive the concept level as an *overhead*

or old-style passive documentation, we put forth the requirement of *transparency* to the programmer. This boils down to ensuring that the manipulation of the concept level is done in the same medium as the code level. In essence, the concept level is manipulated through the same syntax, semantics and environment as the code level. This is referred to as the symbiotic integration of the programming environment with the concept environment.

To validate and verify the practical feasibility of a concept-centric environment we implemented CoBro. CoBro is the proof-of-concept environment which we implemented in Smalltalk. It provides all the necessary elements to cover the requirements put forth in this dissertation. We discussed the implementation of CoBro and investigated the requirements for the code level. The practical application of CoBro was illustrated with examples and on a media library case study. Moreover we have also shown how CoBro was applied to itself to show its power for amplifying the malleability of software.

## 8.2 Conclusions

In this dissertation we proposed and implemented a concept-centric environment in which domain knowledge can be made explicit and can be coupled to the corresponding implementation. Moreover this environment was conceived in a way as so it becomes possible to invoke the domain knowledge to provide the functionality of a software system. Through a symbiotic integration with a programming environment and the concept environment we achieved a level of transparency necessary to motivate developers to spend time at the concept level. As a consequence the boundary between what is considered as domain knowledge and what is considered as programming becomes oblivious to the developer. This was expressed by the thesis statement of this dissertation:

> *A **symbiotic integration** of a programming environment with*
> *an **explicit concept level** that is **coupled** to the code level*
> *supports **domain knowledge documentation***
> *in the context of **agile evolution***
> *and amplifies the **malleability** of software through*
> ***active** concept level **participation**.*

In the remainder of this section we briefly summarise the main contributions of this dissertation.

## Identification of the Need for an Explicit Concept Level

We have identified and analysed the need for an explicit concept level in the context of software evolution. In essence we synthesised the result of this analysis into four key points. First of all the majority of domain knowledge about an implementation and its associated business domain is only available in an *implicit* form to the developers. As a result, a lot of time is spent on rediscovering what was known when the system was initially conceived. Secondly, even when this is not the case, the domain knowledge is *detached* from the implementation. As a consequence developers need to reconnect the domain knowledge to the implementation in a retroactive way. Thirdly, domain knowledge is seen as a *passive* asset that does not contribute to the next release of the implementation. The potential benefit is realised on the long term, and is not a certainty. Therefore it is neglected in favor of writing code that does have a clear short term benefit. Lastly developers perceive writing down what they know as an *overhead*. In part this can be attributed to the passive role that is played by the domain knowledge. Another reason is that documentation is done outside the programming environment, which is considered a disturbance of their programming experience.

## Concept-centric Environment

We constructed a conceptual framework for a concept-centric environment in which the key points identified in the previous contribution were addressed. To reduce the possible loss of domain knowledge we set up an *explicit* concept level that is represented in a frame-based ConceptBase. The effort spent by developers on retroactively matching domain knowledge to an implementation is supported by a mechanism that enables the *coupling* of entities from the code level to entities from the concept level. This is supported with an automatic conceptification process that takes care of representing code entities at the concept level. This way they can be coupled to other entities at the concept level, which can be domain concepts or concepts that are the result of conceptification themselves. Moreover the concept level can be *actively* used from within the code level. As a consequence the concept level is involved in realising the behavior of the application. This results in a visible short term benefit for developers, which motivates them to dedicate time to the concept level. Consequently we made it possible to embed concept level statements within normal code. Moreover an up/down mechanism was provided by which (de)conceptification can be triggered in a programmatic way. This lets a developer take advantage of the relationships between code entities that are only available at the concept level. The potential overhead of the concept-centric environment is eliminated by implementing a *symbiotic integration* between the concept level and the code level. As a consequence a degree of *transparency* is achieved in which it becomes oblivious to the programmer whether he is working with concepts or with objects.

### Concept-driven Malleability

The mechanism that enables the active use of concepts in the execution of a program is the main force behind concept-driven malleability. To obtain a malleable implementation, a developer changes the implementation so it makes use of the concept level to provide its functionality. As a consequence the software can be adapted by applying changes at the concept level. Even though this results in less explicit code, the corresponding concept network that contains the implicit assumptions of the code level, remains available to the developers at all times.

The active use of the concept level is facilitated through the symbiotic integration of the programming environment and concept environment, and the availability of an up/down mechanism. This enables among others to use horizontal relationships between code level entities that are only available at the concept level.

Evolving the concept level to strengthen what can be done at the code level results in an iterative incremental interplay which was characterised as co-evolution between both levels.

Moreover, the concept-centric environment provides the basic infrastructure required to set up a malleable implementation. Consequently a developer can focus all his attention to the operational level and the meta-level of the implementation, instead of setting up his own dedicated meta-level infrastructure.

### CoBro

We implemented a highly extensible proof-of-concept environment named CoBro which was used as a vehicle to validate the feasibility of a concept-centric environment. We discussed the prerequisites to implement a concept-centric environment as well as the technical implementation of this environment. The CoBro environment illustrates that it is indeed feasible to have a strong symbiotic integration between the concept environment and the programming environment. The integration manifests itself throughout the different tools (e.g. CoBro-Nav) that are provided to a developer. The symbiosis is the result of using the same syntax and interaction mechanisms of the programming language which resulted in CoBro-CML. We validated the practical applicability of a concept-centric environment by evolving a media library case study with CoBro. Moreover, the power of malleability was stressed by applying CoBro to itself.

## 8.3 Future Work

In this section we present a number of directions for future research as well as a number of possible improvements.

## ConceptBase Versioning and Management

Currently we have no mechanisms in place for supporting different versions of concepts in the ConceptBase. This is needed to be able to support a large network of interrelated concepts where certain relations only hold for a particular version of the concept network. Evolving the concept network often results in the addition or deletion of relations to a concept definition. This introduces a new version of a concept, hence it should be verified if related concepts still 'commit' to the new definition. Therefore each concept should be versioned, as well as the slots that hold the relations. In part we believe we can base ourselves on temporal ontologies. In addition to this we want to verify if we can use concepts from temporal ontologies to annotate software versions as well.

We would like to explore different ways to manage the concept network in the ConceptBase. In part this is solved by providing tool support, but we would like to explore support at the concept level as well. For example to organise the slots in the definition frame of a concept it would be useful to have a grouping mechanism available (e.g. analogously to traits in prototype-based programming languages). Moreover support is needed to verify the validity of the core ontology. For example verifying whether the core ontology meets the prerequisites for a particular application. In part we have already indicated that this can be done by setting up a validation mechanism at the concept level or by writing S-Unit test cases. Yet this should be further investigated to provide a more rigourous approach to validation.

## Granularity of Conceptification

In the current implementation of CoBro we have restricted the granularity of conceptification to the class level. We already discussed in this dissertation how this granularity can be extended to include for example methods at the concept level. We would like to investigate the potential use of concept level relationships between such finer grained code level entities. This could be used for example to express relations between two particular methods or instance variables. The conceptification process could make use of existing software meta-models to decide how to represent code level entities at the concept level.

Representing finer grained concepts and versions of concepts at the concept level introduces a large number of concepts in the concept network. This requires a scalability study to see whether this is feasible in practice. Each time concepts are accessed there is a performance penalty by accessing the ConceptBase. Implementations that depend heavily on concepts to provide their behavior generate a lot of ConceptBase traffic at runtime. Consequently special care is needed with respect to the performance of ConceptBase access. We need to explore the different possibilities further to improve the performance of the concept level (e.g. caching, indexing). Moreover, depending on what to represent in the concept

level, it might be better to move away from the current relational database to an object-oriented database.

## Concept-based Pointcuts

We would like to explore the use of CoBro in the context of Aspect-Oriented Software Development. More specifically we want to investigate concept-based pointcuts. A pointcut defines where and when an aspect must be invoked in the execution of the code. Since traditional pointcuts rely upon the structure of the code, only the structural information can be used in the specification. Since concepts are actively invoked by code that is written in a concept-oriented fashion, we can use the concept level 'annotations' to specify pointcuts that take domain information into account. This is related to the work on a model-driven pointcut language by Kellens et al. [KMBG06].

## User Interface Composition

We would like to investigate how a concept-centric environment can be applied to obtain reusable user interface components. This is also an experiment that is based on the granularity of the conceptification process. Initial experiments were already performed together with Sofie Goderis [GD04, GD05]. In these experiments we conceptified a user interface specification up to the level of the individual user interface components (e.g. text boxes, buttons). This made it possible to relate these components to each other and to annotate them with domain concepts (e.g. part-of edit-group). Example relations that were used indicated the relative position of the components with respect to other components (e.g. left-of, above), and the grouping of components at the concept-level (e.g. buttons that are enabled or disabled according to the current edit mode of a window). The resulting concept network was used by a system called Deuce to automatically lay out the user interface components according to the information provided at the concept level. Also the grouping of components was used by this system to be able to enable or disable user interface components based on this domain knowledge.

We would like to investigate further how the composition of user interfaces can be facilitated at the concept level. For this we envision a situation in which domain concepts are annotated with small user interfaces. For example an `Address` concept contains a `Street`, `Town`, and `Country` concept. Each of these concepts have associated user interfaces (e.g. `Street` is represented by a label and a textbox) which are used to compose the user interface of the `Address` concept. This requires among others a number of domain knowledge annotations with respect to lay out and component dependencies.

## Integration with Existing Research Artifacts

One of the strengths of the current implementation of CoBro is its extensibility. On a short term basis we want to explore the added value of connecting CoBro to existing research environments. This can be done by creating dedicated terminals and value interpreters at the concept level. Examples of such environments are SOUL [Wuy01] and intensiVE [MK06]. These environments are more oriented towards code-based knowledge (e.g. design or architectural knowledge), whereas CoBro is oriented towards domain-based knowledge. This means that they have investigated different mechanisms to be able to query the code base. This can be used for example to couple domain knowledge to code based on such a query.

Another endeavour is the connection between CoBro and Domain-Specific Languages research (more specifically the work performed by Thomas Cleenewerck [Cle03]). We want to investigate how an explicit concept level can be used to ground and document the domain-specific language constructs in the context of evolution.

## Concept-Oriented Programming

Refactoring an existing implementation into a concept-centric implementation requires further research. First of all a decision needs to be made about *what* to represent at the concept level and what to represent at the code level. This should not be done on an ad hoc basis, but requires a number of guidelines. Secondly, dedicated concept-centric refactoring steps are required. The definition of refactoring states that it is a technique for restructuring an existing body of code, where the internal structure of the code is altered, but not the external behavior [FBB+99]. So we would like to identify a number of small behavior preserving transformation steps (concept-centric refactorings) that transform code into concept-centric code.

This is a first step towards a concept-oriented programming language in which the interplay between the concept level and the code level is not seen as something special, but as the normal way of doing things.

# Bibliography


[ABDT04] A. Abran, P. Bourque, R. Dupuis, and L.L. Tripp. Guide to the software engineering body of knowledge (ironman version). Technical report, IEEE Computer Society, 2004. 2, 14, 16, 26

[AJ02] S. Ambler and R. Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley Computer Publishing, 2002. 24

[Bec97] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997. 41

[Bec02] S. Bechhofer. Ontoweb report : Ontology language standardisation efforts. Technical report, IST Project, IST-2000-29243, 2002. 60

[BGH02] S. Bechhofer, C. Goble, and I. Horrocks. Ontoweb report : Requirements of ontology languages. Technical report, IST Project, IST-2000-29243, 2002. 60

[Boo87] G. Booch. *Software Components with Ada*. Benjamin / Cummings, 1987. 46

[BP01] J. Bézivin and N. Ploquin. Tooling the mda framework: a new software maintenance and evolution scheme proposal. *Journal of Object-Oriented Programming JOOP*, 2001. 54

[BR00] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000. ACM Press. 16

[Bra95] J. M. Brant. *Hotdraw*. Master thesis, University of Illinois at Urbana Champaign, 1995. 110

[Bri93] D. Brill. Loom reference manual v2.0. Technical report, Information Sciences Institute, University of Southern California, 1993. 58

[Bri05] J. Brichau. *Integrative Composition of Program Generators*. PhD thesis, Vrije Universiteit Brussel, 2005. 186

[CC05] G. Canfora and A. Cimitile. Software maintenance. *IT Metrics and Productivity Journal*, November 2005. Online article - e-Zine. 16, 26

[CDBD04] T. Cleenewerck, D. Deridder, J. Brichau, and T. D'Hondt. On the evolution of imedia implementations. In *Proceedings of the European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology*, 2004. 186

[CE00] C. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 36

[CGP00] O. Corcho and A. Gómez-Pérez. Evaluating knowledge representation and reasoning capabilities of ontology specification languages. In *ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods. Berlin, Germany*, 2000. 59, 60

[Che76] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. 47

[CHK+01] N. Chapin, J.E. Hale, K.Md. Khan, J.F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001. 15, 16

[CJB99] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. What are ontologies and why do we need them? *IEEE Intelligent Systems and their Applications*, 14(1):20–26, 1999. 49, 53, 56

[Cle03] T. Cleenewerck. Component-based dsl development. In *In Proceedings of GPCE. Lecture Notes in Computer Science 2830*, pages 245–264. Springer Verlag, 2003. 186, 196

[Cor05] R. Corazzon. Ontology: A resource guide for philosophers, 2005. 49

[Cyc05] Cyc. http://www.cyc.com/, 2005. 58

[DCBD05] D. Deridder, T. Cleenewerck, J. Brichau, and T. D'Hondt. Software automation meets interactive media development, 2005. 186

[DDVMW00] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the International Symposium on Software Architectures and Component Technology*, 2000. 25

[Der02a] D. Deridder. A concept-oriented approach to support software maintenance and reuse activities. In T. Welzer et al., editor, *Knowledge-based Software Engineering, Frontiers in Artificial Intelligence and Applications, Vol. 80*. IOS Press, 2002. 182

[Der02b] D. Deridder. Facilitating software maintenance and reuse activities with a concept-oriented approach. In *Workshop on Knowledge-Based Object-Oriented Software Engineering (KBOOSE), 16th European Conference on Object-Oriented Programming (ECOOP)*, 2002. 182

[DH98] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 1998. 27

[DP00] T. H. Davenport and L. Prusak. *Working Knowledge: How organizations Manage What They Know*. Harvard Business School Press, 2000. 39, 40, 43

[DS03] D. Deridder and P. Soetens. Revaluation of a large-scale thesaurus for multi-media indexing: An experience report. In *Industry Program Workshop, OTM'2003 Conference, Lecture Notes in Computer Science*, volume 2889/2003, pages 35–45. Springer-Verlag, Heidelberg, 2003. 50, 185

[DWL00] D. Deridder, B. Wouters, and W. Lybaert. The use of an ontology to support a coupling between software models and implementation. In *European Conference on Object-Oriented Programming (ECOOP 2000), International Workshop on Model Engineering*, 2000. 184

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999. 67, 196

[FK00] A. Finkelstein and J. Kramer. Software engineering: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000. ACM Press. 66

[Fow98] M. Fowler. Keeping software soft. *Distributed Computing*, 1998. 20, 69

[Fow01] M. Fowler. To be explicit. *IEEE Software*, November, December 2001. 20, 69

[Fow05] M. Fowler. Code as documentation. RSS feed, march 2005. 24

[GD04] S. Goderis and D. Deridder. Declarative dsl approach to ui specification - making ui's programming language independent. In *Workshop on Evolution and Reuse of Language Specifications for DSLs (ERLS), Ecoop*, 2004. 113, 195

[GD05] S. Goderis and D. and Deridder. Declarative user interfaces : specifying ui variabilities. In *2nd Workshop on Managing Variabilities Consistently in Design and Code, OOPSLA*, 2005. 113, 195

[GG95] N. Guarino and P. Giaretta. Ontologies and knowledge bases - towards a terminological clarification. In N. J. I. Mars, editor, *Towards Very Large Knowledge Bases, IOS Press, Amsterdam*, 1995. 48

[GGM+02] A. Gangemi, N. Guarino, C. Masolo, A. Oltrami, and L. Schneider. Sweetening ontologies with dolce. In V.R. Benjamins A. Gómez-Pérez, editor, *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002*, pages 166–181. Springer Verlag, October 2002. 53

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995. 41

[Gru93] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers. 48, 52

[Gua98] N. Guarino. Formal ontology and information systems. In *Proceedings of FOIS 1998, Trento, Italy*. IOS Press, Amsterdam, June 1998. 48, 53, 62

[Hal01] T. A. Halpin. *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers, 2001. 47

[Har02] M. Harsu. A survey on domain engineering. Technical report, Institute of Software Systems, Tampere University of Technology, 2002. 45

[HI00] J. A. Highsmith III. *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, 2000. 22

[IEE98] IEEE. *IEEE Std 1219-1998: IEEE Standard for Software Maintenance*. The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1998. 14

[ISO99] ISO/IEC. *ISO/IEC 14764-1999 : Software Engineering - Software Maintenance*, 1999. 14, 15

[Jac04] E. K. Jacob. Classification and categorization: A difference that makes a difference. *Library Trends*, 52(3):515–540, 2004. 58

[Kal02] Y. Kalfoglou. *The Handbook of Software Engineering and Knowledge Engineering : Fundamentals*, volume 1, chapter Exploring Ontologies, pages 863–887. World Scientific, 2002. 52

[Kav03] M. Kavouras. A unified ontological framework for semantic integration. In *International Workshop on Next Generation Geospatial Information*, October 2003. 53

[KMBG06] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP, to appear*, 2006. 195

[Kri98] G. Kristen. *Kennis is Macht (2nd Edition)*. Academic Service, 1998. 27

[KSP04] J. Koskinen, A. Salminen, and J. Paakki. Hypertext support for the information needs of software maintainers. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:187–215, 2004. 3, 29, 68

[Leh96] M.M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996. 16, 17

[Lin03] D. S. Linthicum. Leveraging ontologies and application integration. *eAI Journal*, pages 37–39, May 2003. 53

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980. 16

[LS98] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence - Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998. 59

[MBRZ02] T. Mens, J. Buckley, A. Rashid, and M. Zenger. Towards a taxonomy of software evolution. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel, Programming Technology Lab, Pleinlaan 2, 1050 Brussel, Belgium, 2002. 15

[Men00] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 2000. 25

[Min75] M. Minsky. *The Psychology of Computer Vision*, chapter A Framework for Representing Knowledge. McGraw-Hill, 1975. 77

[MK06] K. Mens and A. Kellens. Intensive, a toolsuite for documenting and checking structural source-code regularities. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2006. 196

[MS04] D. Mayhew and D. Siebert. Ontology: The discipline and the tool. In *First International Workshop on Philosophy and Informatics, University of Applied Sciences, Cologne, Germany*, March 2004. 49, 54

[MW05] Merriam-Webster. Merriam-webster online dictionary, 2005. 34, 50

[Myl80] J. Mylopoulos. An overview of knowledge representation. *ACM*, 1980. 58

[Nei80] J. M. Neighbors. *Software Construction Using Components*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1980. 43

[NM01] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Technical Report SMI-2001-0880, Knowledge Systems Laboratory, Stanford University, 2001. 55, 60, 61

[NP01] I. Niles and A. Pease. Towards a standard upper ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS 2001)*, October 2001. 53, 58

[NT95] I. Nonaka and H. Takeuchi. *The Knowledge-Creating Company - How Japanese Companies Create the Dynamics of Innovation.* Oxford University Press, 1995. 39, 40, 42, 43, 59

[OG98] C. O'Dell and C. J. Grayson. *If Only We Knew What We Know: The Transfer of Internal Knowledge and Best Practice.* Free Press, 1998. 40

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. 67

[Ope05] OpenCyc. http://www.opencyc.org/, 2005. 58

[Par94] D. L. Parnas. Software aging. In *roceedings of the 16th International Conference on Software Engineering*, pages 279–287, 1994. 18, 27

[PD90] R. Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSoft Software Engineering Notes*, 15(2):47–54, 1990. 27

[PDA91] R. Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, 1991. 35

[Pfl98] S. L. Pfleeger. The nature of system change. *IEEE Software*, pages 87–90, May/June 1998. 17

[Pig96] T. M. Pigoski. *Practical Software Maintenance - Best Practices for Managing Your Software Investment.* Wiley Computer Publishing, 1996. 15, 16

[Pla05] TELEBIB2 Platform. e-business standards to facilitate policy administration, claims handling and accounting for the belgian insurance sector, 2005. 39

[PNVB+03] E. Parton, L. Nachtergaele, K. Van Bruwaene, P. Schelkens, and D. Deridder. Mpeg-4, bouwsteen voor multimedia. *Het Ingenieursblad*, 10(11), 2003. 184

[RB00] V. Rajlich and K. Bennett. A staged model for the software life cycle. *IEEE Computer*, pages 66–71, July 2000. 18, 29

[Ree05] J. W. Reeves. Code as design: Three essays. Code as Design: Three Essays, 2005. 24, 27

[Rei97] D. J. Reifer. *Practical Software Reuse.* Wiley Computer Publishing, 1997. 45

[RL02] I. Rus and M. Lindvall. Knowledge management in software engineering. *IEEE Software, 0740-7459/02*, pages 26–38, May/June 2002. 41

[RLS01] I. Rus, M. Lindvall, and S. S. SinhaI. Knowledge management in software engineering: A dacs state-of-the-art report. Technical Report SP0700-98-4000, Fraunhofer Center for Experimental Software Engineering Maryland, 2001. 26, 41

[RTJ05] D. Riehle, M. Tilman, and R. Johnson. The dynamic object model pattern. (to appear), (to appear) 2005. 20, 62

[SAA+00] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde, and B. Wielinga. *Knowledge Engineering and Management - The CommonKADS Methodology - A software engineering approach for knowledge intensive systems.* MIT Press, 2000. 42, 43, 67

[SEI05] Carnegie-Mellon University SEI, Software Engineering Institute. Software technology roadmap: Domain engineering and domain analysis, 2005. 46

[Sim96] M. A. Simos. Organization domain modeling: A tailorable, extensible framework for domain engineering. In *Proceedings of the 4th International Conference on Software Reuse, ICSR'96.* IEEE, 1996. 36

[SIM99] F. M. Shipman III and C. C. Marshall. Formality considered harmful: Experiences, emerging themes, and directions. *Computer-Supported Cooperative Work*, 8(4):333–352, 1999. 76

[SM89] S. Shlaer and S. J. Mellor. An object-oriented approach to domain analysis. *ACM SIGSOFT, Software Engineering Notes*, 14(5):66–77, July 1989. 45

[Sow99] J. F. Sowa. *On Logical, Philosophical, and Computational Foundations of Knowledge Representations.* PhD thesis, Vrije Universiteit Brussel, June 1999. 49

[SUM05] SUMO. http://suo.ieee.org/suo/sumo/, 2005. 53

[Swa76] E.B. Swanson. The dimension of maintenance. In *Proceedings of the Second International Conference on Software Engineering*, pages 492–497, October 1976. 14

[Swa97] N. Swartz. Definitions, dictionaries, and meanings. Technical report, Department of Philosophy, Simon Fraser University, 1997. 57

[Tai97] A. Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. *Journal of Object-Oriented Programming*, 10(7):44–50, Nov/Dec 1997. 57

[UG96] M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. Technical Report AIAI-TR-191, University of Edinburgh, 1996. 51, 52

[Usc98] M. Uschold. Knowledge level modelling: Concepts and terminology. Technical Report AIAI-TR-196, Artificial Intelligence Applications Institute, University of Edinburgh, 1998. 49

[vMV95a] A. von Mayrhauser and A.M. Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10(5):171–182, 1995. 29

[vMV95b] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, August 1995. 30

[vMV97] A. von Mayrhauser and A.M. Vans. Program understanding needs during corrective maintenance of large scale software. In *Proceedings 21st International Computer Software and Applications Conference (COMPSAC97)*, pages 630–637, Los Alamitos CA, 1997. IEEE Computer Society Press. 29

[vMV98] A. von Mayrhauser and A.M. Vans. Program understanding during software adaptation tasks. In *Proceedings International Conference on Software Maintenance (ICSM 1998)*, pages 316–325, Los Alamitos CA, 1998. IEEE Computer Society Press. 29

[vMVH97] A. von Mayrhauser, A.M. Vans, and A.E. Howe. Program understanding behaviour during enhancement of large-scale software. *Software Maintenance: Research and Practice*, 9(5):299–327, 1997. 29

[Voa98] J. Voas. Are cots products and component packaging killing software malleability. In *Proceedings of the 14th IEEE International Conference on Software Maintenance (ICSM98)*, page 156, 1998. 19

[Wag90] W. Wagner. Issues in knowledge acquisition. *ACM 089791-416-3/90/0010/0247*, pages 247–261, 1990. 36

[WDVP00] B. Wouters, D. Deridder, and E. Van Paesschen. The use of ontologies as a backbone for use case management. In *European Conference on Object-Oriented Programming(ECOOP 2000), Workshop : Objects and Classifications, a natural convergence*, 2000. 182

[Weg02] H. Wegener. Agility in model-driven software development? implications for organization, process, and architecture. In *OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2002. 23

[Wel95] C. A. Welty. *An Integrated Representation for Software Development and Discovery.* PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, Troy, USA, 1995. 60

[WF94] C. A. Welty and D. A. Ferrucci. What's in an instance? Technical report, RPI Computer Science, 1994. 61

[Wit04] L. Wittgenstein. Family resemblances. In B. Aarts, D. Denison, E. Keizer, and G. Popova, editors, *Fuzzy Grammar - a Reader*, pages 41–44. Oxford University Press, 2004. 57

[WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering - A Family-Based Software Development Process.* Addison-Wesley, 1999. 45, 182

[Wor05] Wordnet. http://wordnet.princeton.edu/, 2005. 58

[WPD92] S. Wartik and R. Prieto-Diaz. Criteria for comparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):403, 431 1992. 34, 42

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Lab, Brussels, Belgium, 2001. 25, 196

# Index

adaptive software development, 23
agile software development, 22, 23
application domain, 34, 38
application engineering, 45, 48
application ontology, *see* domain-specific ontology

business domain, 34, 38

class-based ontology, 56, 61
clean slate situation, *see* green-field situation
co-evolution of code and model, 25
concept categorisation, 57
concept classification, 57
concept granularity, 56
conceptualisation, 52
continued development, 16, 18, 19, 22
continuing change, 18
controlled vocabulary, 50

dictionary, 51
diffused domain, 36
distributed domain, 38
documentation, 24
domain, 34, 38
domain analysis, 39, 43, 48
domain boundary, 34
domain composition, 38
domain coverage, 38
domain engineering, 45, 48
domain knowledge, 26, 27, 37, 43, 44
domain lexicon, 47, 48
domain model, 39, 42, 43, 46, 48
domain ontology, 53
domain vocabulary, 39
domain-composition, 35
domain-coverage, 35
domain-specific ontology, 53, 55
dynamic object model, 62

encapsulated domain, 36, 38
explicit knowledge, 40, 44
extensional concept definition, 57

foundational ontology, 53, 55

glossary, 51
green-field situation, 35

horizontal domain, 36, 38
horizontal domain analysis, 46, 48
horizontal ontology, 53, 55

inference knowledge, 43
intangible knowledge, *see* tacit knowledge
intensional concept definition, 57

knowledge, 37, 44
knowledge acquisition, 41
knowledge conversion, 40
    combination, 41, 44

externalisation, 41, 44
internalisation, 41, 44
socialisation, 41, 44
knowledge spiral, 40

lexicon, 51

malleable software, 19, 27
development-time malleability, 20
run-time malleability, 20

ontological commitment, 51, 52
ontology, 48, 51, 52, 54
ontology alignment, 58
ontology application-layer, 54
ontology core-layer, 54
ontology development
bottom-up, 60
top-down, 60
ontology domain-layer, 54
ontology granularity, 52, 55
ontology representation language, 59
ontology-aware application, 62
ontology-driven application, 62

ploughed-field situation, 35
problem-level concept, 42
program comprehension, 26, 28
prototype-based ontology, 56, 61

software aging, 18
ignorant surgery, 18
lack of movement, 18
software closedown, 19
software development, 20
code-centric, 23, 27
guided approach, 23
model-centric, 23, 27
planning approach, 23
software evolution, 16, 18
E-type system, 17
laws, 17
P-type system, 17
S-type system, 16
software genericity, 20, 27
software maintenance, 14
adaptive maintenance, 15, 16
corrective maintenance, 15, 16
cost, 16
perfective maintenance, 15, 16
preventive maintenance, 15
taxonomies, 14
software malleability, 62
software phaseout, 19
software servicing, 19
software warranty, 14
solution-level concept, 42
staged software lifecycle, 18
synchronisation of code and model, 25

tacit knowledge, 40, 44
tangible knowledge, *see* explicit knowledge
task knowledge, 43
task ontology, 53
taxonomy, 50
thesaurus, 50, 51
top-level ontology, *see* upper ontology

upper ontology, 53, 55

vertical domain, 36, 38
vertical domain analysis, 46, 48
vertical ontology, 53, 55
vocabulary, 51