Universität Ulm, Fakultät für Informatik,
Abteilung Programmiermethodik und Compilerbau

An Agile Development Methodology for Knowledge-Based Systems Including a Java Framework for Knowledge Modeling and Appropriate Tool Support

Dissertation zur Erlangung des Doktorgrades Dr.rer.nat.
der Fakultät für Informatik der Universität Ulm



Holger Knublauch
aus Berlin-Lichterfelde

Amtierender Dekan:

Prof. Dr. Günther Palm

Gutachter:

Prof. Dr. Helmuth Partsch
Prof. Dr. Dr. Franz-Josef Radermacher

Datum der Promotionsprüfung: 16.10.2002

Abstract

The goal of this thesis is to help make the development of knowledge-based systems more efficient. For that purpose, it proposes a new, agile software and knowledge engineering methodology, called XP.K (eXtreme Programming of Knowledge-based systems). This methodology is based on the four values *simplicity*, *community*, *feedback*, and *courage*, and applies object-oriented Round-Trip Engineering to knowledge modeling.

The thesis is founded on the observation that for most knowledge-based systems, knowledge must necessarily be modeled evolutionary, in a close collaboration between domain experts and engineers. The author argues that existing "heavy-weight" development methodologies from object-oriented Software Engineering and Knowledge Engineering are often inefficient, because they make changes in knowledge models too expensive. Furthermore, they provide little support for the transitions between knowledge, knowledge models, and the remaining executable system. The starting point of XP.K is the hypothesis that "light-weight" – or *agile* – development processes (such as Extreme Programming) are suitable for knowledge modeling, because they are optimized for projects with frequently changing requirements and models and rely on a minimum of modeling artifacts with smooth transitions between them.

XP.K applies the main principles of Extreme Programming to knowledge modeling. The development process relies heavily on communication. Domain experts and knowledge engineers collaborate in the definition of metamodels (ontologies) and knowledge is acquired and tested in pairs. Ontologies are implemented in the object-oriented language which is also used for the remaining modules of the system. These ontologies transparently expose their structure and semantics both at run-time and build-time. At run-time, reflection is used to analyze the ontology, so that generic knowledge acquisition tools and inference engines can be (re-)used, reducing the cost of changing ontologies. At build-time, Round-Trip Engineering is used to extract and edit the structure of the ontology in UML, so that all information and documentation can remain centrally in the code. Prototypes are produced rapidly and frequently exposed to automated tests and constraint checks.

XP.K is supported by a comprehensive, ready-to-use framework for implementing knowledge-based systems in Java. This includes an extension to the Java component model JavaBeans for the representation of semantic constraints on object states and a generic framework for knowledge acquisition tools. Evidence for the efficiency of XP.K is provided by case studies from areas such as clinical decision-support and process modeling for multi-agent systems.

Prologue

> The wind will not stop. Gusts of sand swirl before me, stinging my face. But there is still too much to see and marvel at, the world very much alive in the bright light and wind, exultant with the fever of spring, the delight of morning. Strolling on, it seems to me that the strangeness and wonder of existence are emphasized here, in the desert, by the comparative sparsity of the flora and fauna: life not crowded upon life as in other places but scattered abroad in spareness and simplicity, with a generous gift of space for each herb and bush and tree, each steam of grass, as that the living organism stands out bold and brave and vivid against the lifeless sand and barren rock. The extreme clarity of the desert light is equaled by the extreme individuation of desert life forms. Love flowers best in openness and freedom.
>
> Edward Abbey, *Desert Solitaire* (1968)

Preface

This thesis is based on the results of two related research projects carried out at the Department of Business Processes and Telematics at the Research Institute for Applied Knowledge Processing (FAW) at the University of Ulm, Germany. The project *ANIS* (1998–2000, funded by the German governmental department of Education and Research (BMBF)) aimed at developing a prototypical knowledge-based patient monitor for decision support in anesthesia. The project *AGIL* (2000–2002, funded by the German Research Foundation (DFG)) focused on the development of a multi-agent system to optimize the clinical information flow. The author was principal researcher in both projects which were conducted in cooperation with clinical experts and researchers from the Institute for Psychology and Ergonomics at the Technical University of Berlin.

Acknowledgements

I am indebted to Dr. Thomas Rose, head of the Department of Business Processes and Telematics at the FAW, Ulm, and the project leader in AGIL and ANIS. Thomas has made valuable critical comments and other contributions to this research and my scientific work. Furthermore, he has generated an inspiring working environment in which I could unfold freely. An essential part of this environment was my long running roomate Gerhard Peter, who taught me many important lessons on scientific research and contributed to a friendly and motivating workplace. Special thanks to Martin Sedlmayr, who has started as a student research assistant in the ANIS project and later became a leading creative mind at the FAW. Several discussions with Martin have lead to important advances in ANIS and have laid the foundation of the KBeans approach. On this occasion, I would also like to express my gratitude to my (former) colleagues in the department, in particular to Martin Fünffinger, Dr. Christian Greiner and Dr. Christian Rupprecht.

I would like to acknowledge Holger Köth, M.D., who was my partner in the AGIL project in Berlin, and who has contributed to our knowledge modeling approach and the resulting clinical multi-agent system. Many thanks also to the eager participants of the Extreme Programming course at the University Ulm in fall, 2001, in which I had the opportunity to evaluate parts of my Extreme Programming approach with a real-world project.

Last but not least, I would like to express my gratitude to my supervisors Prof. Dr. Helmuth Partsch and Prof. Dr. Dr. Franz-Josef Radermacher for supporting this thesis and the Extreme Programming courses despite the rather unconventional topic. Prof.

Dr. Partsch has also laid the foundation of my scientific career by introducing me to the basics of compiler construction, requirements engineering and software technology, and by supervising my masters thesis in which I learned to appreciate the benefits of systematic engineering approaches – in those cases when requirements are relatively stable...

Contents

1. Introduction **1**
- 1.1. Problem . . . 1
- 1.2. Existing Approaches and their Limitations . . . 3
- 1.3. My Approach . . . 4
- 1.4. Research Contributions . . . 6
- 1.5. Overview of this Thesis . . . 7

2. Requirements of Development Methodologies for Knowledge-Based Systems **9**
- 2.1. Development Methodologies . . . 9
- 2.2. From Knowledge to Knowledge-Based Systems (and back) . . . 11
  - 2.2.1. Knowledge-Based Systems . . . 11
  - 2.2.2. Roles and Artifacts in the Development Process . . . 12
  - 2.2.3. Knowledge Modeling as a Collaborative Process . . . 14
  - 2.2.4. Knowledge Modeling as an Evolutionary Process . . . 15
- 2.3. Requirements of Development Methodologies . . . 18
  - 2.3.1. Requirements of the Modeling Process . . . 18
  - 2.3.2. Requirements of the Modeling Languages . . . 19
  - 2.3.3. Requirements of the Modeling Tools . . . 20
- 2.4. Summary . . . 21

3. Object-Oriented Software Engineering **23**
- 3.1. Paradigms . . . 23
  - 3.1.1. The Four Principles of Object-Orientation . . . 23
  - 3.1.2. Objects, Components, Patterns, Architectures, Frameworks . . . 24
- 3.2. Processes . . . 26
  - 3.2.1. Systematic Engineering Processes . . . 26
  - 3.2.2. Agile Processes . . . 31
- 3.3. Languages . . . 36
  - 3.3.1. The Unified Modeling Language (UML) . . . 36
  - 3.3.2. Metamodels and the Meta Object Facility (MOF) . . . 38
  - 3.3.3. Object-Oriented Programming Languages . . . 39
- 3.4. Tools . . . 39
  - 3.4.1. Integrated Development Environments (IDEs) . . . 39

3.4.2. CASE Tools and Round-Trip Engineering . . . 40
3.5. Summary . . . 41

4. Knowledge Engineering 43
4.1. Paradigms . . . 43
4.1.1. Ontologies . . . 44
4.1.2. Problem-Solving Methods . . . 44
4.1.3. An Architecture for Knowledge-Based Systems . . . 45
4.2. Process . . . 46
4.2.1. Ontology Modeling Approaches . . . 47
4.2.2. Knowledge Engineering Methodologies . . . 48
4.3. Languages . . . 50
4.3.1. Ontology Specification with OKBC . . . 51
4.3.2. Languages for Problem-Solving Methods (UPML) . . . 54
4.4. Tools . . . 55
4.4.1. Protégé-2000 . . . 55
4.4.2. Other Tools . . . 58
4.5. Summary . . . 58

5. Overview of the XP.K Methodology 61
5.1. Paradigms . . . 63
5.1.1. Differences between Object-Orientation and Ontologies . . . 64
5.1.2. Semantic Transparency for Object Models . . . 66
5.2. Process . . . 68
5.2.1. The Values of XP.K . . . 68
5.2.2. The Principles of XP.K . . . 71
5.2.3. The Practices of XP.K . . . 72
5.2.4. Putting it all Together . . . 77
5.3. Languages . . . 79
5.4. Tools . . . 82
5.4.1. Ontology (Class) Editors . . . 82
5.4.2. Knowledge Base (Instance) Editors . . . 83
5.5. Summary . . . 84

6. KBeans: Implementing XP.K in Java 87
6.1. Reflection and JavaBeans . . . 88
6.2. KBeans: Adding Facets to JavaBeans . . . 89
6.2.1. Dynamic Facet Declarations . . . 91
6.2.2. Static Facet Declarations . . . 92
6.2.3. Coding Conventions . . . 93
6.2.4. Accessing and Processing Facets . . . 94
6.3. A Catalog of KBeans Facets . . . 95
6.4. Application Scenarios . . . 96

6.4.1. Implementation of Semantically Rich Domain Models 97
6.4.2. Ontology Implementation 98
6.4.3. Ontology Sharing and Mapping 99
6.4.4. Constraint Checking 100
6.4.5. Semantic Annotation of Reusable Components 101
6.5. Discussion 102
6.5.1. Related Work 102
6.5.2. Benefits and Limitations of KBeans 104
6.6. Summary 105

7. KBeansShell: A Generic Framework for Knowledge Modeling Tools 107
7.1. Architecture 108
7.2. Standard Modules 112
7.2.1. Persistence Modules 112
7.2.2. View Modules 113
7.3. Using KBeansShell 120
7.4. The MetaKBeansShell 121
7.5. Summary 123

8. Case Studies 125
8.1. The Clinical Decision Support System ANIS 125
8.1.1. Domain Knowledge Acquisition in ANIS 126
8.1.2. Problem-Solving Methods in ANIS 128
8.1.3. The Knowledge Acquisition Bottleneck 128
8.1.4. A Rule Engine as a Byproduct of ANIS 129
8.2. The Clinical Multi-Agent System AGIL 130
8.2.1. Process Knowledge Acquisition in AGIL 132
8.2.2. Agent Implementation and Test 135
8.2.3. Discussion of XP.K for Agent Development 139
8.3. Summary 141

9. Discussion and Conclusion 143
9.1. Discussion 143
9.2. The Application Domain of XP.K 150
9.3. Open Issues and Potential Future Work 152
9.4. Conclusion 153

Bibliography 155

Appendix 170

A. The Standard Facet Types of KBeans 171

B. A Sample KBeans Ontology Class **183**

C. A Sample KBeans Constraint Class **191**

D. The Facets of XML Schema **193**

E. The BeanModel Java interface **195**

F. Class Diagram of the Meta Object Facility (MOF) **197**

G. Java Libraries for Knowledge-Based Systems and Artificial Intelligence **199**

H. The KBeans Logic package **201**

1. Introduction

My goal is to help make the development of knowledge-based systems more efficient. For that purpose, I will present a new software and knowledge engineering methodology, called *XP.K* (*eXtreme Programming of Knowledge-based systems*).

In this introductory chapter I will provide a brief overview of this document. First I will clarify the problem domain by describing the requirements of a typical knowledge-based system (section 1.1). Then I will argue that existing development methodologies for such systems are often inefficient, because they fail to produce systems which are easy to change (section 1.2). The XP.K methodology reduces the cost of change by using an evolutionary agile process based on *eXtreme Programming* (XP) [8]. I will outline XP.K in section 1.3. Finally, I will list my main contributions to research (section 1.4) and provide an overview of the structure of the remaining document (section 1.5).

1.1. Problem

In order to introduce the problem and application domain of this document, I will start with a simple example scenario. This scenario is based on various research and development projects that I have been involved in, especially the projects ANIS [108, 76] and AGIL [105, 167, 102]. Later chapters will refer to parts of this scenario.

Let's assume our task is to develop a computer system that supports decision-making in a local hospital. As indicated in figure 1.1, this hospital has various departments, such as the reception, the ward, the operating room, and the laboratory. These departments share a hospital-wide computer network with a central patient database. This database is permanently updated with various types of patient data, such as administrative data, blood sample tests, and measurements of blood pressure and heart rate. Due to the huge amount of data, many clinical decision-makers find it hard to get a sufficiently precise overview of the patient's state. Therefore, the system to be developed shall permanently monitor the patient's data, detect patterns of important patient variables, pro-actively provide the responsible clinical staff with a suitable overview of the patient's state, and generate proposals for future treatment plans.

In order to fulfill these requirements, our system must be based on *knowledge*. Clearly, this knowledge must be somehow formally represented to allow the system to reason about it. I will call this formal body of knowledge the *knowledge model*. For example, a simple

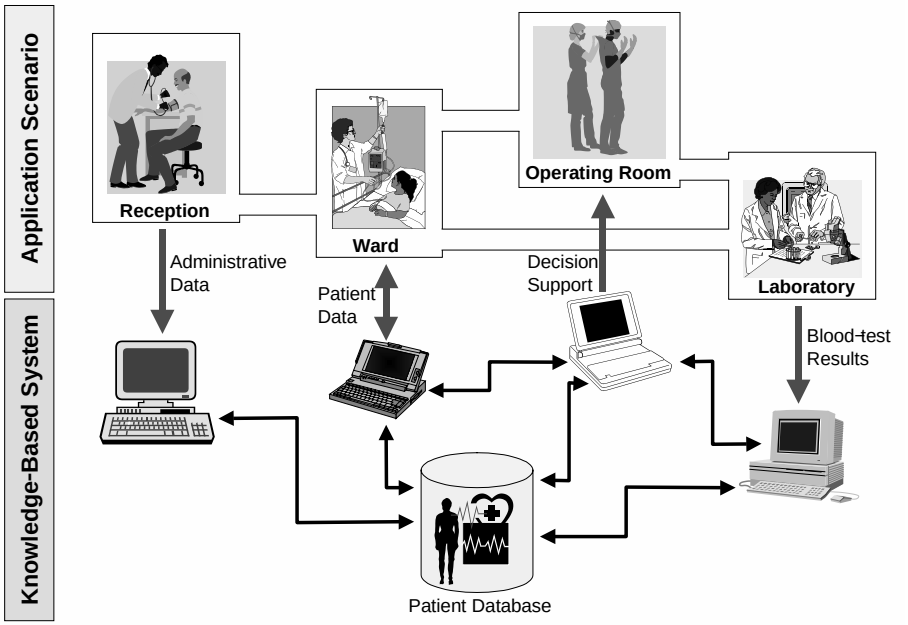

Figure 1.1.: An example scenario from the problem domain of knowledge-based systems.

medical knowledge model might contain information about symptoms, which indicate the presence of certain diseases, and about drugs or treatment plans which can be used to cure these diseases.

In the scenarios which are considered in this document, we assume that such knowledge can only be acquired by analyzing and modeling the clinical working processes and medical background knowledge of domain experts, such as anesthesiologists and nurses. We exclude machine learning methods like artificial neuronal networks, since they result in *sub-symbolic* [152] knowledge models that are far beyond the scope of this work. Instead, the modeling process itself is mainly performed by humans.

In chapter 2, I will show that this modeling process must necessarily be highly iterative, because knowledge – or the experts' representation of it – will change during model construction and after the system is exposed to data from tests or practice. Particularly in specialized domains such as medicine, knowledge models will change frequently due to the vagueness and complexity of knowledge. Furthermore, the distribution of knowledge among several experts and engineers with individual view points makes preplanning of the knowledge modeling process hard and requires close collaboration.

As a consequence, any development methodology for knowledge-based systems must provide methods, modeling languages, and modeling tools which support *change*, *feedback*, and *collaboration* efficiently. To summarize, the problem tackled in this document can be stated as follows:

Problem: How can we develop knowledge-based systems efficiently, given the

facts that knowledge models will need to be changed frequently in the face of feedback, and that collaboration is required in knowledge modeling?

1.2. Existing Approaches and their Limitations

In this section, I will suggest that existing development approaches for knowledge-based systems provide only limited support for changing knowledge models. The most important contributions to methodologies in this area originate from the fields of Software Engineering [168] and Knowledge Engineering [175]. Details on these methodologies are provided in chapters 3 and 4, but here is already a brief summary.

Software Engineering

Decades of research in Software Engineering have attempted to structure the art of ad-hoc programming into reliable modeling processes. The currently most widely used Software Engineering methodologies are based on the object-oriented paradigm, in which entities from the problem domain are represented by means of *objects*. Objects have *attributes* to store an entity's state and *methods* to modify or process object states. All objects are grouped into *classes*, which can be arranged in a subclass hierarchy. The premise of object-orientation is that classes and objects are an efficient and intuitive way of modeling. It is hoped that classes (and the software architectures they are embedded in) can be reused, saving development costs.

Various approaches for object-oriented modeling processes exist. Rather traditional approaches, such as Fusion [40] and the Rational Unified Process (RUP) [109], lead the development teams through various phases, typically including requirements analysis, system design, and implementation. These phases result in *models* which represent the problem domain from different view points. Various well-known languages for representing requirements, designs, and implementations exist, including the Unified Modeling Language (UML) [21] and Java [6]. These languages are particularly supported by modeling tools of industrial strength.

A fundamental benefit of Object-Orientation is that its languages and tools support relatively smooth transitions between informal analysis models and executable systems, because the basic modeling primitives (classes, objects, attributes, methods) are present throughout all phases. Despite this, many of the traditional development methodologies are often considered too "heavy", especially when requirements are uncertain and change frequently, because they enforce extra modeling and documentation efforts with high costs of change. As a response, there is a growing interest in so-called "light-weight" methodologies, such as Extreme Programming (XP) [8], which rely on simplicity and the rapid assembly of well-tested units, in the hope of reducing the cost of change.

All of these Software Engineering approaches aim at being applicable to *any* type of software and thus provide little specific support for the particular domain of knowledge-based systems. Especially, languages and tools for *knowledge* modeling are missing.

Knowledge Engineering

Knowledge Engineering is often regarded as a spin-off from Artificial Intelligence research. Its main goal is to structure the development and use of knowledge models. For that purpose, the most widely known Knowledge Engineering approaches (such as Common-KADS [164]) are based on the paradigm that knowledge should be represented in formal and explicit specifications (often called *ontologies* [79]). Ontologies capture the semantics of the knowledge in a format that is intended to be both easy to maintain and efficient to process by reasoning algorithms (often called *Problem-Solving Methods* [55]). The construction of both ontologies and Problem-Solvers is supported by various modeling methods, the phases and models of which resemble traditional Software Engineering approaches.

By separating ontologies from methods it is hoped to enable reuse of both domain and reasoning knowledge. However, practice has to date not produced much evidence that this reuse is feasible on a large scale. Especially the formalization of knowledge without having an application domain in mind has shown to be hard (an issue often called the *interaction problem* [24]). Furthermore, the transitions between high-level conceptual models and the implementation platform are insufficiently supported [10]. Finally, professional knowledge modeling tools are missing. For these reasons, methods from Knowledge Engineering are often too expensive to apply and – in contrast to approaches from Software Engineering – virtually not used in industry, even for the construction of knowledge-based systems [5, 27].

Joint Concepts of Software Engineering and Knowledge Engineering

Despite these apparent differences, research in both Software and Knowledge Engineering has independently lead to many similar results. Both regard the object-oriented paradigm as the most suitable compromise between computational efficiency and human intuition. Furthermore, both have in recent years attempted to find standard languages that can be widely supported by tools. Finally, both agree on the need to *iterate* during the modeling process. Just like any object-oriented design usually goes through many revisions, knowledge models need to be changed and updated frequently. These observations are the starting points of the approach that I will present in this document.

1.3. My Approach

In the previous section, I have indicated that results from both Software and Knowledge Engineering have their individual benefits and weaknesses. Software Engineering approaches are widely used but not optimized for the domain of knowledge-based systems. Knowledge Engineering research has so far mainly contributed to the theory of knowledge modeling. In order to gain a methodology that is both ready-to-use and optimized for knowledge modeling, I therefore propose to extend well-known Software Engineering technology with approaches from Knowledge Engineering.

I will argue that from the current Software Engineering approaches, the evolutionary, agile methodology of Extreme Programming is in many cases the most suitable platform,

since it is optimized for projects with frequently changing requirements. Extreme Programming is based on the four values Communication, Simplicity, Feedback, and Courage, which are well-suited for the needs of the knowledge-based system development process. *Communication* between the engineers and domain experts is essential, because only domain experts are able to build and evaluate knowledge models, whereas engineers need to transfer these models into an executable format. *Simplicity* prevents teams from wasting resources on modeling artifacts that are changed later anyway, especially the necessarily changing knowledge models. Simple models are also easier to communicate between domain and computer experts. The rapid availability of *Feedback* suits nicely to the experimental nature of knowledge modeling. Finally, *Courage* appeals to human creativity, which is an essential ingredient to make the other three values work.

I will now provide a brief overview of the elements of XP.K. This will necessarily include many forward references to concepts that I will elaborate in chapters 5, 6, and 7.

Modeling in XP.K is based on the object-oriented paradigm. Objects are close to the expert's intuition and can be efficiently implemented and processed. Furthermore, Object-Orientation simplifies the integration of industrial standard processes, languages, and tools.

The XP.K modeling process is characterized by a close collaboration between software and knowledge engineers and domain experts. XP.K promotes the value of *humility* to enable a fair collaboration between these distinct groups. The knowledge metamodel (i.e., the ontology) is jointly designed by engineers and experts, who try to agree on a language that is both sufficiently efficient and still close to the structure of the problem domain. The intended semantics of the ontology concepts are specified in test cases and semantic constraints. The specific instances of the ontology concepts (the knowledge base) are produced by pairs of domain experts using a knowledge acquisition tool which allows to run the test cases and pro-actively reports inconsistencies. Prototypes are created frequently and exposed to simulated or real test environments.

XP.K ontologies are specified in a simple object-oriented metamodel, which provides means for representing classes, attributes, relationships, and constraints, and thus resembles frame-based [127] Knowledge Engineering languages. This metamodel allows to customize the knowledge representation format for the domain with modeling primitives such as hierarchies, flowcharts, or if–then–else expression trees. The ontologies are modeled in UML and automatically translated into the object-oriented programming language that is also used for the remaining components of the system. Thus, the transitions between analysis, design, and implementation are rapid and interactions between the knowledge model and other classes can be efficiently implemented. Reverse engineering and object-oriented reflection is used to extract the ontology at both build-time and run-time. This enables the developers to leave almost all information about the ontology in the source code. Run-time reflection allows to (re-)use generic algorithms and tools that operate on *any* ontology. XP.K technology has been optimized for the programming language Java, but is also applicable to other object-oriented languages such as C# and Smalltalk.

As I will show in chapter 3, Extreme Programming is based on a number of practices.

In chapter 5, I will show that extending XP to knowledge-based systems means to apply these practices to knowledge modeling. The XP practice of Pair Programming has its counterpart in *Joint Ontology Design*. The XP practice of maintaining little documentation apart from the source code has its counterpart in *Round-Trip Engineering of Ontologies*. The XP practice of rapid prototyping is supported by *reusing* generic knowledge models and knowledge acquisition tools, and by using *industrial standards* whenever possible. The XP practice of unit testing means to use *Constraint Checking* during knowledge acquisition and to write *test cases*. XP is generally considered to be restricted to small to medium-sized projects only. I expect this restriction to hold for XP.K, too.

My solution to the problem of efficiently developing knowledge-based systems can be summarized in the following thesis.

> **Thesis:** A considerable class of small to medium sized knowledge-based systems can be efficiently developed with XP.K, the Extreme Programming approach presented in this document. The approach is based on the four values *Simplicity*, *Community*, *Feedback*, and *Courage*, and applies object-oriented Round-Trip Engineering to knowledge modeling.

This thesis contains some vagueness. Which "considerable class" of systems is covered by this methodology? What does "efficiently" mean? To clarify these questions, a catalogue of criteria that projects should fulfill is provided in chapter 9, and some criteria of efficient software development are listed in chapter 2.

1.4. Research Contributions

The major contributions of this research are (ordered by chapters):

- A *comparing analysis of approaches from Software and Knowledge Engineering* (chapters 3 and 4). My main contribution here is to show that existing approaches are often too rigid and inflexible for projects that involve knowledge modeling.
- A *new, agile methodology* for the development of knowledge-based systems (XP.K, chapter 5). My main contribution here is to identify Extreme Programming as a suitable base methodology for knowledge-based systems, and to introduce various important adaptations of its practices to optimize for the specific demands of knowledge modeling.
- A pragmatic, ready-to-use *framework for implementing knowledge-based systems in Java*, consisting of the following:
  - *KBeans*: A *mechanism for the representation and evaluation of metadata (particularly constraints) on Java objects*, which is useful to specify the semantics of knowledge models, but also for other types of software components in general (chapter 6). KBeans is supported by an efficient constraint engine in Java which can be ported to other object-oriented languages as well.

  - *KBeansShell*: The architecture and implementation of a flexible *framework for knowledge acquisition tools* based on KBeans (chapter 7). This complex software is freely available and has been successfully used in various projects.

- Further *evidence for the usefulness of Extreme Programming* by applying most parts of it to case-studies from the domain of knowledge-based systems (chapter 8). A mayor contribution here is to bring forth particularly strong evidence that this approach is suitable for multi-agent systems.

It is important to note that a common problem of works on software methodologies is that it is hard to provide conclusive evidence or "proofs" that any given methodology fulfills its promises. Some of my theories must therefore be based on practical experience. However, I did not have the resources to check all aspects of XP.K, because they can only be tested in teams of at least six programmers and domain experts. Furthermore, my ideas grew out of existing research projects, with no initial idea of XP in mind. Therefore, this document can be regarded as a discussion to help decision-makers whether to adopt XP.K in their projects or not.

1.5. Overview of this Thesis

This document describes the design rationales behind the XP.K methodology and details on an implementation of it in Java. As I will show in the following chapter, methodologies can be described along the five dimensions of (modeling) paradigms, processes, languages, tools, and applications in case studies. As indicated in figure 1.2, the chapters and sections of this document are arranged along these five dimensions.

Chapter 2 provides some *background on the problem domain* of developing knowledge-based systems. It especially stresses the necessity of changing knowledge models during the development process.

Chapter 3 reports on state-of-the-art technology from the field of *Software Engineering*, including background on object-oriented modeling and an introduction to the basics of Extreme Programming.

Chapter 4 reflects on research results from the domain of *Knowledge Engineering*. This will introduce the notion of ontology and its role in a general architecture of knowledge-based systems.

Chapter 5 provides an *overview of the XP.K methodology*, including the application of Extreme Programming to knowledge modeling, a generic object-oriented metamodel, and some implementation aspects.

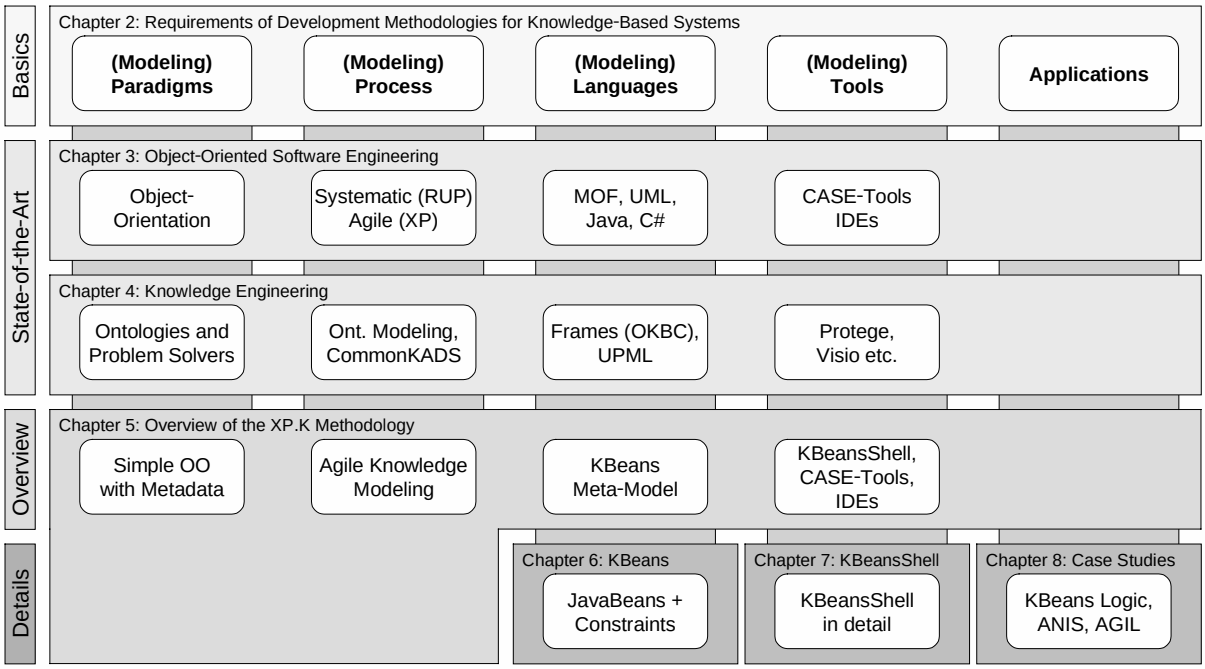

Figure 1.2.: A map of this document.

Chapter 6 demonstrates how XP.K can be put into practice using Java. This will introduce the knowledge modeling language *KBeans* which extends the JavaBeans component model by primitives for the declaration of semantic constraints.

Chapter 7 presents a Java-based framework for knowledge modeling tools, called *KBeansShell*. This framework can be employed for object (instance) modeling, as well as meta-modeling on class level.

Chapter 8 reports on some research prototypes that have been developed with parts of the XP.K technology. Among others, a clinical decision-support system for anesthesia and a multi-agent system for pro-active information provision are presented.

Chapter 9 discusses strengths and weaknesses of XP.K and concludes with a recapitulation of the basic results of this document. It also points at potential future work.

The appendix provides some information on implementation details that are not required for understanding the main text.

2. Requirements of Development Methodologies for Knowledge-Based Systems

In this chapter, I will explore the requirements that any development methodology for knowledge-based systems – including XP.K – should fulfill. First, I will identify the elements that constitute any software development methodology (section 2.1). Then, I will take a closer look at the nature of knowledge and the implications of this nature for knowledge modeling, namely that knowledge should be modeled *evolutionarily* and in a close *collaboration* between domain experts and engineers (section 2.2). This leads to a catalog of requirements for development methodologies (section 2.3) and a brief summary (section 2.4).

2.1. Development Methodologies

Many definitions of *methodology* can be found in the literature [35, 75, 112, 164]. In this document I will adopt the view of Alistair Cockburn [35], who defines a methodology as "an agreement of how multiple people will work together. It spells out what roles they play, what decisions they must reach, how and what they will communicate". Note that even in very small projects with only one developer there will be multiple roles involved such as contractors and end users.

Figure 2.1 provides some details on this definition by relating the basic building blocks of a methodology to each other. Here, the goal of any methodology is the construction and delivery of models that satisfy given *quality* criteria. These models, or *deliverables*, which obey a variety of notational *standards*, typically include system requirements, a technical design, and (hopefully) the executable system. Since the production of executable software is the main goal, it is fair to say that software construction is basically a modeling process. This process is performed by *teams*, the members of which have certain *skills* that enable them to take *roles*, such as project managers, class designers, and programmers. The team members use *tools*, apply various *techniques* (e.g., Joint Application Design, Java Programming, or Use Case Modeling), and perform several types of *activities* (e.g., meetings, tests, or reviews). Finally, any methodology is based on *values* which are accepted by the teams, such as the underlying philosophies of software development (e.g., simplicity, rapid

prototyping).

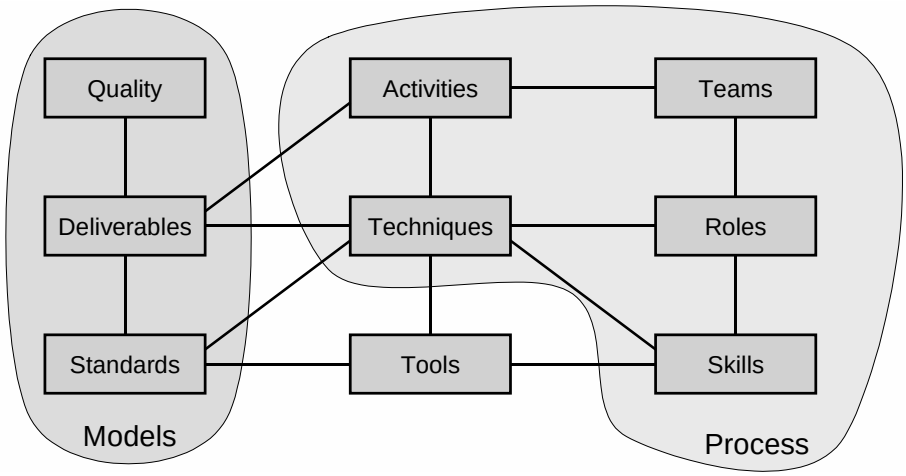

Figure 2.1.: The elements of software development methodologies according to Alistair Cockburn [35]. Here, the elements are loosely grouped into clusters of those relating to the models, and those describing the process of modeling.

For the pragmatics of a (textual) thesis, it is necessary to map Cockburn's conceptual network onto a linear sequence of paragraphs and sections. For that purpose, I have loosely grouped the elements of Cockburn's definition into the clusters "models" and "process" (figure 2.1). Both aspects are equally important, and it is hardly possible to describe any of each independently, since they have a mutual impact. For example, the use of a semi-formal knowledge specification language requires to add a formalization activity into the process, before the system can be tested. Any discussion of process must therefore be preceded by a definition of the models which are the intended results of the process, and any discussion of models must be accompanied by techniques that guide through the process of building models. My pragmatic path through the methodology jungle therefore starts with a discussion of the basic principles or *paradigms* that are behind the intended process and models. A sample paradigm is Object-Orientation. After that, I will proceed with the elements of the modeling *process* (including roles and techniques) and the details of the modeling *languages*. Finally, I will discuss modeling *tools*, which depend on both process and languages (figure 2.2). This leads to the following definition.

> **Definition (Development Methodology)**: A *development methodology* is an agreement of how multiple people will work together. It defines a *process* in which the development team will build *models*, including the executable system. These models are built in modeling *languages* with suitable modeling *tools*. Processes, languages and tools are based on modeling *paradigms*.

It is widely agreed [112], that no single methodology can cover all requirements from any type of project. The choice of a methodology depends on factors such as the number of

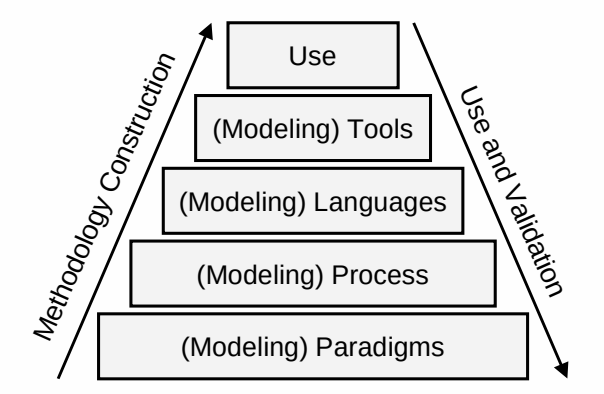

Figure 2.2.: The notion of methodology used in this document (cf. the "Methodological Pyramid" by Schreiber et al. [164] and the triangle by Partsch [144, page 14]).

people involved and the criticality of the resulting system [35]. Therefore, projects should be able to configure existing methodologies to better meet their specific requirements, or combine existing methods, languages and tools from different approaches. Chapter 9 will list some criteria that will aid managers with their decision whether and how to adopt an XP.K-based methodology in their project.

2.2. From Knowledge to Knowledge-Based Systems (and back)

With these general elements of a software development methodology in mind, I will now take a closer look at the specific class of software which is the topic of this document, namely knowledge-based systems. I will first clarify what I mean with a knowledge-based system.

2.2.1. Knowledge-Based Systems

In the context of human intelligence, the notion of knowledge is very difficult to define, because the cognitive reasoning processes of our minds are not well understood [151]. In the context of computers, knowledge must not be confused with the terms "data" and "information". *Data* are uninterpreted signals, such as the value of 39 degrees on a thermometer scale. *Information* is data equipped with meaning, e.g. that the patient Eve's body temperature is 39 degrees. In the context of this document, *knowledge* can be regarded as *information about information* (cf. [164]). A physician can use information about the patient's temperature to diagnose that she has a fever which may be caused by a flu.

A computer system that shall perform the same diagnosis task requires the physician's

knowledge in an executable format, i.e. a *knowledge model*. Since this knowledge model must be stored in the computer's memory in terms of bits and bytes, it is clear that the borderlines between data, information and knowledge are not clearly marked. Knowledge can – to a certain extent – be represented as data. By integrating such data into a piece of software, the vision of Artificial Intelligence is to let the computer solve tasks that are typically assigned to human experts. This vision has partially been fulfilled for very specific tasks such as business decision-support, clinical patient monitoring, product configuration, and fault diagnosis, for which (at least prototypical) knowledge-based systems have been built. As already mentioned in chapter 1, I focus on systems that possess knowledge models which are explicitly built and maintained by humans. I therefore define the following.

> **Definition (Knowledge-Based System)**: In the context of this document, a *knowledge-based system* is a computer system that solves tasks using an explicit and maintainable *model of human expertise*.

Agents [93] are autonomous knowledge-based systems which are embedded in an environment (e.g., a computer network) and which communicate (in so-called *multi-agent systems*) to solve problems. Due to the growing importance of agent technology, I will in this document regularly point to the applicability of XP.K to multi-agent systems.

Apart from the knowledge model, knowledge-based systems consist of many other components, such as graphical user interfaces, clinical device controllers, reasoning algorithms, database managers, and network connectors [191]. The program code of user interfaces alone typically accounts for 30–50 per cent of the system size [17, 130]. It is therefore essential to discuss knowledge-based systems in the broader context of Software Engineering and not only from an Artificial Intelligence perspective.

2.2.2. Roles and Artifacts in the Development Process

Backed by this notion of knowledge-based system, I will now identify the essential roles of people involved in the development process, and define an appropriate terminology for the rest of the document.

- *Domain experts* (e.g., medical doctors) provide the domain knowledge either directly or by pointing to other knowledge sources like text books. Experts either edit the knowledge base themselves or support the engineers in the knowledge acquisition task. Furthermore, they are able to define and evaluate test and application scenarios.
- *Knowledge engineers* are the main link between the domain experts and the (technical) system. Their main tasks are to select or adjust a suitable knowledge modeling language, to supervise or support the experts' knowledge acquisition process, and to ensure that the resulting models can be processed by the computer system.
- *Tool developers* build or adapt tools that enable the domain experts and knowledge engineers to model knowledge in the modeling language. If only standard tools are (re-)used, the role of the tool developer is vacant.

- *System developers* build the overall system by integrating the knowledge modules with other components.

Figure 2.3 summarizes the relationships between these roles and some of the artifacts involved in a typical development process for knowledge-based systems. The various artifacts are defined as follows.

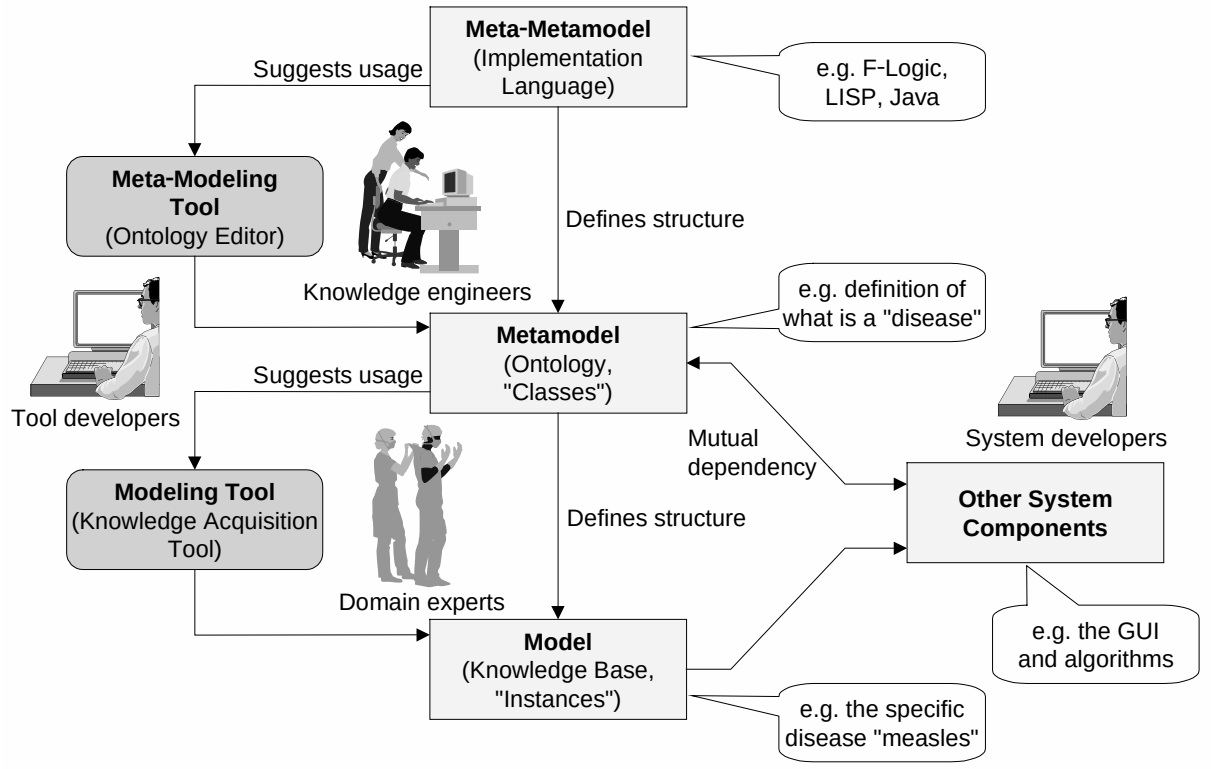

Figure 2.3.: The principal roles and artifacts typically involved in the development of a knowledge-based system.

- The *model* (interchangeably called *knowledge base*) represents domain expertise in a machine-readable form. This model can, for example, contain logical expressions which describe the symptoms of a disease or process patterns which represent clinical treatment plans.

- The structure of such a model is specified by means of a *metamodel* (or *ontology*). The metamodel specifies the knowledge modeling language used by the domain experts and engineers. For example, it may define the attributes of the entity class “disease”.

- The metamodel itself is specified in yet another (knowledge) meta-modeling language. This is the *meta-metamodel*, or ontology specification language, such as frame-logic,

LISP, or Java. Note that meta language and meta-meta language might be the same, if the distinction between model "instances" and "classes" is not clearly marked. I will return to this issue in chapters 3 and 4, but until now it suffices to regard both as "modeling languages".

- *Modeling tools* are used to visualize and edit the (meta) models. The definition or choice of a metamodel has a significant impact on the way the modeling tool will or should be used. If the model's metamodel suggests to formalize knowledge in a textual form, simple word processors (e.g., Prolog editors) can be sufficient. However, a knowledge base founded on a metamodel with a mostly hierarchical structure should be edited and visualized graphically in trees.

- Finally, the executable knowledge-based system consists of *other components*, such as reasoning algorithms ("inference engines") and user interfaces. These components use the knowledge base for deriving and displaying information for the system's end users. There is a mutual dependency between the metamodel and these components, which will be elaborated in the context of the so-called *interaction problem* in chapter 4.

2.2.3. Knowledge Modeling as a Collaborative Process

After having clarified the terminology, we can now take a closer look at the roles involved in knowledge modeling. To start, it is important to note the principal difference in the attitudes and goals of domain experts and knowledge engineers (cf. [152, page 182]):

Domain expert's logic: Domain experts are usually oriented towards the individual case of their daily working processes, e.g. the individual patients. Their knowledge is optimized for solutions that are appropriate for the given situation. They try to consider as many factors as possible and are tolerant against inconsistencies.

Knowledge engineer's logic: Knowledge engineers try to identify global solutions, which are appropriate and legitimizable for all possible contexts. They aim at obtaining knowledge models which are transparent, objective, and which consider a finite number of factors.

Despite these different logics, knowledge engineers require the domain experts' agreement to cooperate and communicate, because the engineers usually do not possess the domain expertise needed for building valid knowledge bases. By the way, this is one of the main differences between knowledge-based systems and other types of software, where the engineers are usually able to acquire a sufficiently deep understanding of the problem domain, so that they can build the system with little or no further assistance by domain experts.

Apart from the distribution of knowledge between domain experts and engineers, complex and highly specialized domains such as medicine are further characterized by a distribution of knowledge between domain experts. Specialists for anesthesiology will rarely

presume to build knowledge models for cardiac surgery. Different experts – even from one and the same discipline – will have their own personal preferences and mental models. In this context, the educational psychologist Gavriel Salomon [161] points at the mutually-fertilizing value of collaboration:

> "Knowledge is commonly socially constructed, through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives."

These different perspectives will not only improve the quality of the resulting models, but also ensure that the models will meet the requirements from different user groups, especially from both the technical and the application domain. Domain experts must ensure that the system will be accepted and trusted by their peers. For example, the rather conservative user group of medical doctors will reject a clinical decision-support system which is solely designed from an engineer's perspective.

For these reasons, knowledge modeling must be heavily based on communication and will usually require compromises. In this context, Rammert et al. [152, page 139] state that models are "negotiated in a social relationship". This negotiation is often difficult, and experience shows that the bottleneck of building knowledge models lies more in the social process than in the technology [46].

2.2.4. Knowledge Modeling as an Evolutionary Process

In the context of knowledge modeling, it is beneficial to take a closer look at the human cognition process, because knowledge first has to be built up in a domain expert's mind before it is ready to be modeled. Bernd Schmidt [162] regards human cognition and scientific theory construction as iterative processes (figure 2.4). In his view, cognition is based on the construction of theoretical models that are exposed to experimental data from real or simulated worlds. His view leads to the important observation that *human cognition is driven by feedback*. Theories must be validated or updated if new observations are made. The experimental acquisition of case data is essential in many scientific disciplines, such as chemistry and medicine. Furthermore, the choice of experiments and the construction of simulation models has an impact on the resulting theoretical models.

As indicated in figure 2.5, we can transfer Schmidt's model of human cognition to the domain of building knowledge-based systems, if we regard knowledge modeling as a kind of theory construction. Here, human experts have to construct formal theories about the domain, backed by knowledge which either resides informally in their heads, or which can be acquired from some other knowledge source. The resulting knowledge model is part of a knowledge-based system which can operate in real or simulated worlds. Tests in both worlds produce feedback which allows the domain expert to revise the knowledge models. When installed in the real application scenario, the system even changes the real world and thus produces new requirements, which recursively suggest changes to the knowledge model. In terms of cybernetics, knowledge-based systems are *open systems* [185], which

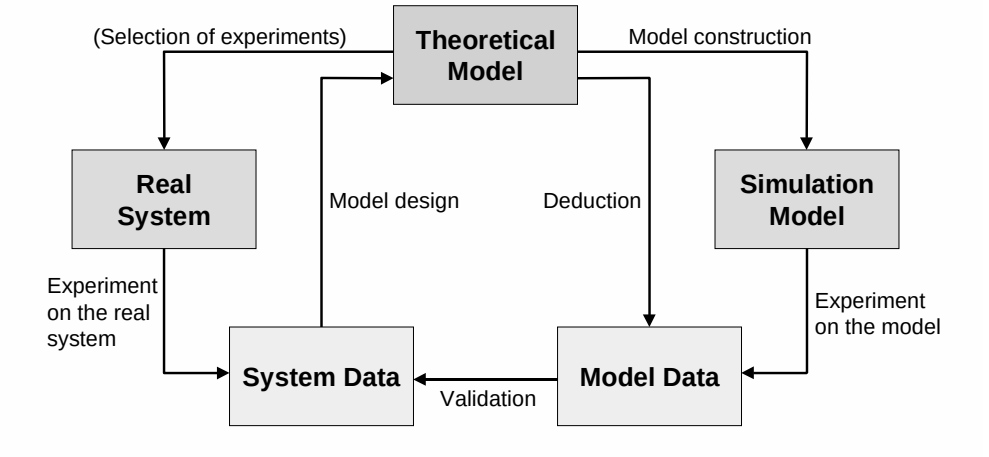

Figure 2.4.: The scientific cognition process is based on the construction of theoretical models that are exposed to data from the real or simulated worlds (adapted from [162])

can not be separated from the surrounding environment and are therefore inherently hard to predict.

There are various other reasons why knowledge models will almost necessarily change while the knowledge-based system is built and used. (Many of the following reasons are quoted from the comprehensive German study "Construction and Application of Expert Systems – Consequences for Knowledge, Communication, and Organization", the results of which are collected by Rammert et al. [152].)

- Finding requirements is hard [44]. First, since we do not understand how humans carry out reasoning tasks, it hardly possible to set out a detailed specification for software to imitate humans [168, page 12]. Second, the potential users are often unable to assess the benefits or usage scenarios of the new system, especially when they are inexperienced computer users. Third, the system modifies the work processes in which it is installed. In daily practice, users modify their environment and their use of the system, so that a new working culture emerges [152, page 41]. Any change of requirements implies that knowledge models must be updated.

- The knowledge acquisition process itself can not be completely planned, because the various developers and groups involved in the process face each other with different and unknown cognitive and social perspectives [152, page 158]. Furthermore, the behavior of distributed and highly interactive systems – such as multi-agent systems – is hard to predict [93].

- Knowledge models are often based on wrong assumptions. This is because knowledge modeling requires the domain experts to transparently expose their daily practice,

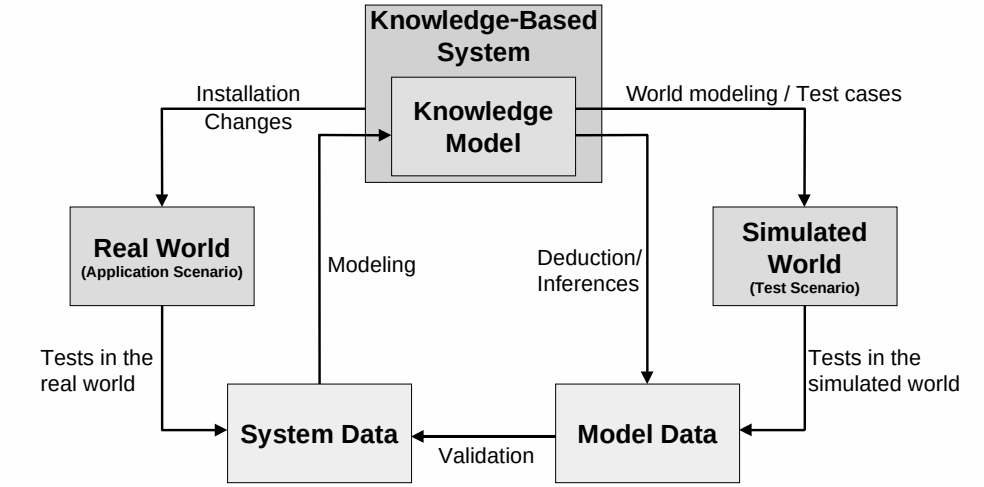

Figure 2.5.: The knowledge modeling process can be regarded as a kind of theory construction, comparable to scientific cognition processes (figure 2.4).

but this "practice necessarily operates with deception" [152, page 179]. Furthermore, every model is only an approximation of reality [175] and the actors involved in the modeling process speak different "languages".

- Knowledge – especially in non-deterministic domains such as medicine – is inherently *complex* and *vague* [152, page 163]. In contrast, computers require formal and evaluable data structures, e.g. threshold values of patient observables. Experts will tend to use trial-and-error methods to determine such thresholds, until the system exposes the expected behavior. Furthermore, scientific progress might question the beliefs reflected in a knowledge base.

- The knowledge modeling process itself produces new knowledge. The self-observation performed during analysis of the existing work processes can lead to new insights [152, page 11]. For the new medium, knowledge is being translated and reorganized. It evolves in the process of being encoded and formatted for the system [152, page 11]. The existing work processes are challenged when analyzed ("Redesign during modeling" [152, page 183]).

- Often, the installation of knowledge-based systems requires to "digitize" the data flow in the process. For example, prior to installing an intelligent information system in a hospital, an automated, digital access to the patient database is required. This leads to a co-evolutionary behavior of the system and the automated data acquisition devices it is connected to [152, page 109].

- Finally, the communication involved in the knowledge modeling process will also influence the resulting models. "Knowledge can not be mined and processed like a raw

material, but rather comes into existence during the communication" [152, page 10]. However, this communication process is characterized by reciprocities between engineers and experts, and the information provided by the expert depends on the context [152, page 163]. As a domain expert gets more and more used to the formal view of the knowledge engineer, she will adjust her modeling style, and vice-versa.

2.3. Requirements of Development Methodologies

So far, I have identified the four basic elements of any methodology and pointed at the importance of collaboration and evolution in knowledge modeling. In this section, I will define criteria or requirements against which development methodologies (including XP.K, and the Software and Knowledge Engineering approaches from the next two chapters) can be evaluated.

These methodologies must be evaluated in the light of the *quality* of their intended results – the knowledge-based systems. Since knowledge-based systems are a special type of software, the general criteria of software quality are relevant for them as well. According to Sommerville [168], these criteria are *maintainability*, *dependability* (reliability, security), *efficiency*, and *usability*. For knowledge-based systems, maintainability is especially important, since it should be possible to evolve the software to changing requirements and knowledge models. Furthermore, reliability (e.g., correctness of the models) is important, especially in critical domains such as hospitals. An orthogonal aspect to these criteria is the *price* of the system, since a low budget can lead to quality cutbacks in all other areas.

In the following subsections, I will list some requirements for development methodologies along their dimensions process, language, and tools. All of these dimensions need to be mutually consistent, and follow a common paradigm. For example, the tools must understand the modeling language, and the process must employ the tools and languages in an adequate way.

2.3.1. Requirements of the Modeling Process

Sommerville [168, page 9] lists several common-sense criteria of good process models. In his view, processes should be understandable, supportable by tools, acceptable, reliable, robust, maintainable, visible (traceable), and rapid. Because knowledge-based systems are in general an "experimental technology" [152, page 23], *flexibility* (i.e., robustness and maintainability in case of unexpected problems) is essential for the process. Furthermore, *rapid* processes ensure that prototypes and other traceable results are produced regularly.

The consequences of the need for collaboration and feedback in knowledge modeling on suitable development processes are summarized by Rammert et al. at the end of their book [152, page 260]:

> "A major reason for the failure of the expert systems is the large gap between the world of developers and the world of users. It concerns the lack of reflection

> of this gap during development, and the missing feedback with the application domain. If the development of knowledge machines would attune to the application scenario early, for example with a evolutionary and recursive process of software development, wrong perfectionalism and premature rigidity would be prevented."

In other words, the development process of knowledge-based systems should consider the three essential factors *feedback*, *collaboration*, and *change*. Clearly, these factors suggest to follow an iterative, evolutionary process model which produces executable prototypes rapidly and frequently.

In anticipation of later chapters, I would like to point out that the preference of evolutionary process models does not necessarily discredit structured and systematic development methods. Most of the modern Software and Knowledge Engineering approaches presented in the review chapters 3 and 4 are in fact evolutionary – at least in a sense that they admit the need for iteration. However, they differ significantly in the duration of their iterations and their use of prototypes.

2.3.2. Requirements of the Modeling Languages

Any development process aims at producing models, and the choice of a suitable modeling language has a significant impact on the efficiency of the process. The following criteria may guide this choice. They are founded on a catalog of criteria from a Software Engineering perspective by Partsch [144, page 39f], and research from Knowledge Engineering (e.g. [41, 164]).

In the following, I will focus on requirements of *knowledge* modeling languages. Most projects will employ other languages – especially general-purpose programming languages – for components such as user interfaces, but the choice of these languages is beyond the scope of this discussion.

A major task of modeling languages is to support and simplify the modeling process, i.e. to efficiently lead from abstract mental models to executable systems. For that purpose, the languages should be *easy to learn* and have a richness of expression which is *adequate* for the domain. Suitable languages will reflect the terminology, grammar, and way-of-thinking of the domain experts, and yet be formal enough to support rapid feedback from prototypes. For that reason, languages and modeling paradigms with smooth transitions between high-level models and the executable system are more suitable than languages which must be manually translated. All languages employed by a project must be mutually compatible or *translatable*.

Translating knowledge models is furthermore required to map the models onto different view points, because the development team is an inhomogeneous group which consists of domain experts, knowledge engineers, and technicians. Each member of these groups might have a different mental image of a given model element. For example, a clinical doctor will think of a "symptom" in terms of observations and measurements, whereas a technician might regard a symptom as a boolean condition.

In order to reflect the evolutionary character of knowledge modeling, the languages should support *incremental change* and be tolerant against partially incomplete, *explorative* models. Languages should enable their users to build well-structured and compact models, which are easier to understand and change than unstructured, bloated ones.

In order to lead to reliable knowledge-based systems, modeling languages should have a precise and unambiguous syntax, and allow to check the models for consistency, precision, and completeness. Languages which are formal enough to allow to perform such correctness checks automatically by tools are more suitable than informal ones.

Finally, in order to support reuse, communication, and distribution of models, *compatibility* to existing standard languages is a considerable goal. Languages which are widely supported by trusted organizations should usually be given preference to prototypical ones defined by small research teams. Widely used languages provide access to a larger repository of tools, reusable (ontology) libraries, and knowledgeable developers. Compatibility also makes sure that models can survive transitions between the different modeling tools.

2.3.3. Requirements of the Modeling Tools

The tools employed by a methodology should fulfill general criteria of well-designed software. They should be *easy to use and learn*, especially if domain experts are confronted with them. This means that knowledge modeling tools should provide an intuitive and consistent (graphical) user interface, be reliable, et cetera.

In support of rapid prototyping and immediate user feedback, the tools should reduce the turn-around times between models and executable systems. Furthermore, the tools should grant assistance in the construction of correct models. Comparable to programming tools, which include compilers to check the syntax of the source code, knowledge modeling tools should pro-actively point their users to missing or wrong elements in the knowledge models (cf. [144, page 46]). Such a certain kind of *mindfulness* can relieve the users from standard tasks, and detect errors before they enter the system.

Since models and metamodels might change frequently, it should be possible to *adapt* the tools easily. If a tool fails to meet changing requirements, it should be possible to replace or extend some of its functionality, or to move to a different tool. The tools should allow to transfer and translate evolving models. For the sake of reducing the complexity of knowledge models and appealing to the user's creativity, tools should provide graphical components like graphs, trees, and forms [7]. Optional textual representations are required due to the "scaling-up problem" for visual modeling languages [23].

Last but not least, the modeling tools – as well as the whole methodology – should be *enjoyable* to work with. A motivated team will not suffer from movement of labor and expertise, and most probably communicate better.

In the real world, these requirements must be heavily weighed with the project resources time and money. The chosen metamodel suggests or even prescribes the way of using the modeling tool. As long as this metamodel is changing frequently, tool developers will usually not be able to deliver custom-tailored, high-quality editors for each project. In the

early stages of the project, tool users must therefore often take potluck with standard tools and other compromises. Once again, communication – this time between tool developers and domain experts – is a key factor in project success.

2.4. Summary

As illustrated in the upper part of figure 2.6, the challenge of developing knowledge-based systems lies in economically transforming expert knowledge into reliable and efficient systems. The development methodology employed for this transformation must consider the facts that knowledge is usually distributed, complex, and often vague. These facts and feedback from practical use will cause the model of expertise to change frequently. As a consequence, the development methodology should be based on a feedback-driven, communication-intensive, evolutionary process which operates on a maintainable system. As shown in the lower part of figure 2.6, such a process suggests criteria for the choice of appropriate modeling languages and tools.

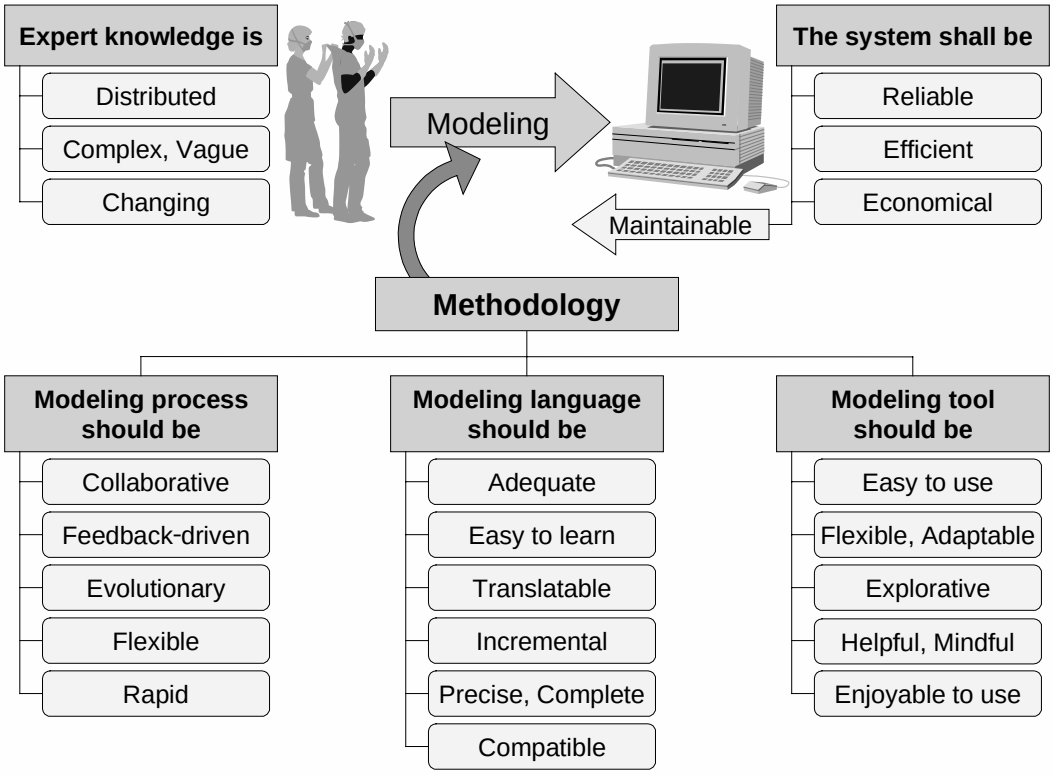

Figure 2.6.: The task of developing knowledge-based systems (upper part), and some requirements for development methodologies (lower part).

3. Object-Oriented Software Engineering

In this chapter, I will review state-of-the-art methodologies from Object-Oriented Software Engineering, and analyze their potentials and limitations for the development of knowledge-based systems. In accordance with the notion of methodology introduced in the previous chapter, I will structure the presentation of the approaches by their paradigms (section 3.1), process models (section 3.2), modeling and programming languages (section 3.3), and tools (section 3.4). Throughout this presentation and in the summary (section 3.5), I will evaluate the methodologies against the requirements defined in chapter 2.

3.1. Paradigms

The basic assumption of Object-Orientation is that any problem domain can be described in terms of things or *entities*, which have behavioral characteristics that represent what an entity "does", and structural characteristics that represent what an entity "is" and how it relates to other entities. According to this view, entities with common characteristics can be grouped into *classes*. The ancient Greek's *Theory of Forms* (cf. [172, page 183]), which shares many ideas with Object-Orientation [2], states that arranging entities into classes (or *Forms*) is an important way of achieving well-founded knowledge of the world. In other words, "to know the Form of a thing is to understand the nature of that thing" [2]. The Theory of Forms suggests that Object-Orientation is a natural and intuitive way of analyzing and modeling a problem domain. Object-Orientation is close to our own natural perception of the real world [117].

3.1.1. The Four Principles of Object-Orientation

Object-Orientation is founded on the following principles (cf. [34, 2, 144]):

- *Abstraction* is the formulation of models by focusing on similarities and differences among a set of entities to extract relevant common characteristics, ignoring those aspects that are not relevant to the current purpose. The main goal of abstraction is managing complexity.

- *Encapsulation* (often referred to as *Information Hiding*) facilitates abstraction, by hiding the details of a design decision in a packaged model element. An entity exposes what it is through a specification (or interface), and describes how it is realized by means of an internal implementation. Encapsulation keeps related content together, with the goal of reducing the cost of change.
- *Inheritance* is a mechanism for expressing similarity among entity classes. It allows to relate, reuse, and extend representations. The goal of inheritance is to reduce duplication and to prevent inconsistencies.
- *Polymorphism* means that different model elements can have the same specification, but different implementations. This means that the same message can trigger different operations, depending on the class of the target entity. Polymorphism allows to extend existing models with new elements, without having to change the elements that are already in the model.

These principles aim at capturing the world's complexity into maintainable models. The paradigms of encapsulation and polymorphism *reduce the cost of change*, and – together with abstraction and inheritance – support the *management of complexity* [2]. The following subsection will introduce the concepts that implement these four principles in the domain of Software Engineering.

3.1.2. Objects, Components, Patterns, Architectures, Frameworks

Object-Orientation regards structural and behavorial characteristics of entities as complete units. For that purpose, object-oriented models are centered around *objects*, which represent (abstractions of) items, persons, or notions [144]. An object describes its structural characteristics by means of *attributes* and *associations* (or *relationships*), and exposes its behavorial characteristics through *operations* (or *services*). Objects communicate with each other by passing *messages*, which cause the recipient to perform an operation, and to return a result to the sender. All objects are grouped into *classes*, which are arranged in an inheritance hierarchy.

Related to classes, the concept of components [147] is fundamental to modern object-oriented systems. A *component* is a reusable building block which can be (visually) plugged together with other components. For that purpose, a component exposes a list of *properties* and *services* that other components can link to. Although components are often implemented by a single class, they might also encompass multiple classes. The most widely used libraries of components contain graphical user interface elements like buttons, labels, and lists.

Besides the low-level modeling elements like objects and components, object-oriented methodologies also provide mechanisms to describe larger structures and best-practices. The so-called *Design Patterns* [67] document recurring solutions to common problems. Patterns have a context in which they apply and must balance a set of opposing consequences or *forces*. Patterns capture modeling experience from which others may learn,

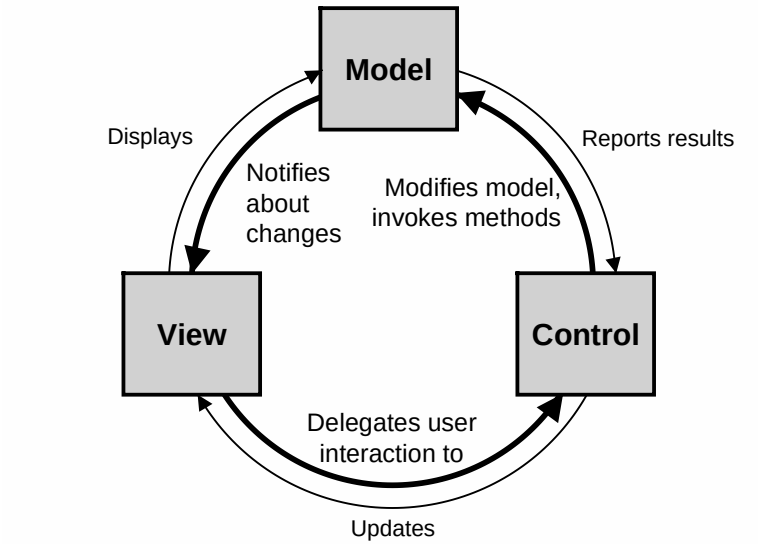

Figure 3.1.: The Model-Control-View architecture makes a clean distinction between objects that represent the domain model, the information display, and the user interactions.

and provide a vocabulary (or *Pattern Language*) which allows to communicate and discuss design decisions on a high level of abstraction.

An example will clarify the role of Design Patterns in Object-Orientation. The *Observer Pattern* [67] defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This is a widely used pattern in graphical user interfaces, which separate the visual components from the underlying application data. For example, each time new heart rate data are measured for a patient, the chart displaying the heart rate must be updated. Applying the Observer Pattern here means that the chart component registers itself as an "observer" of the object that manages the heart rate value, and the latter sends a notification message each time it changes.

Patterns, such as the Observer Pattern, can play an important role in the coarse-grained organization or *architecture* [68] of a software product. The so-called *Model-Control-View (MCV) Architecture* [73], which is illustrated in figure 3.1, is frequently used in modern object-oriented languages and will be picked up in the context of XP.K in later chapters. Modeling artifacts built along the lines of MCV use three types of objects to decouple business data (or domain knowledge) from the user interface. The *Model* objects manage the data, the *View* objects represent the visual elements, and the *Control* objects react on user input, e.g. by changing the state of the Model objects. Interactions between these objects are mainly based on the Observer Pattern. This separation of concerns simplifies system maintenance, supports reuse, and enables visual programming with components.

Related to architectures is the notion of frameworks. A *framework* is a collection of

several components with predefined co-operations between them [148]. Frameworks allow to reuse not only code but architectural design and therefore play an important role in rapid software development.

For the recent decade, Object-Orientation has become increasingly popular for many kinds of modeling tasks and is now the most widely used paradigm for software development. It allows to build abstractions of the world's entities quite intuitively, and – although a certain learning process might be required – often leads to clearly structured models [144, page 135]. Thus, object-oriented models *facilitate communication*, even between developers and little-trained domain experts [34].

3.2. Processes

In this section, I will evaluate modern software process models in the light of their ability to cope with change, to support communication, and to rapidly provide feedback. I will first review systematic engineering approaches (such as the *Waterfall Model*, *Fusion* [40], and the *Rational Unified Process* [109]) and show that they are often too heavy and inflexible for the domain of knowledge-based systems. Then I will introduce the ideas of the recent "light-weight" or "agile" approaches (such as *Extreme Programming* [8]), which try to achieve a greater flexibility by reducing the cost of change.

3.2.1. Systematic Engineering Processes

In the early decades of computing, software has mostly been created in an ad-hoc fashion. Chaotic software processes typically have a higher risk of failing to meet user requirements, of slipping schedule, of producing defect software, and of leading to systems that are becoming increasingly hard to maintain [168, 8]. With the goal of reducing this risk, research and practice of Software Engineering has produced a number of models for structured, systematic development processes [119].

The Waterfall Model

One of the most important achievements of early Software Engineering research was to recognize that software development should be performed in steps, with each step covering a certain development phase [144]. Software development was compared to traditional engineering disciplines such as civil engineering, in which engineers create a precise design, which is then transformed into a house or bridge [39]. The engineering view assumes that well-defined engineering rules and plans exist, which only need to be applied in the correct order to produce the desired software. Further, engineering assumes that the implementation process itself will be straight-forward, if only the design is correct.

The so-called *Waterfall Model* follows this engineering metaphor, by guiding the developers systematically through the phases of requirements analysis, design, implementation,

test, and maintenance. Each of these phases bases on the documents or models produced by its predecessors. Whereas the early analysis and design models only describe the coarse-grained, high-level architecture, the implementation models go into the smallest programming language details. Boehm [18] has shown that since the basic architecture is far more important than the implementation details, wrong decisions made early in the process are much more expensive than later decisions. This exponential rise of costs is illustrated in figure 3.2.

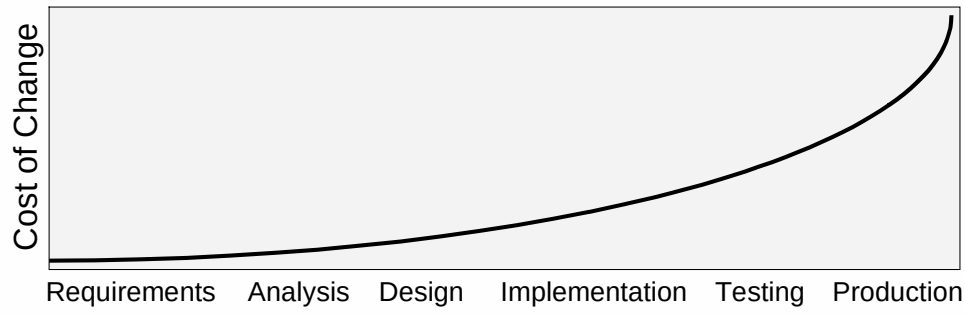

Figure 3.2.: Systematic Software Engineering methodologies are based on the premise that the impact of design decisions (and the cost of changing them) rises exponentially over time.

Although the Waterfall Model is still widely applied, especially by government agencies and large software procurers [168], its inflexible partitioning of the project into distinct stages is based on simplified assumptions which do not reflect the reality of software development [144]. It is unrealistic that all documents and models are complete and error-free upon the first creation. Especially the system requirements are often not clear before the customers can put their hands on a system prototype. Therefore, waterfall-based projects tend to do very much rework at the very end [110].

As a response, approaches such as Barry Boehm's *Spiral Model* [19] and the *V-Model* (cf. [168]) strengthen the role of feedback and evaluation of risk in the development process. The major progress of these approaches compared to the Waterfall Model is their capability of ruling out sub-optimal alternatives and errors early in the process. Their concepts had a significant impact on the current generation of systematic engineering methodologies, such as Fusion and the Rational Unified Process, which are described in the following.

Fusion

Fusion [40] is a detailed and generic object-oriented methodology, which has integrated and extended other existing approaches from the early 1990's, like OMT [159], Booch [20], and OOSE [90]. The goal of Fusion is to provide a direct route from a requirements definition through to a programming-language implementation. As illustrated in figure 3.3, this

route roughly traverses the cascades of the Waterfall Model, with its analysis, design, and implementation phases.

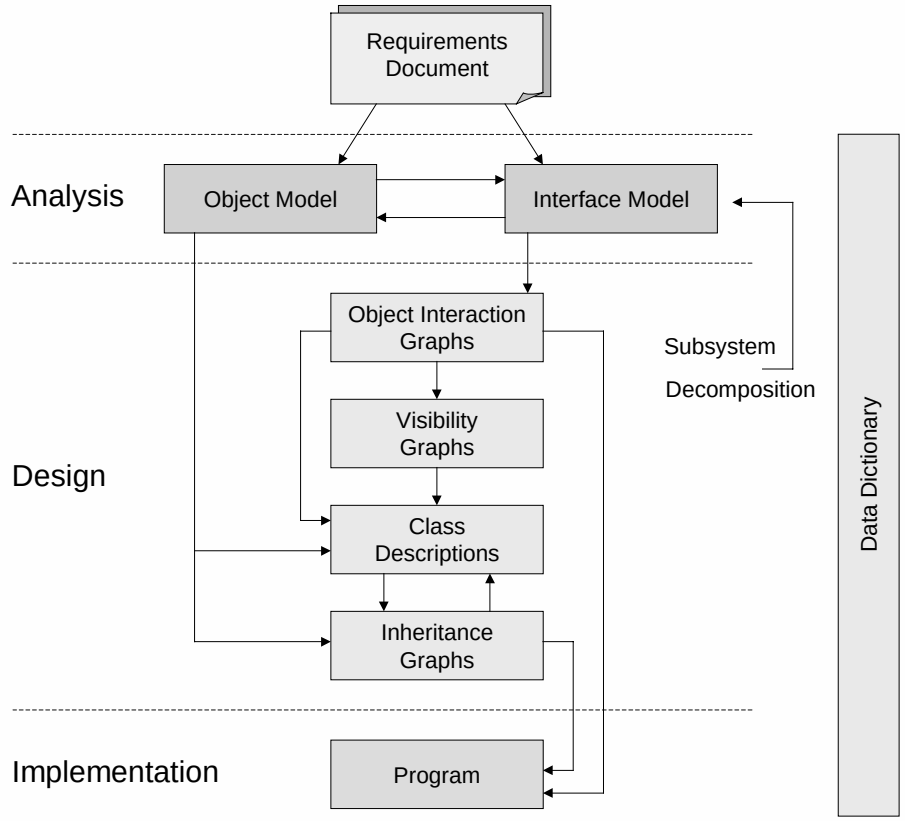

Figure 3.3.: An overview of the Fusion methodology. Fusion is an example for a methodology where systematic, comprehensive analysis and design activities are performed before coding and testing.

In the analysis phase, the informal requirements documents are translated into a declarative description of the desired system behavior. This uncovers the objects and classes in the system, their relationships, and the operations that the system can perform. The analysis models lay the foundation for the design phase, in which the developers decide how to represent the system operations by the interactions of related objects, and how those objects gain access to each other. The design model is geared to be detailed enough for a relatively straight-forward system implementation in an object-oriented language. Thus Fusion can be regarded as a waterfall-based methodology. However, Fusion assumes realistically that any of the modeling artifacts can change, and therefore allows the developers to iteratively step back to any phase in the development cycle.

Its well-structured process with successive models and comprehensive formal notations suggests that Fusion is an attractive methodology for projects in which the initial requirements are relatively fixed. For example, I have applied Fusion for a successful redesign of

an existing expert system for my Master's Thesis [100]. However, feedback from industrial practice (e.g., `http://c2.com/cgi/wiki?FusionMethodology`) reports that Fusion – as well as other waterfall-based approaches – requires a considerable overhead to keep the various modeling artifacts in sync when requirements or design decisions have changed. A change in Fusion's Object Model must be passed on to the Class Descriptions, Inheritance Graphs, and the program – and vice versa. These changes are expensive and slow down the process. Therefore, the exponential cost curve from figure 3.2 is not only an assumption, but also a *consequence* of waterfall-based processes.

To summarize, its lack of support for frequently changing models and the resulting delay of prototypes make Fusion – as well as related methodologies where extensive analysis and design processes take place before coding and testing – rather unsuitable for developing knowledge-based systems.

The Rational Unified Process (RUP)

The *Rational Unified Process* (RUP) [109] is a commercial process product developed and maintained by the world-leading CASE tool vendor Rational Software Corporation. RUP is supported by the influential "Three Amigos" Booch, Jacobson, and Rumbaugh, who were also involved in the definition of the Unified Modeling Language (UML, subsection 3.3.1).

RUP is basically a meta-methodology (or process template), which must be adapted to specific project circumstances. RUP comes with several out-of-the-box roadmaps for various common types of software projects. It also defines a uniform terminology for artifacts, activities, roles, et cetera and gives advice on tool use.

With the goal of reducing risk, RUP emphasizes the adoption of certain "best practices" of modern software development [146], such as the use of component-based architectures and visual modeling. The best practices are applied in an incremental, iterative process, in which requirements can be refined as the project evolves. As illustrated in figure 3.4, the process is divided into the four coarse-grained *phases* inception, elaboration, construction, and transition. Each phase is split into one or more *iterations*, in which *activities*, such as business modeling, analysis, or implementation, are performed in varying levels of detail. For example, the amount of resources put into testing activities increases, when the project moves from the construction to the transition phase.

The beginning of an activity is not bound to the end of another, e.g. design does not start when analysis completes, but the various artifacts associated with the activities are revised as the problem or the requirements are better understood. RUP can therefore be regarded as a generalization of the Waterfall Model.

RUP is not meant to be document-driven: Its main artifact must remain, at all times, the software product itself. The documentation should remain lean and limited to the few documents that bring real value to the project from a management or technical point of view. RUP suggests to edit management artifacts (e.g., business case and status assessment), technical artifacts (e.g., user's manual and software architecture), and requirement artifacts (e.g., the project's vision).

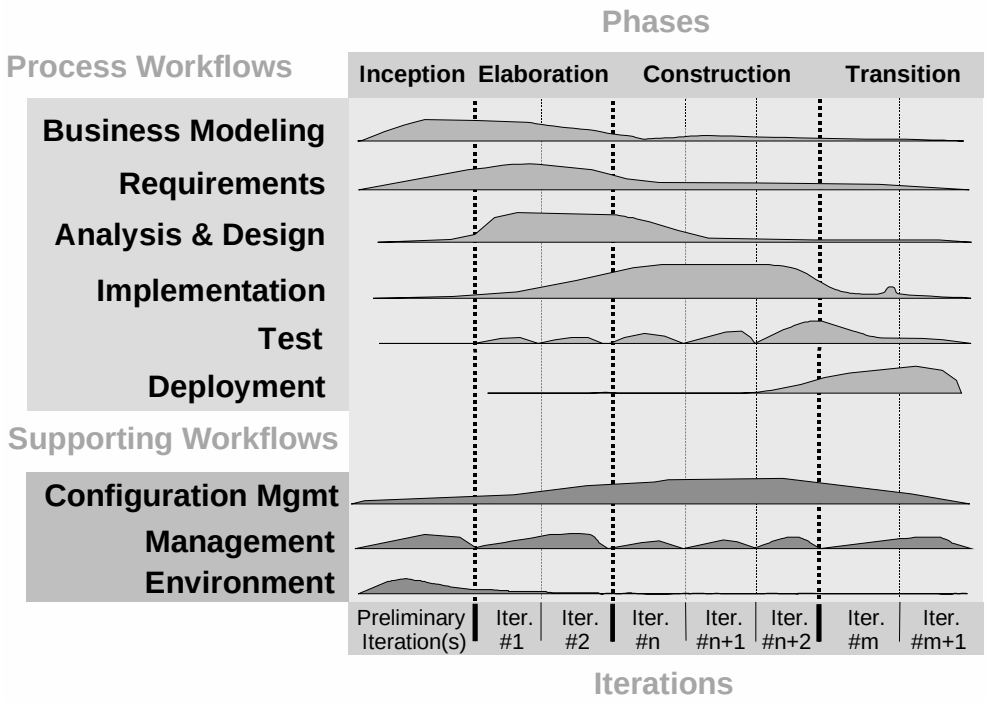

Figure 3.4.: An overview of the Rational Unified Process.

Advantages and Limitations of Systematic Processes

Systematic software processes such as Fusion and RUP are often considered to be *heavy-weight processes* (cf. e.g., [39]), insofar that they stress the importance of planning, rely on well-structured processes, and suggest to maintain a considerable amount of modeling artifacts with traceable cross-references. Industrial experience suggests [119], that medium or large projects that do not pay attention to establishing effective processes early are forced to slap them together late, when it takes more time. Heavy processes are also convenient for project management, because companies expect explicit requirements to be produced or at least a fixed price and time [155]. A well-defined process which regularly produces comprehensive documentation is controllable and allows to measure success. Furthermore, documentation can be easily communicated and reproduced, especially in spatially distributed, large teams. High-level design models can help to introduce new developers into the team, because they are easier to grasp than the source code.

Although heavy-weight, phase-oriented processes have these benefits, they also suffer from inherent problems. Especially, it is impossible to completely define a complex system, and the user is relatively excluded from the phases [144, page 5]. This is very critical for – typically very complex – knowledge-based systems. Partsch [144, page 10] notes

> "The more complex a task, the more difficult to describe it precisely and completely, and thus to get clear objectives. The less clear the objectives, the more

> difficult the communication between the different people involved in the project (e.g., customers, end users, analyzers, designers) [...]"

On a philosophical level, the comparison of software development with conventional technical engineering disciplines is misleading. Oestereich [138, page 19] states

> "Since software is more interspersed with human abstractions than other technical systems, its complexity, perfidies, and peculiarities rather resemble the structures of human organization than typically technical ones."

Another important problem is that writing and synchronizing the various models and documents is a considerable overhead. The cost of change rises with each artifact that has to be maintained. "Traditional methodologies for software delivery are not geared to the rates of change and speed necessary today" [84]. Furthermore, software developers are often not keen on producing and using documentation, so that documents are often poorly written [39]. Finally, the amount of modeling artifacts and things to consider for planning make methodologies such as RUP very complex. The complexity of RUP is manifested by the fact that the text books on RUP [109, 89] just scratch the surface: Companies that intend to introduce RUP usually must buy an additional CD-ROM, hire a consultant, and introduce various development tools as well. A reason for this complexity is that RUP tries to cover all eventualities of any type of project. Although RUP can be adapted and custom-tailored, team leaders still need to know all of its elements well to make a wise decision what to choose and what to leave out.

3.2.2. Agile Processes

In the late 1990's, a group of practitioners and academics such as Kent Beck, Alistair Cockburn, Martin Fowler, and James Highsmith began to experiment with so-called *lightweight* or *agile* software processes that would not suffer from the negative side-effects and complexity of systematic, heavy-weight Software Engineering approaches.

Reducing the Cost of Change

In his influential 1999 book, Kent Beck [8] seriously questions the traditional belief that the cost of changing a program necessarily rises exponentially over time (as shown before in figure 3.2). He asks the following:

> "The software development community has spent enormous resources in recent decades trying to reduce the cost of change – better languages, better database technology, better programming techniques, better environments and tools, new notations. What would we do if all that investment paid off?" [8, page 23]

The assumption of a considerably flattened – or even asymptotic – cost curve is *the* fundamental premise of agile processes. If projects succeeded in reducing the cost of change,

they could take an entirely different approach, since much of the overhead produced by traditional engineering methods would no longer be required (cf. [1]): Instead of writing a comprehensive documentation, they could focus on producing working software. Instead of relying on strong processes and tools, they could support individuals and interactions. Instead of following a plan, they could respond to change dynamically. Instead of fixing the system requirements in contract negotiations, they could get the customer on board, and allow her steer the development process according to her changing knowledge.

Therefore, instead of putting most of their resources into comprehensive analysis and design activities, which shall help detecting potential errors and problems before they are implemented, agile processes focus their efforts on producing and maintaining *changeable* software. However, the somehow revolutionary – but essential – assumption of a flattened cost curve is doubted by many researchers (e.g., [47]). I will show later which specific techniques are employed by the existing agile methodologies – and my XP.K approach – for keeping the cost of change low, and point to case studies which provide evidence that they work in practice.

Among the most prominent light-weight approaches are *Adaptive Software Development* [83], the *Crystal Family of Methods* [37], and *Extreme Programming* [8], which is described in the following.

Extreme Programming (XP)

Extreme Programming (XP) [8] was conceived in 1996 by Kent Beck to address the specific needs of software development conducted by small teams in the face of vague and changing requirements. In a nutshell, the main idea of XP is to iterate a process in which developers generate tests together with the customer, code what is necessary to run those tests, and integrate the product. XP is based on four *values*:

- *Communication.* Many projects fail because a programmer does not tell someone else about a critical change or because a knowledge engineer does not ask a domain expert the right question. Fluent communication supports brain-storming, creativity, and a motivating team spirit.

- *Simplicity.* XP argues that it is better to do a simple thing today and pay a little more tomorrow to change it if demand arises. Beck suggests to ask "What is the simplest thing that could possibly work?" [8, page 30]. Simple designs are easier to implement, to change, to communicate, and to understand.

- *Feedback.* Frequent feedback allows to evaluate designs and makes sure that changes in one part of the system did not damage other parts. Prototypes support communication, especially with customers or domain experts, and allow the management to get a picture of the project's schedule.

- *Courage.* XP's design strategy resembles a hill-climbing algorithm, in which you start with a simple design, then make it a little more complex, then a little simpler, and

so on. However, you might reach a local optimum, from which only complex changes will improve the situation. Applying such complex changes requires courage – also the courage to throw away one's beloved code.

These four values are the long-term goals which development teams should pursue at all times. Based on these values, XP derives several basic *principles*, which serve as rather short-term guidelines. The most important of these principles are

- *Rapid Feedback.* Get feedback, interpret it, and put what is learned back into the system quickly.
- *Assume Simplicity.* Treat every problem as if it can be solved with ridiculous simplicity.
- *Incremental Change.* Don't risk the overall system by making big changes all at once.
- *Embracing Change.* Solve the most pressing problem first, but preserve as many options as possible.
- *Quality Work.* A team doing a good job will most probably enjoy its work.

Beck mentions some more – less central – principles, such as *Open, Honest Communication*, *Teach Learning*, *Small Initial Investment*, and *Travel Light*. Traveling Light means to maintain only few, simple, and valuable artifacts instead of a comprehensive monolithic design. In XP the source code is the main artifact, and few – if any – other design models should be maintained.

In addition to these rather high-level values and principles, XP is based on the following *practices*, which guide the daily work.

- *The Planning Game.* The scope of the next release is determined by combining business priorities and technical estimates. Beck suggests to use simple "story cards" which can be stacked according to their priority.
- *Small Releases.* Simple versions are put into practice quickly.
- *Metaphor.* All developers are guided by a simple shared story and vocabulary of how the system works. The role of the metaphor is comparable to the *architecture* in object-oriented design ("the big boxes and connections" [8]).
- *Simple Design.* The system is designed as simple as possible, so that there are only few models to update on changes. Instead of designing a system that already covers all possible future extensions ("You ain't gonna need it" [8]), a system should only cover what is needed "today", with the option of generalizing it later.

- *Testing.* The users or customers define and test functional criteria. Programmers continually write unit tests to simplify integration and ensure quality. Automated test tools (e.g., JUnit [66]) support them. The unit tests also specify and document the intended functionality of a program. Beck advises to write the tests prior to the implementation, so that programmers are forced to define a rudimentary specification beforehand.

- *Continuous Integration.* Code is integrated and tested after a few hours.

- *Refactoring* [63]. Programmers restructure the system without changing its behavior to remove duplication, to improve communication, or to add flexibility. Typical refactorings are splitting long methods and large classes. Refactoring enables the incremental improvement of simple designs in support of maintainability.

- *Pair Programming.* All code is written by two programmers at one machine. While one programmer implements the current task, the other is thinking more strategically. Pair programming is a dialog at many levels, assisted by and focused on a computer.

- *Collective Ownership.* Anyone who sees an opportunity to add value to any portion of the code can (and should) do so at any time.

- *Coding Standards.* Collective ownership requires well-defined conventions for code layout, naming of variables, amount of documentation, etc.

- *40-Hour-Week.* Developers should not work overtime, so that they remain creative and motivated.

- *On-Site Customer.* A real customer or user is included in the team, available to answer questions and evaluate alternatives.

None of these practices is exceptionally new and some practices might remind conservative software engineers to plain hacking that has long proven to be obsolete. However, as illustrated in figure 3.5, XP assumes that these practices support each other, so that the weakness of one practice is compensated by the strengths of another. Especially, several of the practices, such as Refactoring, Testing, Coding Standards, and Collective Ownership, aim at reducing the cost of change and thus allow to start with a simple design. Thus, the designers of XP claim that an efficient development methodology *emerges* from the combination of various simple practices.

Evaluating Agile Processes

Due to their (apparent) ability to cope with change, their support for communication, and their focus on feedback, agile processes seem to meet the requirements of knowledge-based systems better than systematic, heavy-weight processes. In this document, I will therefore propose the XP.K methodology, which can be regarded as an agile methodology related

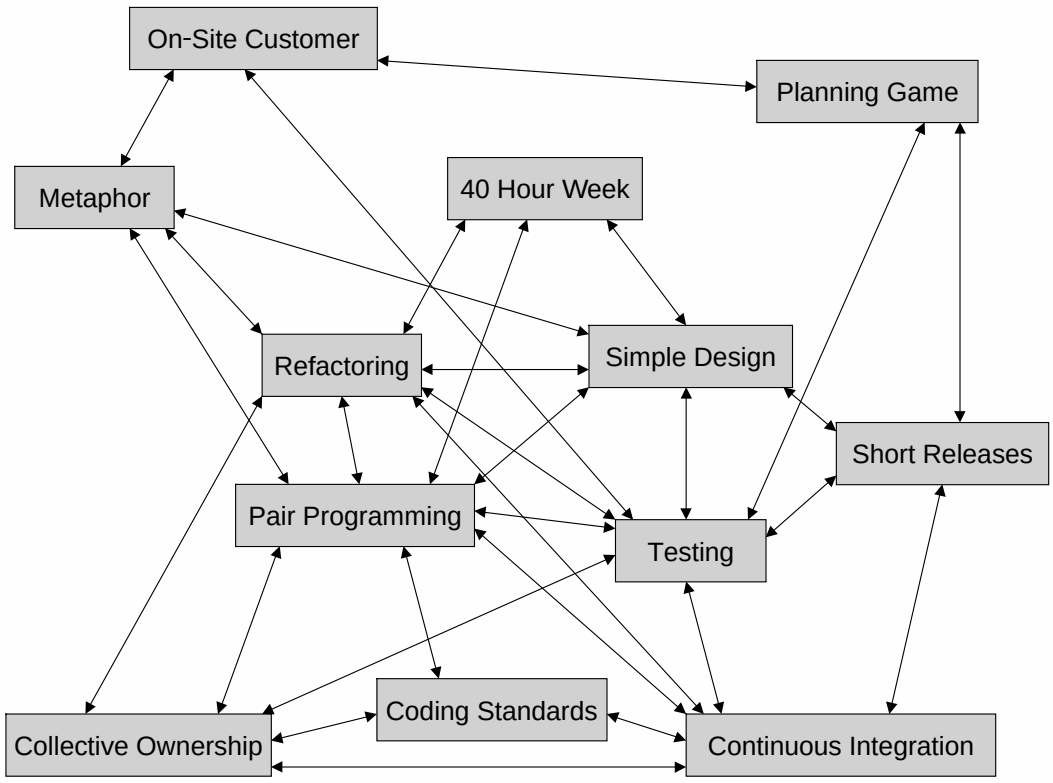

Figure 3.5.: The practices of Extreme Programming support each other (figure adapted from [8]).

to Extreme Programming. Of course, prior to being able to proof the benefits of XP.K, I will need to discuss in how far the values, principles, and practices of XP really work as their inventors claim. Due to the limited space, I could only briefly sketch XP. Many aspects such as the benefits of pair programming and the lack of a well-written design model might appear critical and demand further discussion (see [47] for a critics of XP). However, in order to reduce duplication, I will discuss and evaluate details of these issues in the context of XP.K in later chapters. The arguments for and against agile approaches will be summarized in chapter 9.

Until then I will assume that XP is a useful technology for a considerably large class of projects. This is stressed by positive feedback from industry [70, 25, 145, 133, 163], where small teams conducting XP style development even outperformed much larger teams in the same task. However, due to the newness of agile methodologies and the general problems of comparing as complex entities as development methodologies reliable data are still scarce. Independent large-scale evaluations and comparisons against traditional Software Engineering approaches are not available yet [156].

3.3. Languages

Flexible development methodologies for knowledge-based systems should allow the developers to choose any modeling language they prefer. However, the efficiency of a methodology strongly depends on its interplay with specific modeling and programming languages, because a methodology with concrete guidelines and efficient tools is easier to put into practice. Although for that purpose it would be possible to use or define a proprietary, object-oriented language that is optimized for the domain of knowledge-based systems, commercially applied standard languages should be preferred. First, relying on standard languages makes it easier to recruit teams of developers. Second, widely-used languages will have a better tool support. Third, the evolution and advancement of successful languages is secured. Forth, there will be a greater pool of reusable components and libraries.

For these reasons, this section will take a glimpse at the current status quo in object-oriented modeling and programming languages. I will first give a very brief overview of the *Unified Modeling Language* (UML), which is *the* standard language for object-oriented analysis and design. This overview will also clarify the syntax and semantics of the various class diagrams which appear throughout this document. Then I will outline the meta-modeling language *Meta Object Facility* (MOF) and its application in the definition of UML and similar languages. Finally, I will list the currently most important object-oriented programming languages, as of 2001.

3.3.1. The Unified Modeling Language (UML)

In the 1990's, Object-Orientation was characterized by a confusing number of different methodologies. These methodologies did not only strongly diverge in their modeling processes, but also came with individual modeling languages and notations. Since the differences between these languages were only minor, a group of leading methodologists decided to join forces under the umbrella of the *Object Management Group* (OMG) with the goal of defining a standard language for object-oriented analysis and design. The resulting *Unified Modeling Language* (UML) [21, 64], which is continuously being revised and extended, is now the de-facto standard language for object-oriented modeling artifacts [122].

UML provides a graphical notation for various types of diagrams. Each of these types can be used to represent a different view point on the system being developed.

Being the most widely-used type of diagrams, *Class Diagrams* describe the types of objects in the system and the various static relationships that exist among them. The syntax of Class Diagrams is illustrated in figure 3.6. Class Diagrams are very popular due to their intuitiveness, simplicity, and their similarity to the well-known Entity-Relationship Diagrams [142]. Depending on the perspective and the phase of the modeling process, Class Diagrams can be used in varying granularity (cf. [64]). First, they can be used informally to identify the concepts in the domain and their relationships. This simplifies the communication between domain experts and engineers. Second, Class Diagrams can be used to visualize and design the implementation details of the program's classes.

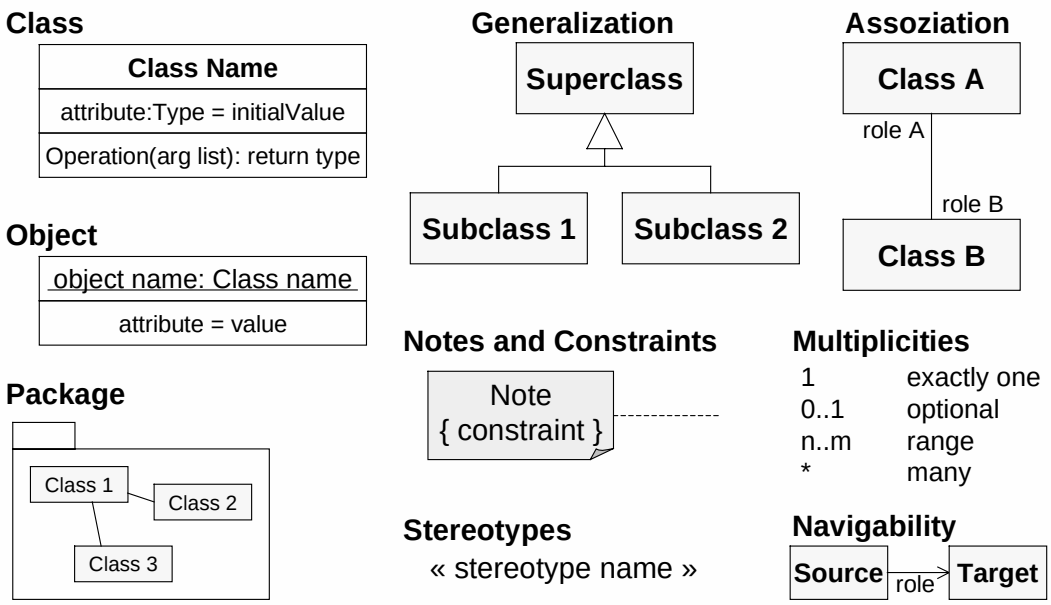

Figure 3.6.: An overview of the notation of UML Class Diagrams.

Another important UML diagram type is *Use Case Diagrams*. A Use Case is a collection of interactions between users (the *actors*) and the system. Use Cases are a popular means of capturing and illustrating requirements. Besides Class and Use Case Diagrams, UML provides additional – less frequently used – types of diagrams to capture the dynamic behavior of the system. *Activity Diagrams* are comparable to workflow diagrams, and describe the sequencing of activities, with support for both conditional and parallel behavior. *State Diagrams* describe all possible states that a particular object can get into and how the object's state changes on incoming events. *Interaction Diagrams* describe how multiple objects collaborate in some behavior. UML supports two types of Interaction Diagrams: *Sequence Diagrams* (also known as "swim-lane" diagrams) and *Collaboration Diagrams*.

UML also provides a formal specification language, called the *Object Constraint Language* (OCL) [194], which allows to rigorously define an operation's purpose and the legal state of objects. OCL can be used to communicate designs and to validate the executing system. However, this language is only slowly gaining acceptance [190].

Since UML is the widely-agreed standard language for the foreseeable future, none of the current methodologies with support for analysis and design activities should neglect supporting UML. Knowledge modeling approaches with a high degree of compatibility to UML can profit from massive industrial tool support and the other advantages mentioned above. Even the syntax of the leading Knowledge Engineering methodology CommonKADS (see chapter 4) has converged towards UML. This is not surprising, since UML is quite easy to learn, and its visual diagrams (especially conceptual Class Diagrams, Use Cases, and Activity Diagrams [64, page 11]) are adapted for the communication with domain experts.

However, the (increasing) size and complexity UML is often criticized [122, 88]. Only some of its diagram types and only about 20 % of the syntax of Class Diagrams are frequently used in the real world [64, 138]. Team leaders are therefore frequently advised to choose only a small subset of UML for their projects.

3.3.2. Metamodels and the Meta Object Facility (MOF)

The essentials of visual languages such as UML can be easily communicated *informally* between humans. However, in order to allow computers to automatically process and exchange UML models, a stronger specification of the syntax and semantics of UML is required. Furthermore, distributed development teams which cannot profit from face-to-face communication to resolve ambiguities require exchangeable models with well-defined semantics. As illustrated in figure 3.7, the role of the *Meta Object Facility* (MOF) [137] is to provide a rigorous metamodel for modeling languages, such as UML – and subsets or extensions thereof.

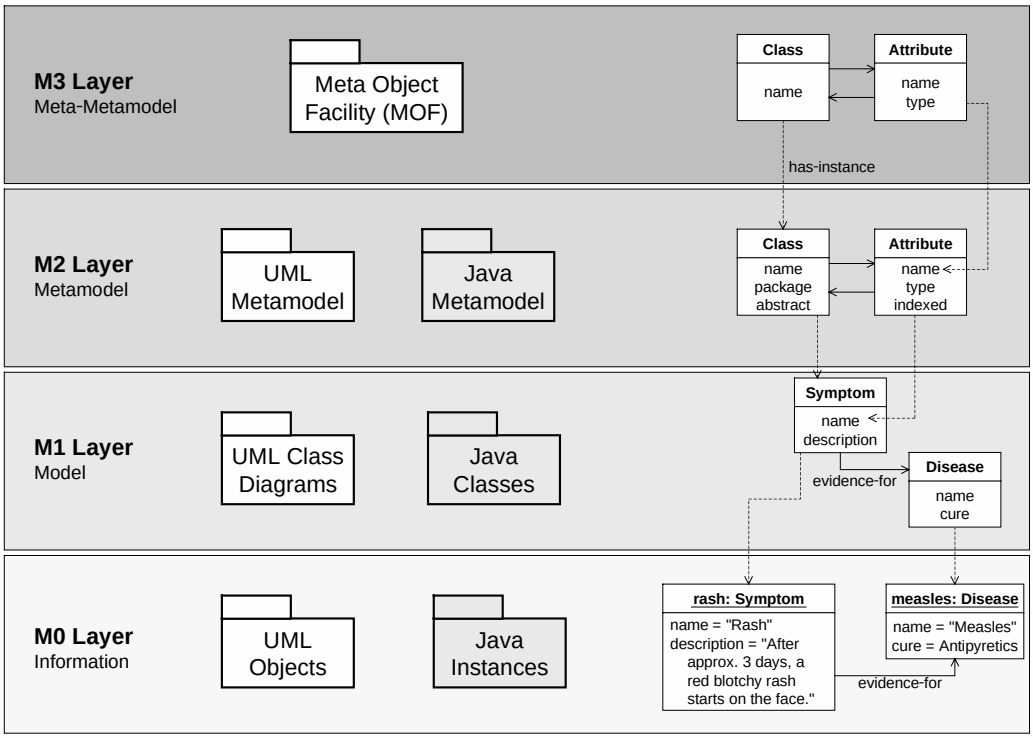

Figure 3.7.: A four-layered metadata architecture of object-oriented systems.

MOF is defined in itself, so that no additional languages above the meta-meta level are required. MOF can be represented in UML Class Diagrams (an excerpt of this diagram is shown in Appendix F). Apart from the UML notation, MOF models can also be represented textually in a language called *XML Metadata Interchange* (XMI) [140].

3.3.3. Object-Oriented Programming Languages

As stated in chapter 2, most useful knowledge-based systems will comprise of modules written in off-the-shelf software development languages. These are general-purpose languages, which – unlike descriptive or functional languages such as Prolog and LISP, respectively – do not provide explicit support for knowledge modeling or "intelligent" reasoning. The choice of an implementation language for the system components that are not directly involved in knowledge processing depends on factors such as legacy system integration, compatibility, personal preferences of the developers, and the availability of tools and libraries for building user interfaces. Since it is therefore unwise to tie one's methodology to one specific language, I will briefly list those languages that a methodology will most probably have to cope with in the real world.

Languages such as *Smalltalk* [114], *Beta* [117] and *Eiffel* [125] feature some of the cleanest object-oriented concepts and would provide an interesting platform for knowledge-based systems. Eiffel even supports semantically rich data structures. However, these languages are not widely used in industry. On the other extreme, the currently most widely used languages for desktop applications on Windows – *Visual Basic* and *C++* – can barely be regarded as object-oriented languages, because they suffer from the heritage of their imperative predecessors. Recent surveys (e.g., [50, 12]) indicate a significant migration of developers from these languages to *Java* [6] and Microsoft's Java pendant *C#* [126]. With already more than half of the developers using Java regularly, Java is expected to overtake C and C++ in the year 2002 [65]. The positive acceptance and extensive use of Java in universities (including the dominance of Java on leading conferences such as OOPSLA) and the increasing importance of the internet might accelerate this trend.

3.4. Tools

In order to keep this document self-contained, I will now give a short impression of the main features of modern software development tools. This will encompass Integrated Development Environments, which basically add some comfort to plain hacking, and modern CASE tools with support for Round-Trip Engineering between UML diagrams and source code. Both types of tools will be employed by XP.K.

3.4.1. Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs), such as Forte for Java [180] (figure 3.8, left), typically provide features for project management, hierarchical class browsing, source code editing, compiling, and debugging. Most of these tools also support visual programming of graphical user interfaces, including source code generation. IDEs are focused on the code but relieve the programmer of burdensome standard tasks.

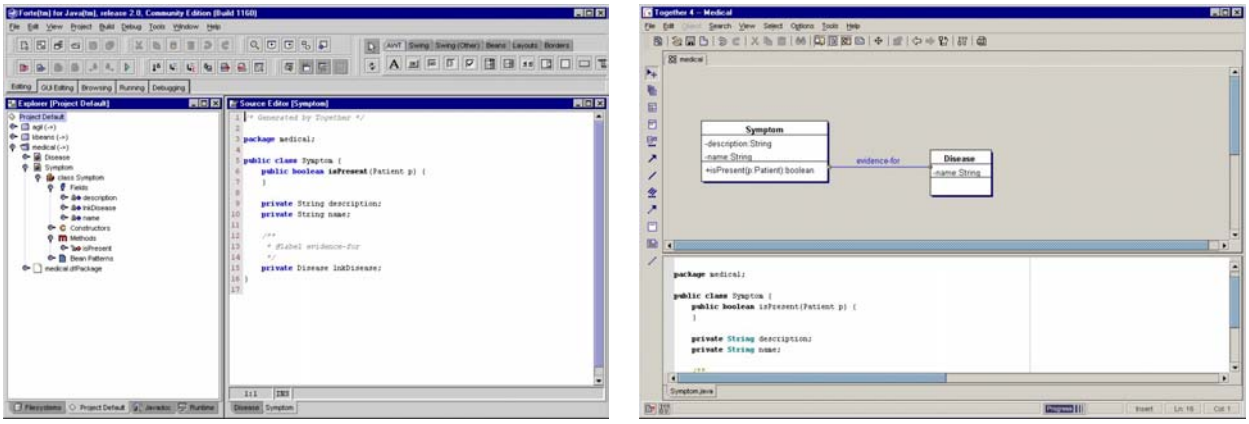

Figure 3.8.: Screenshots of an Integrated Development Environment (Forte for Java [180], left) and a CASE tool with Round-Trip Engineering between UML Class Diagrams and Java code (Together for Java [184], right).

3.4.2. CASE Tools and Round-Trip Engineering

Whereas IDE's focus on the implementation phase, CASE (Computer-Aided Software Engineering) tools support analysis and design activities. As shown in figure 3.8 (right), modern CASE tools are comfortable drawing tools for UML diagrams, especially Class Diagrams. The resulting models can be exported and distributed in the XMI format.

The major benefit of CASE tools is their coupling of high-level UML diagrams with the respective source code. This coupling is bidirectional: CASE tools can generate the method templates from Class Diagrams (*Forward Engineering*), but also extract Class Diagrams from source code (*Reverse Engineering*), even if the source code was modified externally. This bidirectional coupling is often called *Round-Trip Engineering*. It enables developers to arbitrarily switch between programming and designing. This considerably reduces the need to synchronize the models manually and is therefore a means of reducing the cost of change.

Round-Trip Engineering is made possible because UML and object-oriented programming languages have a shared set of concepts. Although UML has some shortcomings with Round-Trip Engineering [45], especially with dynamic aspects such as method bodies, it is generally straight-forward to find a translation from UML to the programming language. The other way round is harder, because UML concepts, such as relationships, can be implemented in many different ways. Therefore, the generated UML models often must be prettified, or even corrected, manually.

3.5. Summary

This chapter has provided an overview of the paradigms, processes, languages, and tools of current object-oriented methodologies. There are basically two predominant approaches to the development process, each of which has its individual strengths and application scenarios.

Systematic engineering approaches are preferable for projects with relatively fixed requirements or large teams. They aim at guiding the team iteratively from high-level analysis models to a traceable implementation. Powerful CASE tools and the intuitive visual language UML are used to design and communicate models. The main limitation of systematic methods is their inability to cope with frequent changes in requirements or knowledge models.

Agile approaches are increasingly used by projects with weakly-defined requirements and small to medium-sized teams. They try to reduce the cost of change by consequently keeping plans, design and modeling artifacts small. Their explicit support for evolutionary development and communication with domain experts suggest that agile approaches are attractive for the domain of knowledge-based systems.

Both approaches make use of a major strength of the object-oriented paradigm: Object-Orientation is based on only very few concepts, which pervade through all models and phases of development. Object-Orientation is therefore relatively easy to learn, to communicate, and enables smooth transitions between the models [144].

The main weakness of object-oriented methodologies in the context of knowledge-based systems is their missing (explicit) support for knowledge representation and intelligent reasoning, because their processes, languages, and tools are general-purpose. However, in later chapters on XP.K, I will show that the general-purpose methodologies can be custom-tailored to knowledge-based systems. Before that, I will report on the state-of-the-art in Knowledge Engineering – a discipline with an Artificial Intelligence background, which has absorbed many concepts from Object-Oriented Software Engineering.

4. Knowledge Engineering

Knowledge Engineering (see [175] or [176] for recent surveys) is often regarded as a mixture of Artifical Intelligence and Software Engineering. Knowledge Engineering aims at defining methodologies that allow to construct knowledge-based systems in a systematic and controllable manner [175].

In this chapter, I will review state-of-the-art methodologies from Knowledge Engineering by their paradigms (section 4.1), process models (section 4.2), knowledge modeling languages (section 4.3), and knowledge acquisition tools (section 4.4). I will conclude this chapter with a summary (section 4.5).

4.1. Paradigms

Early research in Artificial Intelligence was focused on the development of "expert systems" that tried to capture domain knowledge by means of primitives such as production rules which were evaluated by rule interpreters. Despite early success with small prototypical systems, research in expert systems failed to scale up for large systems, because the rather simple formalism of production rules did not support an adequate representation of *different types* of knowledge. "Such a mixture of knowledge types, together with the lack of adequate justifications of the different rules makes the maintenance of such knowledge bases very difficult and time consuming" [176]. Furthermore, knowledge encoded in rules is hard to reuse.

In order to overcome these limitations of expert systems, researchers from Knowledge Engineering suggest to build knowledge-based systems from two kinds of reusable components: Ontologies and Problem-Solving Methods. Ontologies describe domain knowledge in a generic way and provide an agreed-upon understanding of a domain. Problem-Solving Methods describe the reasoning processes of a knowledge-based system in an implementation- and domain-independent manner. In the following, I will introduce the basic concepts behind these types of components and show their respective roles in a general architecture of knowledge-based systems. In the later sections, I will show which modeling processes, languages, and tools can be used to specify them.

4.1.1. Ontologies

The essential promise of ontologies is condensed in the following definition, which is based on an earlier definition by Gruber [79] and is now widely accepted by the Knowledge Engineering community (cf. [187, 176, 52]).

> **Definition (Ontology)**: An *ontology* is a formal, explicit specification of a shared conceptualization.

Taken apart, this means the following. Ontologies are *formal* so that they can be automatically processed by machines. Ontologies are *explicit specifications*, i.e. they define concepts, properties, and relations explicitly. Ontologies are *shared* in so far that they are agreed-upon by various groups of people and that they can be communicated between machines (e.g., in a multi-agent system). Finally, ontologies are *conceptualizations*, i.e. they are abstract models of some phenomenon in the world.

Ontologies have become a popular research topic in Artificial Intelligence, natural-language processing, knowledge management, and – rather recently – intelligent information retrieval on the internet. Examples of ontologies include the *Yahoo!* internet directory, e-commerce standards for data exchange, and the medical structured vocabularies of the SNOMED standard [150].

In the context of knowledge-based systems, ontologies provide a vocabulary of terms and relations with which a domain can be modeled. A major goal of building ontologies is to enable *reuse of knowledge models*. In order to do this efficiently, an ontology should satisfy the following design principles (cf. [187, 189]): An ontology should have a high internal coherence and a low coupling with other ontologies. Ontologies should be modular, extensible, center their definitions around natural categories, have a minimal encoding bias, and a minimal ontological commitment.

4.1.2. Problem-Solving Methods

Whereas the problem-solving behavior of traditional expert systems was encapsulated in generic inference engines like rule interpreters, modern Knowledge Engineering approaches agree that reasoning knowledge itself should be made explicit in Problem-Solving Methods [29].

> **Definition (Problem-Solving Method)**: A *Problem-Solving Method* is an explicit specification of inference steps, a control structure over these steps, and the role that domain knowledge plays in each step.

Figure 4.1 illustrates the inference steps and their control structure of a sample Problem-Solving Method for diagnosis tasks. Each of the inference steps (represented by ovals) receives dynamic knowledge or data from the problem domain as input (e.g., specific patient data) and produces new knowledge as output (e.g., a hypothesis about a disease). Apart from this dynamic knowledge, the steps are further characterized by static knowledge

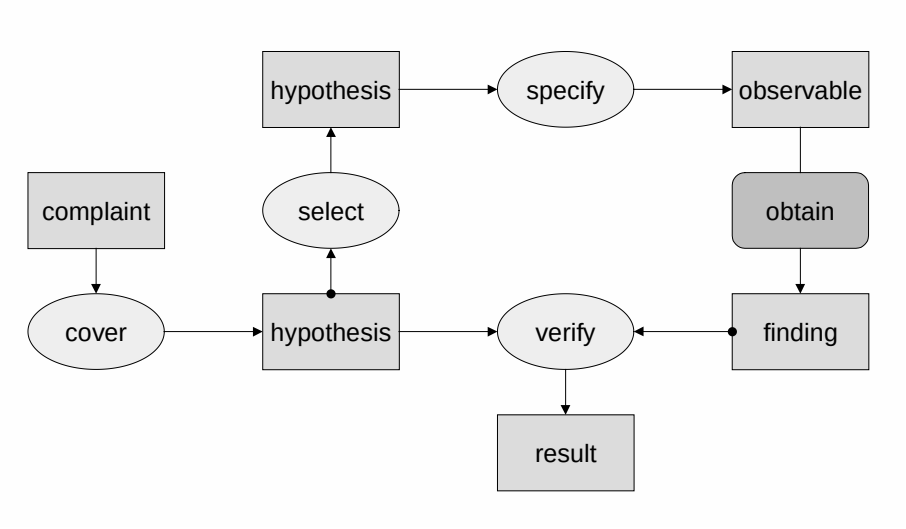

Figure 4.1.: A Problem-Solving Method for diagnosis tasks (adapted from [164]).

about the domain (e.g., the symptoms of diseases). Both roles of knowledge are specified by means of generic *method or task ontologies* [128].

In the terminology of Object-Orientation, Problem-Solving Methods can be regarded as Design Patterns that describe generic components which interact in a controlled order to solve a task. Each of these components (or inference steps) is defined through its interface, which is a class structure that specifies the syntax and semantics of its input and output.

By explicitly specifying problem-solving knowledge, Knowledge Engineering methodologies aim at pursuing three goals [175, 29]. First, the structure of Problem-Solvers can serve as a guideline for the acquisition of static domain knowledge. Second, the reasoning algorithms are made transparent so that experts can better understand the problem-solving process. Third, reasoning knowledge can be reused because Problem-Solvers are components with well-defined interfaces, functionality, and assumptions [57]. The enabling technology behind such reuse is *ontology mapping* [174]. A mapping defines a translation or *bridge* [67] from specific domain ontologies to the generic method ontologies.

4.1.3. An Architecture for Knowledge-Based Systems

As illustrated in figure 4.2, ontologies and Problem-Solving Methods are the central components in a generic architecture for knowledge-based systems (cf. [56, 78]).

Domain ontologies specify the terminology (or the concepts and their relationships) of the (static) knowledge base. The ontology is mapped to the Problem-Solvers' ontologies, so that the domain knowledge can be processed by the reasoning algorithms. The Problem-Solvers access and modify the dynamic case data, which are also modified by the remaining system components and presented to the end users. The user interface may also access the

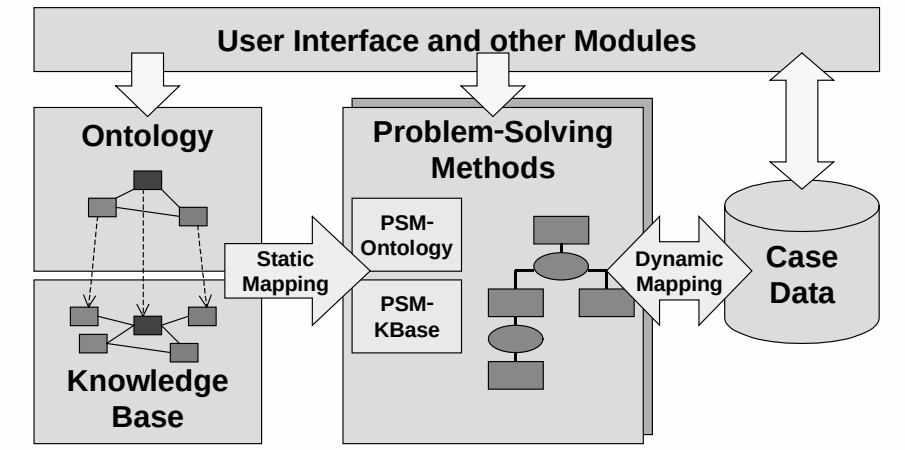

Figure 4.2.: A generic architecture for knowledge-based systems.

domain knowledge model, for example to display details of a specific disease and to enable the user to trace and control the problem-solving mechanisms.

Although this architecture is widely accepted on a theoretical level, it is based on assumptions that do not always reflect real-world requirements. In particular, the premise that ontologies can be designed independently from their potential use in inference algorithms has been criticized by Bylander and Chandrasekaran [24]. They state that prior to building ontologies, a clear understanding of the tasks and inferences that the ontology will be applied to is required. This so called *interaction problem* has remained an inconvenient issue in modern Knowledge Engineering approaches. To make things worse, its potential solution – the use of suitable mapping techniques and guidelines for their application – is recognized as a non-trivial issue and is still a matter of spirited discussion. Even the most prominent Knowledge Engineering approach CommonKADS deals with the mapping problem only very vaguely and notes "an elaborate description of ontology mappings is the subject of further research" [164, page 417]. Blaha and Premerlani [14] state that "the topic of transformations is heavy reading".

4.2. Process

The dominant view in the Knowledge Engineering literature is that the development process of ontologies, Problem-Solving Methods, and their relationships should not be performed in an ad-hoc fashion. "The absence of commonly agreed on guidelines and methods hinders the development of shared and consensual ontologies within and between teams" [74]. If systems are built on a large scale and with some guarantees of correctness and completeness, systematic methodologies should be followed. Furthermore, earlier attempts to simply "translate" domain knowledge from human experts onto formal knowledge representations are regarded as insufficient, in particular due to the important role of

tacit knowledge for an expert's problem-solving capabilities [175]. Therefore, Knowledge Engineering is regarded rather as a *modeling process* than a transfer process of domain knowledge [129]. Since this modeling process depends on subjective interpretations and is therefore typically faulty, it must run through several iterations and involve evaluation steps.

Several research groups from the Knowledge Engineering community have proposed development methodologies for knowledge-based systems (see [115] and [175] for overviews). These groups use different modeling languages and tools, particularly their respective individual research platforms [115], and few of them have been evaluated and widely used by independent groups [10]. "Knowledge Engineering did not get a widespread attention in the past" [173].

4.2.1. Ontology Modeling Approaches

Ontologies are the central building blocks of knowledge-based systems and as such demand particular methodological support. In the following, I will produce a brief yet fairly comprehensive overview of ontology building methods. Recent surveys of these approaches can be found in [94] and [115]. The approaches were built by reflecting on practical experience in the construction of specific ontologies and are therefore relatively immature for the general purpose. As a result, ontology building is still rather a craft than an understood engineering process [94, 74]. However, there is an agreement that ontology construction should be harnessed in a step-by-step process that is supported by a succession of intermediate models and rules of thumb. The approaches also agree to start with informal models and to formalize them relatively late in the process.

Interestingly, most of the approaches take a *task* as the starting point, i.e. they suggest to perform ontology construction with the ontology's usage scenarios in mind. This makes reuse of the resulting ontologies inherently hard and can be regarded as the defeat in the face of the interaction problem [94].

Uschold

Uschold's methodology [186] begins with the identification of the purpose of the ontology and then the key concepts and relationships in the domain. These are first captured in an unambiguous textual form, then mapped onto a precise terminology, and then coded in a formal knowledge representation language and integrated with existing ontologies. The integrated ontology is finally evaluated and documented for later modification and reuse.

Grüninger and Fox

Similarly to Uschold's approach, *Grüninger and Fox's methodology* [80] starts with the capture of motivating scenarios which also clarify the semantics of the later ontologies. The authors then propose to define the *competency* of the ontology by means of informal *questions* that the later system shall be able to respond to. The methodology then lets

the developer first produce a semi-formal and then a formal specification of the ontological concepts. The resulting terminology is then used to formalize the competency questions and the axioms and constraints that declare the semantics of the ontology. These are finally used to evaluate the system by applying the competency questions.

This approach shows remarkable similarities to the "test-first" development style of Extreme Programming. The competency questions are comparable to test cases, which are in XP also envisioned prior to building the respective code and then run by calling the (later implemented) methods.

Methontology

Methontology [15] is more complex than the aforementioned approaches. Beside a waterfall-based development process model, it also identifies project management and support activities. The development itself iterates through the phases specification, knowledge acquisition, conceptualization, integration, implementation, evaluation, and documentation. Like all the other ontology modeling approaches, it suggests a middle-out strategy, in which the most important domain concepts are modeled first and then refined.

4.2.2. Knowledge Engineering Methodologies

There are very few general-purpose methodologies described in the literature which have a broader scope than the ontology building approaches above and cover the whole software life-cycle.

CommonKADS

By far the most frequently cited Knowledge Engineering methodology is CommonKADS [164]. CommonKADS is the result of various European research projects from the 1990's. The evolution of the approach has virtually ceased (e.g., as of mid-2001, the official project homepage has not been updated for more than one year), but some of its main ideas have been adopted by related research projects.

CommonKADS suggests a systematic engineering process in which different models are constructed, each of which capturing a specific aspect of the system. CommonKADS applies the *Knowledge Level* [132] principle, which states that the knowledge modeling process should first concentrate on the conceptual structures and leave the programming details for later. The models in the CommonKADS process are illustrated in figure 4.3.

These models provide a decomposition of knowledge engineering tasks, so that while building one model, the engineers can temporarily neglect certain other aspects. Although the models can be edited in parallel, the CommonKADS process is rather Waterfall-based, i.e. its models base upon each other.

The central model in the CommonKADS process is the *Knowledge Model* (sometimes called the *Model of Expertise*), which consists of three layers:

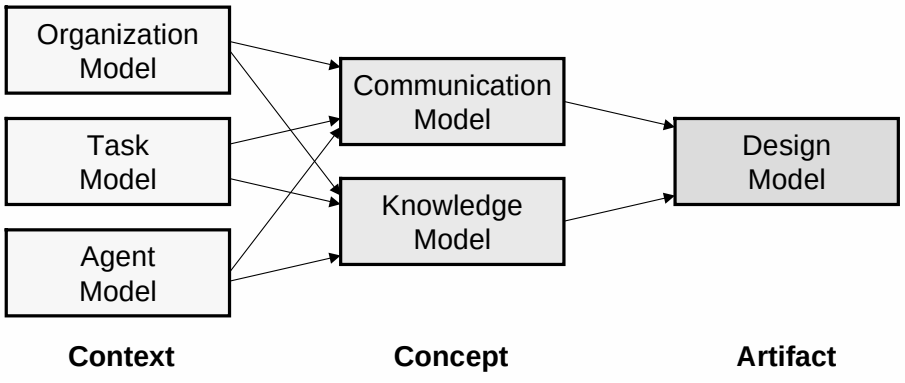

Figure 4.3.: The models in the CommonKADS methodology.

- The *domain layer* formalizes the domain knowledge (including its ontology) needed to solve the task the system shall perform.
- The *inference layer* specifies the reasoning process of the system in terms of Problem-Solving Methods.
- The *task layer* provides a decomposition of tasks into subtasks and inference actions, and specifies the control flow.

The Knowledge Model is in subsequent steps elaborated into a *Design Model*, which can then be translated into an executable implementation. All the modeling activities are embedded in a cyclic, risk-driven process similar to Boehm's Spiral Model [19]. Note that the implementation is not explicitly supported by CommonKADS, but left to the individual developers instead. The CommonKADS textbook [164] only claims that the implementation will be "usually relatively straight-forward" (page 295) and describes two examples for mappings from a design model to two specific Artificial Intelligence languages.

A central aim of CommonKADS (and other Knowledge Engineering approaches) is to allow for reuse of knowledge models, in particular by building libraries of templates for tasks and Problem-Solvers. However, this aim has not been achieved (yet) as there are barely any such libraries available.

Whereas some earlier reports on systems built with CommonKADS exist (e.g., [26]), there is little evidence that CommonKADS is widely used in industry [27]. A major reason for this is the methodology's complexity. Using CommonKADS requires to maintain large and complex documentation artifacts, with a (sometimes) confusing terminology. The formal modeling languages proposed by CommonKADS [58] are barely used. Due to its complexity, CommonKADS requires a steep learning curve [26]. The resulting efforts are estimated to be only worthwhile, if models are reused in future projects, because for a single small project, CommonKADS adds significant overhead to the time required for the project [123]. Another major weakness of CommonKADS is that there are gaps in

the transitions between the phases and models. "A common problem concerns the transformations of the high-level conceptual model into a particular implementation platform, such as for example an object-oriented environment or rules" [10]. Concrete guidelines and software support is missing. In the face of these disadvantages, it is frequently mentioned that CommonKADS does not need to be used in its entirety to be useful.

MIKE

The *MIKE* approach [4] has been developed at the University of Karlsruhe in the mid-1990's. It covers all steps from requirements elicitation through specification to design and implementation. MIKE integrates semi-formal and formal specification techniques and prototyping into an incremental and reversible system development process. Its central modeling artifact is a knowledge model very similar to the Model of Expertise from CommonKADS. In contrast to the latter, MIKE attempts to achieve smoother transitions from informal to formal, executable models. However, it uses very specialized modeling languages and tools which are virtually not used outside of the authors' research laboratories. Therefore, since the end of this project's funding, there are no noteworthy reports about industrial applications.

Protégé

A frequently mentioned Knowledge Engineering method is *Protégé* [78] from Stanford University. The focus of this approach is the construction of knowledge acquisition tools in support of evolutionary knowledge acquisition. Protégé can be regarded as an implementation of the generic architecture of knowledge-based systems from figure 4.2, insofar that it supports the rapid development of ontologies and knowledge bases which instantiate them. The approach makes little assumptions about how the ontologies are constructed but provides an efficient modeling tool (see subsection 4.4.1) which is widely used in the Knowledge Engineering community and (apparently) also outside of research labs. Protégé also supports the idea of ontology mappings as a means of accessing and reusing Problem-Solving Methods, but provides only few guidelines for their construction [135]. Protégé can therefore hardly be called a methodology, unless a very light one.

4.3. Languages

Research in knowledge representation has produced many languages to describe and formalize ontologies, Problem-Solving Methods, and their relationships. Recent overviews of these languages can be found in [41] and [52]. The languages are still evolving, particularly in the face of new requirements arising with the advent of the "semantic" internet [170].

Early knowledge representation approaches such as KIF [69] were based on pure first-order predicate logic. These languages have great expressiveness, but suffer from being inefficient and semi-decidable, and from lacking features to express procedural knowledge.

Furthermore, first-order logic is hard to read, communicate, and comprehend. Formal models are notoriously difficult to build [71].

As a result of these weaknesses, alternative forms of knowledge representation have been proposed, in particular frame-based languages, which organize knowledge in class structures similar to those from Object-Orientation. The idea of *frames* dates back to Marvin Minsky [127], but was also influenced by early work on object-oriented languages such as SIMULA. The frame-based approach has been further developed into *F-Logic* [97] and is implemented by more than 50 ontology specification languages such as *Ontolingua* [79], *XOL* [96], *LOOM* [116], the CommonKADS modeling language [164], and the most recent incarnation of "emerging ontology standards" called *OIL* [54].

The common elements of most frame-based languages are summarized by the *Open Knowledge Base Connectivity* (OKBC) [30], which provides a generic protocol to access and exchange knowledge expressed in various frame-based languages. As such it has an extremely general knowledge model and makes minimal knowledge-representation commitments. However, OKBC covers the by far most important aspects of frames and is therefore very popular on its own and not only as an exchange language. OKBC is now a de-facto standard which has for example been adopted by the FIPA [59] consortium for the exchange of ontologies between agents and as the underlying knowledge model of the Protégé system [135]. Since OKBC is a representative of a large class of ontology specification languages, I will provide an overview of it in the following subsection. In later chapters, I will show that OKBC can – partially – be mapped to Java classes. This mapping is at the core of KBeans, the ontology language of the XP.K methodology.

4.3.1. Ontology Specification with OKBC

The building blocks of the OKBC knowledge model [30] are illustrated in figure 4.4. A *knowledge base* is a collection of frames. A *frame* is either an instance, a class, a slot, or a facet. *Classes* are concepts in the domain of discourse. *Slots* describe properties or attributes of classes and instances. Slots are first-class objects, i.e. they can exist without being attached to a specific other frame. Slots are either primitive or non-primitive. *Primitive slots* can take values of the basic data types (integers, floating point numbers, strings, and booleans). *Non-primitive slots* represent a relation between two frames. *Facets* describe properties of slots, in particular constraints on the valid values. All frames are *instances* of one or more classes. In several implementations of OKBC, e.g. Protégé, frames are instance of exactly one class only. Although classes are themselves instances, i.e. meta-classes exist, knowledge bases are usually treated to consist of two disjoint partitions, namely the classes (the ontology) and the instances [136].

Classes constitute a taxonomic hierarchy with inheritance defined as in object-oriented languages (but with multiple inheritance). Slots can also be arranged in an inheritance hierarchy. OKBC distinguishes between so-called own slots and template slots. *Own slots* can be attached to any frame and store a set of values to describe the frame. Every own slot is related to one template slot. *Template slots* are attached to classes and describe the type and default values of own slots. Whenever an instance of a class is created, its

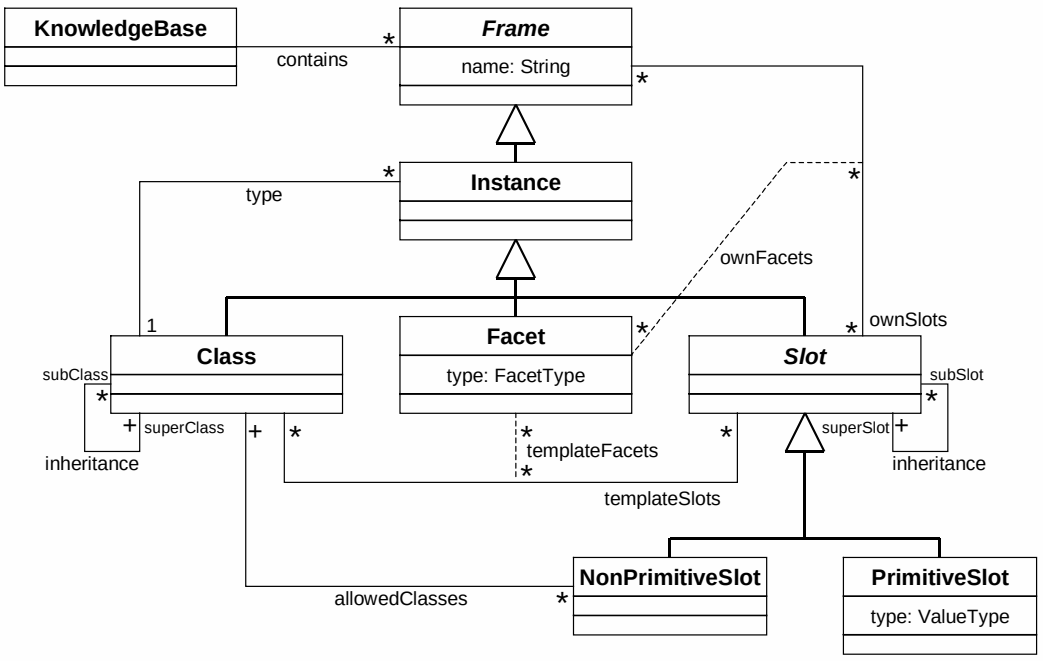

Figure 4.4.: The metamodel of OKBC (as implemented in Protégé) in UML notation.

own slots "inherit" the values of the class' template slots as default values. Thus, template slots can be regarded as *declarations* of own slots on class level.

Every frame/slot pair can be further described by a set of facets. *Facets* provide metadata about slots. They can be used to constrain the space of admissible values of a slot. Facets can only be attached to slots of certain value types. Facets are typed and OKBC defines a list of standard facet types with predefined semantics to cover the most frequently needed kinds of metadata about slots (summarized in table 4.1). For example, facets can be used to constrain the maximum number of values a slot can have (`:MAXIMUM-CARDINALITY`), to specify the numeric minimum of a primitive slot (`:NUMERIC-MINIMUM`), and to declare that a given slot must contain a subset of the values of another slot (`:SUBSET-OF-VALUES`).

Whereas frames can be used to declare structures, many ontology languages also provide support for expressing semantic relationships between frames, especially *axioms*, *rules* and *functions*. Traditional knowledge representation languages capture these semantic relationships by means of sentences in predicate logic, frequently written in the LISP-like prefix notation of KIF [79]. However, predicate logic allows for many syntactic variations to denote the same semantic meaning of an axiom and has other disadvantages as described before. Furthermore, whereas frames can be nicely visualized as conceptual graphs, textual formulae are hard to edit during knowledge acquisiton (in particular if the acquisition is performed by domain experts). Therefore, many ontology languages such as OKBC provide barely any support for axioms or rules. There exist alternatives for the specification

Table 4.1.: The constraining facets of OKBC.

Facet name	Facet type	Semantics
:VALUE-TYPE	class	Type restriction on the values of the slot.
:INVERSE	slot	Any value is also a value of the specified slot.
:CARDINALITY	int ($\geq$0)	Exact number of values.
:MAXIMUM-CARDINALITY	int ($\geq$0)	Maximum number of values.
:MINIMUM-CARDINALITY	int ($\geq$0)	Minimum number of values.
:SAME-VALUES	slot chain	Values are equal to values of specified slot.
:NOT-SAME-VALUES	slot chain	Values are not equal to values of specified slot.
:SUBSET-OF-VALUES	slot chain	Values are a subset of specified slot.
:NUMERIC-MINIMUM	number	Inclusive lower bound of value ($\geq$).
:NUMERIC-MAXIMUM	number	Inclusive upper bound of value ($\leq$).
:SOME-VALUES	<type>[]	Enumeration of valid values.
:COLLECTION-TYPE	{set, list, bag}	Values are treated as set, list, or bag.

of these elements with frames only. First, one can map the meta language of logic onto frames, so that axioms are themselves implemented by special frames such as quantifiers and operators. I will provide an example for such a frame-based axiom language in chapter 8.1. Second, it is possible to classify axioms into axiom types as described in [171] and then describe axioms as complex objects that refer to concepts and relations. This is very similar to the way of specifying semantics with facets.

The purpose of most ontology languages is to inherently support various types of reasoning, so that new knowledge can be generated from existing knowledge without having to translate the knowledge into a different format first. As illustrated in figure 4.5, frame-based languages allow for the use of *classification* (to find the class of a given instance), *constraint checking* (to check whether a given instance violates any of the constraints defined for its class), and *rule chaining* (to derive new facts from an existing knowledge base).

Note that these types of reasoning differ from the concept of Problem-Solving Methods: Problem-Solvers are on a more general level and may use the low-level reasoning methods internally. Apart from these built-in reasoning services, some languages allow to implement *procedures* which can be used to integrate ontologies much closer into the other modules of the knowledge-based system they are embedded in.

To summarize, there is a wide consensus that frames are currently the most efficient means of representing ontologies. Like objects from Software Engineering, they are relatively close to human intuition and thus easier to build and comprehend than pure logic-based languages. Frames thus meet the criteria defined in section 2.3 (adequate, easy to learn, translatable, incremental, precise, complete, and compatible) relatively well. There are noticeable similarities between object-oriented and frame-base languages. In a nutshell, ontology languages can be regarded as object-oriented languages minus methods

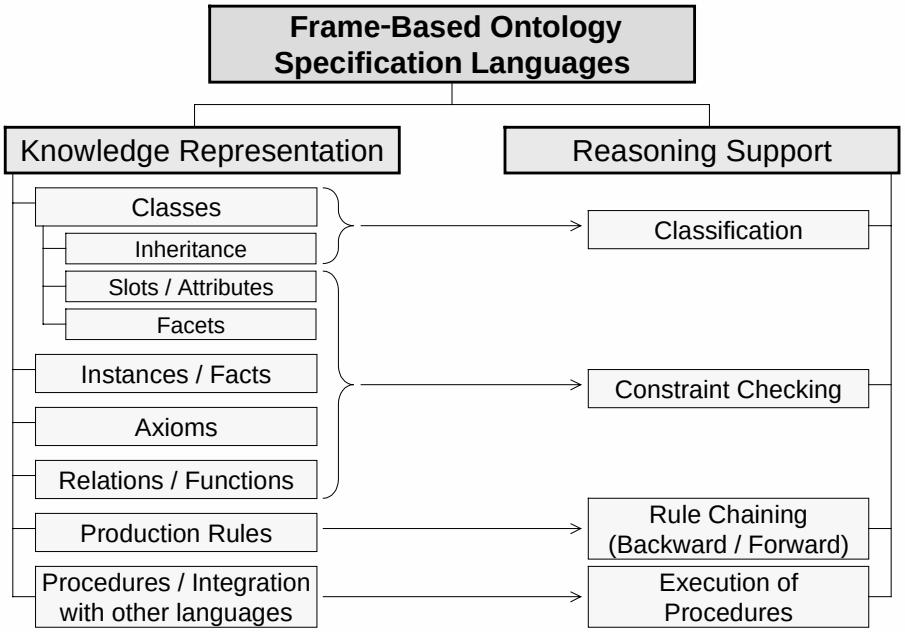

Figure 4.5.: Common knowledge representation and reasoning capabilities of frame-based ontology specification languages (figure based on [41]).

plus primitives for describing axiomatic rules or consistency conditions (cf. [5, 164, 43]). I will later discuss the differences between the two domains and argue that these differences are in many cases so minor that an object-oriented language such as Java only has to be extended slightly so that it suffices for knowledge representation.

4.3.2. Languages for Problem-Solving Methods (UPML)

Beside ontologies, Knowledge Engineering suggests to build systems from Problem-Solving Methods. For more than a decade, there have been various attempts to define method specification languages (see [188] for an overview), for example the CommonKADS knowledge specification language *CML* [164] and MIKE's language KARL [53]. However, these languages are virtually not used outside their inventor's research labs. Recently, a large consortium of internationally renowned researchers has developed the *Unified Problem-Solving Method Development Language* (UPML) [56], which can be regarded as the state-of-the-art in this area. In the following, I will provide a brief overview of this language.

UPML assumes an architecture of knowledge-based systems which consists of the three elements *tasks* (problem types), *Problem-Solving Methods*, and *domain models*. These three elements share a common *ontology*, which specifies their terminology. In support of reuse, tasks, methods, and domain models are relatively independent from each other and adapters are used to specify mappings between them. UPML supports two types of adapters, namely bridges and refiners. *Bridges* explicitly model the relationships between

two parts of the architecture, e.g. by matching input and output between the domain model and a method. *Refiners* can be used to express a stepwise adaptation of elements, e.g. a Problem-Solver is refined by means of an inheritance and overloading mechanism known from object-oriented languages.

UPML does not prescribe a specific ontology representation language but refers to existing frame-based or logic-based standards. However, it does provide semi-formal specification languages for the three architectural elements. *Tasks* are specified by their ontology, input and output roles, goal, preconditions, and assumptions. *Domain Models* are described by their ontology, properties, assumptions, and domain knowledge (rules). *Problem-Solving Methods* are defined by their ontology, input and output roles, pre-conditions, subtasks, post-conditions, and control flow. By providing these information, UPML aims at supporting the construction of catalogs of tasks and methods which solve them.

Although its language is rather abstract and not sufficiently formal to allow for a direct implementation, UPML proposes some design patterns for the implementation of UPML models in object-oriented languages. Here, every UPML Problem-Solver is mapped to one (Java) class, which communicates with other modules using specific bridge objects. Ontologies and domain models are mapped to an ordinary class hierarchy, which defines the basic terminology used in the domain models and methods. Refiners are implemented by method inheritance. This UPML-to-Java mapping is quite interesting for the KBeans approach described in chapter 6, because it admits that the gap between the Knowledge and Software Engineering world is not that large after all. I will pick up this issue later.

4.4. Tools

The forth pillar of methodologies is tool support. In the context of Knowledge Engineering, tool support is essential for the construction of ontologies and domain knowledge bases, and for their evaluation. An overview of existing knowledge acquisition tools is provided by [46]. These tools are mostly prototypes which were built to support the Knowledge Engineering approach or ontology language of the respective research group. From these, only Protégé-2000 [78] appears to be seriously used in real-world projects. Since Protégé-2000 has also received the best rating in [46] and because I will pick up several of its ideas in chapter 7, I will in the following provide an overview of this tool.

4.4.1. Protégé-2000

Protégé-2000 [78] is the latest generation of a series of knowledge acquisition tools developed at Stanford Medical Informatics. Whereas older versions were native applications for Windows and NeXT platforms, resp., the current tool is written in Java. Protégé-2000 is an OKBC-based knowledge acquisition tool. As such, it is based on a relatively strict separation between ontologies (classes) and domain knowledge (instances). Knowledge engineers who employ the tool first use the hierarchical class editor to define the ontological concepts, their slots, and their facets. Based on this ontology, the system automatically creates a

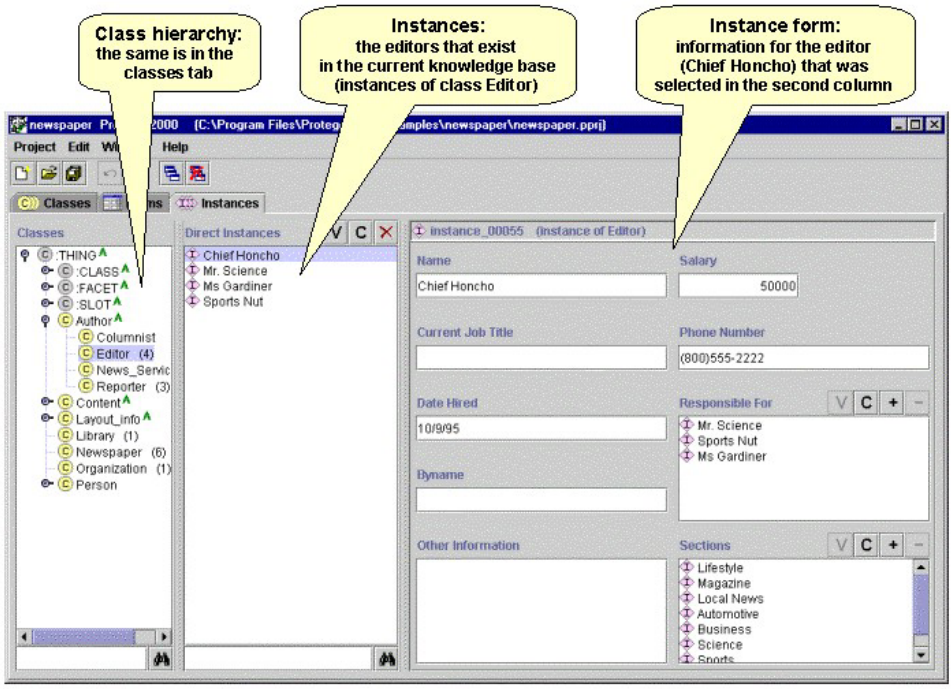

Figure 4.6.: A screenshot of the knowledge acquisition tool Protégé-2000.

knowledge acquisition interface consisting of graphical widgets that are appropriate for editing instances of the ontology classes. For example, Protégé provides standard widgets for editing integer slots and lists, which are arranged in a form or dialog when an instance is edited. Tool developers are (programmatically) able to define custom widgets for certain slot types and can visually modify the layout of the knowledge acquisition forms. When the knowledge acquisition forms are laid out (or left in the frequently sufficient default state) the instances can be edited. A screenshot of the tool is shown in figure 4.6.

Protégé-2000 has a completely modular, extensible architecture which is accessible through a relatively clean open-source API. Developers are able to customize the tool for their project by adding or modifying so-called tabs. The default tabs are class, form, and instance editors, but various other generic tabs are available as plug-ins, e.g. an OKBC interface tab and a query engine.

We have evaluated and used the system during a development project for a clinical system [108]. The following assessments are based on our own experience and reports from the Protégé bugs and wishes list (`http://protege.stanford.edu/bugs/bugs.html`). The system is particularly useful for *rapid* knowledge acquisition. The definition of ontologies, input forms, and domain instances can be performed efficiently, even by inexperienced users such as domain experts. Although the ontology classes will in most projects be designed by experienced knowledge engineers, domain experts can edit domain instances relatively

straight with only little training. Protégé meets the quality criteria defined in section 2.3 insofar that it is easy to use, flexible, adaptable, explorative, and enjoyable to use.

Some shortcomings of Protégé-2000 are due to its status as a non-commercial research prototype with limited funding (e.g., missing undo functionality, various bugs, some ugliness). A rather inscrutable limitation is that although the standard facet types of OKBC are implemented they are not used during knowledge acquisition. In particular, facet-based constraint checking could be used to detect inconsistencies, to report them to the knowledge modeler, and thus to guide user input. Another weakness of the system that we found very inconvenient during knowledge acquisition is the generic visual display of instances, in which instances are represented by generic names (e.g., "person_023") instead of rather suitable adaptive ones (e.g., "Adam"). This is particularly relevant when instances are complex, such as expression trees representing rules and axioms. Furthermore, the default knowledge acquisition widgets are seldom sufficient. Many projects have knowledge bases with graph-like structures such as process models or flowcharts, or rule-based elements. These projects will typically have to implement their own (graph and rule) editors. The very idea of allowing the knowledge engineers to modify the classes frequently and at any time hinders the development of customized widgets. In realistic projects, the rate of changing the class structure will have to be reduced.

In my opinion, the main weakness of Protégé-2000 (and all the other classical Knowledge Engineering approaches) strikes when the ontologies and domain instances are integrated with the knowledge-based system. I argue that an ontology can hardly be built in isolation, without having its functionality within the overall system in mind. Figure 4.7 shows the typical interactions of knowledge modeling components with their software environment (e.g., in our clinical decision support system ANIS [108]).

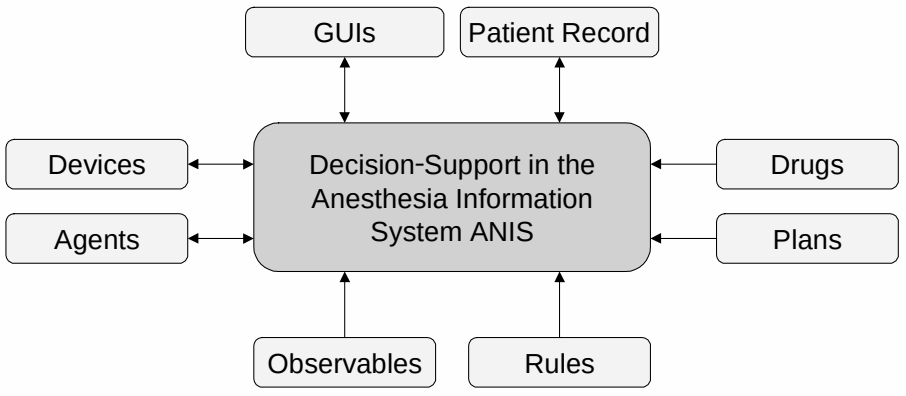

Figure 4.7.: The reasoning capabilities of knowledge-based systems are typically embedded into a software environment consisting of modules written in traditional languages.

For example, classes such as `Symptom` and `Observation` also have their counterpart in the (Java) implementation, e.g. they must be connected to the classes and user interfaces

that deliver and display the respective patient data. Complex instances which represent semantic relationships such as rules will also need to be connected to evaluating methods, especially in the context of method ontologies. Furthermore, a mix of classes that represent knowledge and those that represent case data is frequently needed (e.g., a `Patient` class). Finally, Java-based custom editors also need some hard-coded access to the ontology classes.

These issues make the transition between the knowledge model and the executable application very complicating and require major (code) duplication. The proposed solution of applying bridges or mappings between domain and method ontologies only delays the problem and writing efficiently maintainable mappings is not sufficiently understood yet [164]. As a result, mappings would need to be hard-coded in a programming language. The origin of these problems is that the Protégé knowledge model is implemented as an *Active Object Model* [61], in which classes are themselves objects. Whereas this provides a great flexibility, it also requires translations and duplications, which in turn result in considerably high costs of change. I will later tackle this problem by mapping the ontology classes directly to Java classes.

4.4.2. Other Tools

The certainly most frequently used knowledge acquisition tools are pencil and paper. In particular for early drafts and brain storming sessions, these simple and intuitive tools are sufficient. Advocates of recent light-weight software development methodologies even argue that little else is needed apart from paper and code [3]. For the later project phases, the technical successors of pencil and paper, such as Microsoft PowerPoint, Visio and other drawing tools, are frequently used. Visio even allows to enrich drawing elements with data structures (i.e., classes with attributes of primitive types). Since these data structures can (theoretically) be exported to other formats, Visio can well be regarded as a knowledge acquisition tool, in particular one that profits from massive commercial support. However, it can only extremely hardly be adapted to specific editing needs and lacks support for other editors than graphs.

4.5. Summary

To summarize this brief review of Knowledge Engineering, I will compare its methodologies to those from Object-Oriented Software Engineering. Although the text books from the areas of Software and Knowledge Engineering express a strict separation between the areas and there are few cross-references between the domains [5] there is a clear overlapping.

In contrast to Object-Orientation, the generic architecture of knowledge-based systems proposed by Knowledge Engineering separates data (ontologies and domain knowledge) from behavior (Problem-Solving Methods). It is hoped to be able to link the two aspects only through the input and output ontologies of the Problem-Solvers. However, the (widely recognized) interaction problem [24] suggests that a clear separation of data and methods

might be rarely possible.

Most Knowledge Engineering methodologies rely on very systematic, waterfall-based processes. Various guidelines for a step-by-step path from informal requirements to executable models exist and are particularly valuable for inexperienced knowledge engineers. However, processes such as CommonKADS [164] are quite heavy-weight, i.e. they have a steep learning curve and involve the maintenance of many modeling artifacts. The transitions between those artifacts are expensive [10] and neither supported by methods nor tools. Interestingly, one of the few relatively successful Knowledge Engineering approaches, Protégé, is rather light-weight, explorative, and supported by tools.

The languages promoted by Knowledge Engineering approaches are mostly frame-based and thus resemble the object-oriented view. However, they have richer means of specifying static models. Especially they have primitives for the explicit declaration of semantic constraints. At the same time, they suffer from being hard to integrate with the software architectures of the remaining systems. Therefore they require additional translations and duplications, which in turn result in high costs of change.

The most obvious difference between Knowledge and Software Engineering technology is that Knowledge Engineering approaches are virtually not used in industry, even for the construction of knowledge-based systems [5]: The concept of generic Problem-Solving Methods mostly exists on paper only and few knowledge acquisition tools have outgrown the prototypical stage. Instead, systems are usually built on code level, without comprehensive analysis and design methods, and with tools and languages from Software Engineering. Development tools such as expert system shells are – if at all – used for early analysis phases and prototyping, but the executable system is frequently re-implemented in a more mature and flexible object-oriented platform [148].

The consequences of the limited success of their methodologies are recognized by the community's members themselves. "The Knowledge Engineering community will have to put more effort in cooperating with other disciplines for making their methods and tools more widely known and available" [173]. "Efforts are required along the lines of unifying methodologies to arrive at a situation resembling Software Engineering" [115]. In this spirit, the XP.K methodology attempts to reduce the gap between Software and Knowledge Engineering. I will provide an introduction to XP.K in the following chapter.

5. Overview of the XP.K Methodology

The previous two chapters have identified and compared the basic concepts of Object-Oriented Software Engineering and Knowledge Engineering. The major results were:

- Object-oriented technology is considerably more mature and accessible than Knowledge Engineering approaches, but lacks (explicit) support for knowledge modeling.
- Knowledge Engineering research has produced methods and prototypical tools for knowledge acquisition, but suffers from providing little support for the transitions between the various models and components.

In this chapter, I will provide an overview of XP.K, a light-weight methodology for the development of knowledge-based systems. My goal with XP.K is to help make the development of "real-world" knowledge-based systems more efficient. In order to yield a pragmatic development approach, I have based XP.K as close as possible on industrial software technology. XP.K combines approaches from Software and Knowledge Engineering, so that the weaknesses of one "world" are covered by the strengths of the other. Thus, XP.K is basically an extension of object-oriented standard technology by sub-processes, language elements, and tools to cover the specific requirements of knowledge modeling. In order to meet the previously identified requirements of a feedback-driven, communication-intensive, evolutionary development process, XP.K is an agile methodology which extends Extreme Programming (XP) [8].

Figure 5.1 provides a brief overview of the XP.K process of developing knowledge-based systems. The process relies on communication instead of comprehensive documentation or other artifacts. Starting with a small selection of artifacts such as story cards (use cases) and knowledge protocols (expert interviews), domain experts and knowledge engineers collaborate to define an initial domain ontology, which is a compromise between the informal domain view and the executable system's view. The ontology is modeled in a simple subset of UML with additional declarative elements like constraints, which expose the model's semantics to other humans and computers. The shared ontology is the core model in the system, since it defines the knowledge acquisition vocabulary but is also accessed by other modules such as Problem-Solving Methods, GUIs, etc. The rather conceptual, high-level UML model is automatically forward-engineered into (Java) source code, but can be re-extracted at any time by means of Round-Trip Engineering technology. The Java code is enriched by test code and semantic constraints which support the

modeling process and reasoning algorithms. The domain experts use a generic tool for knowledge (instance) modeling. The tool uses the *semantic transparency* provided by the ontology (using a technology called *reflection*) for the automated generation of suitable editors (trees, tables, forms) and for constraint checking during knowledge acquisition. The developers meanwhile enrich the ontology classes to integrate them with the desired reasoning functionality, e.g. by building subclasses of existing inference classes or by implementing suitable mappings to them. Prototypes are produced frequently and exposed to tests and simulations.

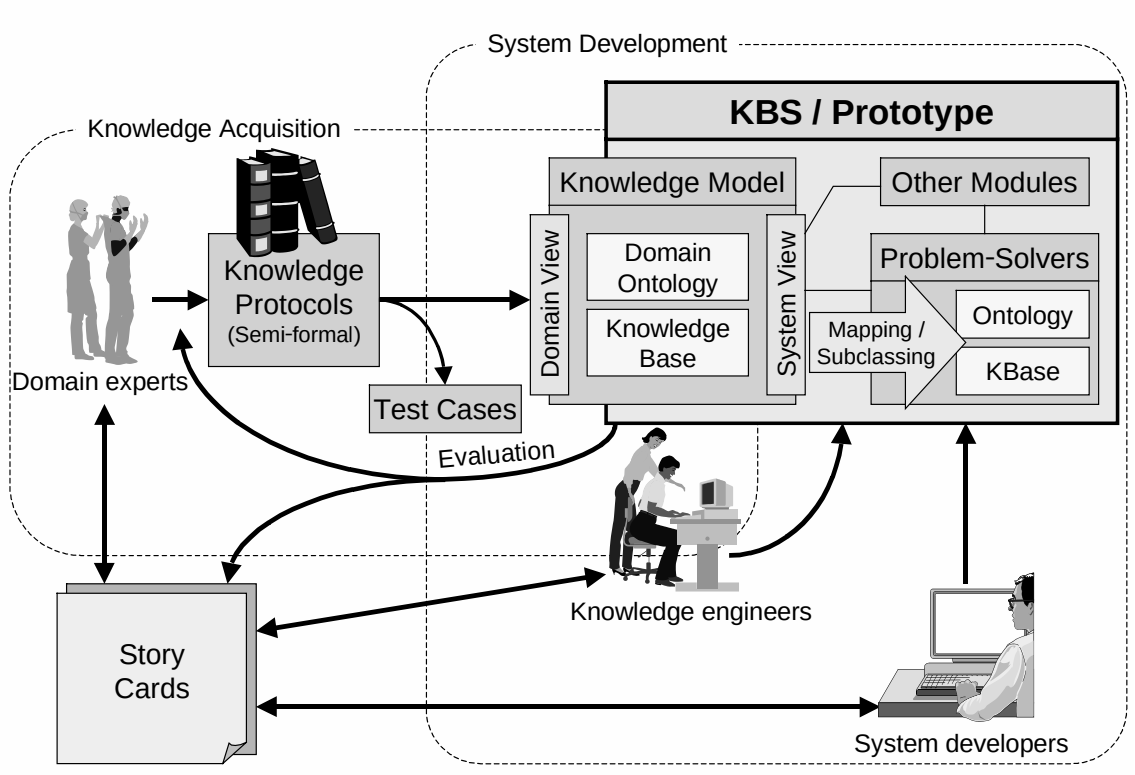

Figure 5.1.: The XP.K development process.

In the following, I will provide details on the paradigms (section 5.1) and processes (section 5.2) of XP.K. Then, I will introduce languages (section 5.3) and tools (section 5.4) which can be used to put XP.K into practice. Details on these languages and tools are elaborated in chapters 6 and 7. Note that in this chapter, I will not discuss potential problems and pitfalls of this methodology, because benefits and limitations will become clearer after the details and case studies have been introduced. An evaluation of XP.K will follow in chapter 9, which will also identify the problem domains for which XP.K is suitable.

5.1. Paradigms

The basic paradigms of the previously described methodologies are Object-Orientation (including objects, components, patterns, architectures, and frameworks) and the generic architecture of knowledge-based systems (including ontologies and Problem-Solving Methods).

I argue that a principal weakness of Knowledge Engineering approaches is to treat knowledge modeling in relative isolation from the remaining application system. As a result, the transitions between the knowledge models and the executable modules are difficult. I have already indicated in figure 4.7 (page 57) that knowledge-processing components require efficient access to resources written in general-purpose languages like Java. Using traditional Knowledge Engineering approaches, this makes it necessary to build hybrid systems, which mix multiple paradigms, languages, and tools, which are hard to maintain and contain various redundancies. Hybrid systems may also suffer from the typical low performance of Artificial Intelligence languages [148].

Another practical obstacle of using Knowledge Engineering technology on a large scale is that industrial standard languages and tools are missing and that languages are not compatible to each other. Both weaknesses lead to relatively high costs of developing and changing knowledge-based systems.

Instead of focusing on the knowledge model as *the* central part of a knowledge-based system, I regard the knowledge model as only one of several other important models. For all these other models (user interface, data management, network infrastructure, etc.), the industrial standards are object-oriented languages and tools, and supporting processes. Thus, the integration of object-oriented technology can not be ignored in knowledge-based systems.

It is fair to ask why developers should rely on special knowledge representation systems at all. "Many of the benefits of knowledge representation systems occur during the process of designing an ontology. This support is undoubtly useful, but in the object-oriented world there is also much support available for the design of models, with mature and commonly used languages, methodologies and tools available" [43]. Is it possible to simply assume Object-Orientation as the starting point and integrate the essential ideas of Knowledge Engineering into it, so that a consistent methodology emerges? I think so.

Let us reconsider the requirements for development methodologies defined in chapter 2. As illustrated in the left part of figure 5.2, the basic requirements identified there were that a methodology should support feedback, change (evolution), and collaboration efficiently.

Supporting feedback requires efficient tools and rapid turn-around times between high-level models and executable code. Supporting evolution requires the modeling artifacts to be maintainable, so that the costs of change are low and that prototyping does not lead to expensive development dead-ends. Supporting collaboration requires modeling languages that are easy to communicate to and between domain experts, but yet sufficiently formal to be useful to developers. My main approach to meet these requirements is that the development process should center around a single shared modeling artifact, more precisely

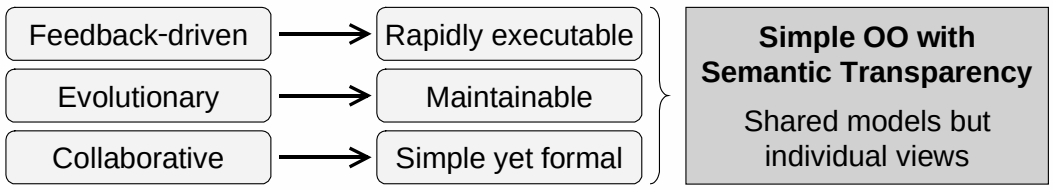

Figure 5.2.: In order to support feedback-driven, evolutionary, collaborative development processes, XP.K relies on simple object-oriented models which expose their syntax and semantics to humans and machines, so that each group of developers can extract individual views.

an object-oriented domain model (ontology). This model is directly executable, supported by standard tools, and maintainable by applying best-practices of Object-Orientation. Objects are relatively close to the domain expert's way of thinking but do not suffer from being as "distant" from executable code as traditional knowledge modeling languages. All groups of developers access different aspects of this model through different *views*: Domain experts only get to see the high-level concepts, relationships and knowledge bases, whereas the system developers focus on the integration of these concepts with the overall software architecture. In order to provide details on how this is approach could work, I will take a deeper look at the differences between objects and ontologies in the following.

5.1.1. Differences between Object-Orientation and Ontologies

As already indicated in section 4.3, there is a large overlap between the concepts of objects and frames. The similarities between both worlds become most obvious if we compare ontologies with object-oriented domain models as in the Model-View-Control Architecture (figure 3.1 (page 25)). Both types of models are abstractions of a given domain. The design principles for class models are almost identical to those from ontologies (cf. section 4.1.1): Modularity, high internal coherence but low coupling, extensibility, minimal encoding bias, and the use of natural categories and names.

Figure 5.3 illustrates how the basic modeling primitives of ontology and object-oriented languages are related to each other. Ontologies are purely *declarative*, i.e. they only describe concepts and the relationships between them. Ontologies and object-oriented languages have very similar approaches for the declaration of static structures, namely classes (concepts), class hierarchies (using inheritance), attributes, relationships, and instances.

Despite these similarities, there are significant differences in the representation of *semantics*. Ontologies use constraints (metadata on slots), axioms, and rules, whereas Object-Orientation relies on methods (sequences of imperative commands). Both approaches are equally expressive, but a declarative approach is better suited for domain modeling than an imperative approach. An example will clarify this.

Figure 5.4 shows a simple domain model representing family relationships. The class

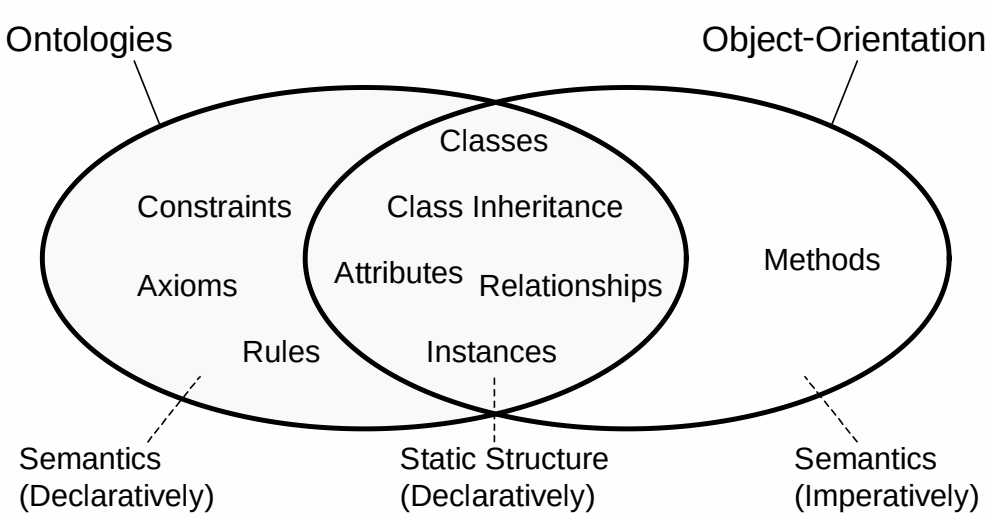

Figure 5.3.: The central modeling primitives of ontologies and object-oriented languages.

diagram shows that even simple domain models can be based on a considerable amount of semantical assumptions. These assumptions are important, because they constrain the space of valid application scenarios, and thus allow to use and reuse models safely and efficiently. However, when class models such the example presented in figure 5.4 are implemented in an imperative object-oriented language like Java, their original semantics are usually obscured behind non-transparent programming primitives. The constraint that a person's age must not be negative might be implemented by `if`-clauses that reject negative values before they are assigned to the `age` attribute (e.g., in the `setAge` method). The constraint that a person's marital status must be one of three predefined constants might be implemented by three appropriate radio buttons. Thus, the semantics are typically implemented *implicitly*, especially by making sure that any execution path through the program will result in valid object states. The complexity of potential execution paths often leads to systems that are error-prone, hard to understand and maintain, and almost impossible to reverse engineer [45]. Most importantly for knowledge-based systems, domain models implemented this way can hardly be used for the explicit representation and exchange of knowledge, and for knowledge-based reasoning tasks.

The origin of these problems is that developers have implemented semantical assumptions that are neither transparent to other developers nor to the software environment in which the domain models are embedded in. I argue that if we could make these assumptions explicit to humans and machines, we can use object-oriented domain models for other tasks than those implicitly encoded by their developers. In particular, we can write domain models that can be used like ontologies, because they explicitly specify the semantics of concepts and their relationships. A key to this is the notion of semantic transparency, which is introduced in the following.

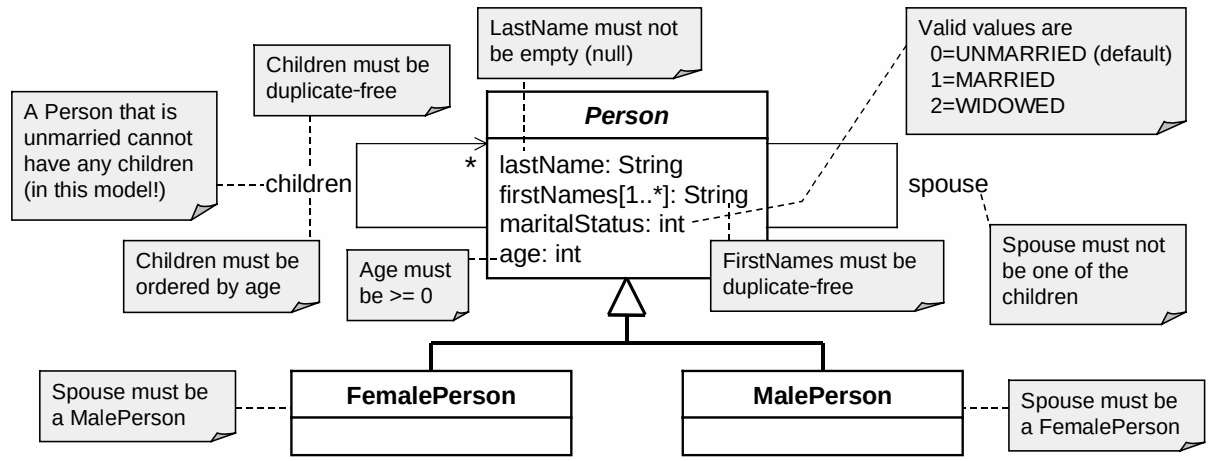

Figure 5.4.: A simple domain model and its intended semantics (UML notation).

5.1.2. Semantic Transparency for Object Models

Semantic transparency means that machines and humans are presented with models that are both *unambiguous* (having a precise, predictably interpreted meaning) and *meaningfully correct* (simultaneously satisfying a number of integrity constraints) [42]. A computer's notion of semantics is restricted to *values* that can be related to other values that have a meaning to humans. Therefore, semantic transparency for machines can be reduced to formal information about the elements of an object model and their relationships. Thus, a key to implementing semantically transparent objects is to provide information (or *metadata*), especially about the valid object states. These metadata could be used to express – among others – constraints, axioms, and rules.

Declarative design languages like UML provide means to express such metadata. As in the example from figure 5.4, design architects can attach comments or formal expressions in the Object Constraint Language (OCL) to UML class diagrams and thus clarify the intended meaning of the classes to other designers. Several projects, including CommonKADS, show that UML is quite an attractive foundation for ontology modeling [43, 164]. However, UML is not executable (at least not in its current version) and the semantic transparency is lost on the move to executable (Java) code, because OCL does not have an equivalent in Java or C#. If we want to express semantics explicitly, and thus make them available as "knowledge" in a machine-readable way to knowledge-based systems, we need to cope with the elements that are available in these programming languages.

In section 5.3, I will describe techniques how object-oriented programming languages can be used to express constraints, axioms, and rules explicitly. In a nutshell, the basic idea is to attach specific constraint or rule objects to classes and their attributes (slots). These metadata objects can be distinguished from other objects both at run-time and build-time. At run-time, the semantic metadata can be used to validate knowledge bases, to perform reasoning by Problem-Solving Methods, and to implement generic, mindful

knowledge acquisition tools. At build-time, the metadata can be edited with "ontology" modeling tools like UML editors. The metadata elements are also ordinary programming language constructs and the domain model classes are normal (Java) classes to the system developers. Using Round-Trip Engineering, this makes it possible to present the domain expert with a model view that is similar to the (intuitive) UML diagram from figure 5.4, although the model is implemented in low-level programming language constructs that are efficiently useful to the system developer. As illustrated in figure 5.5, other groups of developers can be supplied with alternative system views as well. Note that semantic transparency embraces *syntactic transparency*, i.e. the syntactic structure of the models must be formally understandable to machines and humans as well.

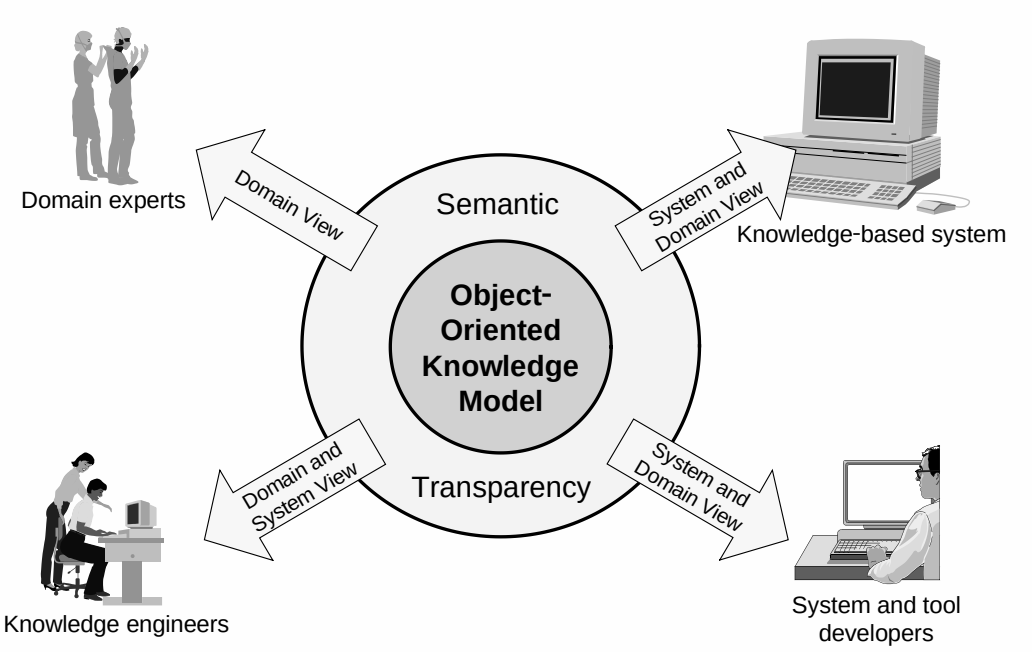

Figure 5.5.: XP.K suggests to use object-oriented technology for building knowledge models which are shared between all stakeholders in the development process. Semantic transparency of these models allows to derive individual views for the different groups.

To summarize, XP.K gives up the clear distinction between object models and ontologies and covers as many knowledge-oriented concepts with standard principles from Object-Orientation as possible, without giving up the advantages of formal, explicit knowledge models. This is achieved by increasing the semantic transparency of the objects to machines and humans at run-time and build-time. Developing knowledge-based systems with XP.K therefore relies on the object-oriented paradigm on the implementation level and on explicit knowledge modeling (ontologies) on the design level. Since ontologies are

the link between domain experts and the system world, we can thus integrate Knowledge Engineering technology into object-oriented knowledge-based systems.

5.2. Process

Now that we have defined the object-oriented paradigm with semantic transparency as the foundation of XP.K, we can examine the process in which the system modeling is being performed. Again, we start with the requirements of modeling processes from chapter 2 (as summarized in figure 2.6 (page 21)). To reiterate, the modeling process of knowledge-based systems should be *collaborative*, *feedback-driven*, *evolutionary*, *flexible*, and *rapid*. Since agile methodologies such as Extreme Programming are based on communication and rapid feedback, and since the software developer community has produced valuable experience and guidelines on how to apply XP in practice, I have decided to base my methodology very closely on XP. XP.K is thus an application of XP to knowledge-based system, or – in object-oriented terminology – a sub-methodology of the XP methodology. This implies that the usual XP techniques and concepts can be applied to all system modules which are not related to knowledge processing in the proper sense.

XP.K does not prescribe a predefined sequence of activities, but is rather of collection of values, principles, and practices which should be followed by the developers. I will explain these dimensions in the following subsections, and then give an overview of how they can be combined and applied in practice.

5.2.1. The Values of XP.K

The values of XP.K include the values of XP, namely *Communication* (to enable collaborative ontology modeling), *Simplicity* (to increase model maintainability), *Feedback* (to enable evolution), and *Courage* (to be able to escape modeling dead-ends). However, XP.K generalizes the value of Communication to *Community*, a term that includes the social constructs that underly the communication. Details on these values follow.

Community. As shown in chapter 2, the development of knowledge-based systems must rely on and support communication. Figure 5.6 illustrates which communication links should be established between the roles involved in a typical XP.K development process.

A productive working environment and a healthy team spirit are requirements of collaboration. The value of Community suggests that teams should spend resources on building and nurturing their collaboration infrastructure. In particular, the principles of Communication and Humility are essential. These are described in the following.

Most of this communication is centered around the definition of suitable ontologies, because they are the main link between the domain world, the knowledge acquisition tool, and the system world. Communication is required because ontologies must necessarily compromise between the views of various stakeholders with partially opposing requirements. Thus, communication is also a way of teaching each other's view point (e.g., training a

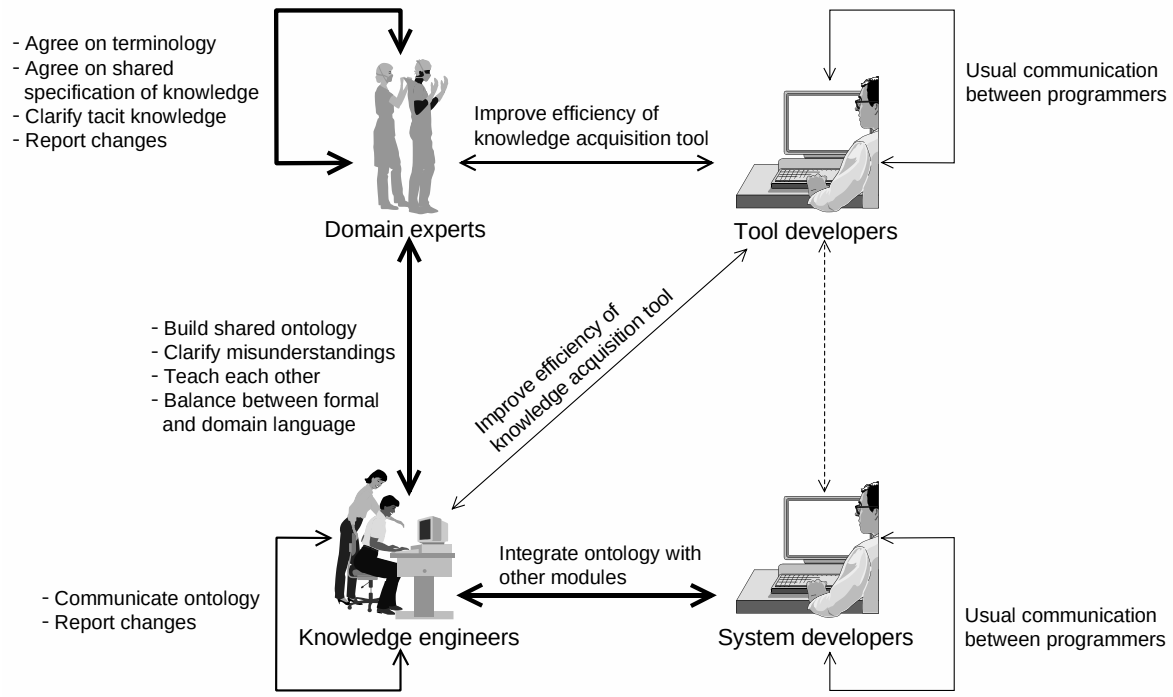

Figure 5.6.: The roles involved in a typical XP.K development process and the purposes of the communication between them. Main communication links are indicated by thicker lines.

domain expert to formal specification methods). Furthermore, communication is essential among the members of each group. For example, several domain experts will usually produce a much more useful domain model than only a single expert with individual preferences. Since fluent communication does not come for free, XP.K supports and even enforces Communication by several practices such as Pair Modeling, On-Site Domain Expert, and Shared Symbol Grounding (see below).

Humility means that all developers involved in a project accept their limited knowledge and respect the attitudes, background, language, and approaches of other developers. The collaboration of very different groups of people in a KBS project enforces a different code of ethics than "normal" software development projects. In particular, the communication between engineers and domain experts, who are often not trained in formal modeling techniques but possess more domain knowledge than engineers, might otherwise lead to misconceptions and social problems. The frequently encountered arrogance of "almighty" software developers towards domain experts who not even understand the semantics of a "simple" flowchart is a major obstacle in project success. Note that in some situations, mutual respect and humility must give way to clear directions and strategic guidelines by an accepted decision-maker. Many XP textbooks stress the importance of a *coach* [8], the task of whom is to solve situations in which the team members are unable to find

sustainable compromises.

Simplicity. The development process of knowledge-based systems is basically a modeling process, in which domain experts and knowledge engineers build suitable domain abstractions. According to Spitzer [169, page 318], *suitable* means to build models "which reflect the domain characteristics as good as possible while being as simple as possible". Simple models are easier to understand and communicate, in particular when people with as different backgrounds as (clinical) domain experts and programmers need to operate with them.

Extreme Programming promotes simplicity for another reason, too. Beck [8] argues that "it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway". This assertion is counter-intuitive to the way modeling is commonly taught in Computer Science courses. Whereas students are advised to think several steps ahead and to make their models fit for potential future application scenarios, I argue that knowledge models are too unstable to make these investments profitable. Wrong perfectionalism and premature rigidity should be prevented [152, page 260].

Feedback. The necessity of constant feedback during the development process has already been motivated in chapter 2. To reiterate, feedback is necessary due to the complexity and vagueness of knowledge and the respective models, the softness of requirements, and the dynamic impact of a system on the application scenario. Feedback producers such as prototypes and simulations are valuable means of clarifying requirements, of evolving knowledge models, of resolving inconsistencies, of enabling communication between developers and experts, and of supervising project progress. Beck [8, page 31] states that "optimism is an occupational hazard of programming. Feedback is the treatment."

XP.K promotes constant feedback throughout the whole development process. First, knowledge engineers define semantic constraints on the ontology that restrict the space of admissible knowledge bases. These constraints can be used to detect inconsistencies early in the process and to guide knowledge acquisition. Second, developers write unit tests for all the Problem-Solving Methods and other modules that could possibly break. This ensures that the system's correctness survives changes in other modules, such as the ontology. Third, prototypes are frequently exposed to potential users and the project management, so that progress can be estimated and the project's steering wheel be adjusted.

Feedback, Community, and Simplicity supplement each other: The more feedback, the easier to communicate. The better the communication, the easier to find out what to test and what the most simple solution will be. Simple systems are also easier to test and most probably lead to feedback more frequently.

Courage. The fourth value of XP.K, Courage, was chosen for the same reasons as in XP. Courage means to entitle developers and domain modelers to make relatively big changes if needed, in order to escape from situations where there is no way forward by only minor

evolutionary changes. The reason why courage is important is that development in XP is being performed at high speed, with explicitly little pre-planning.

5.2.2. The Principles of XP.K

Based on the four values which represent long-term goals, the XP.K development process is guided in the mid-term by various principles (as illustrated in figure 5.7).

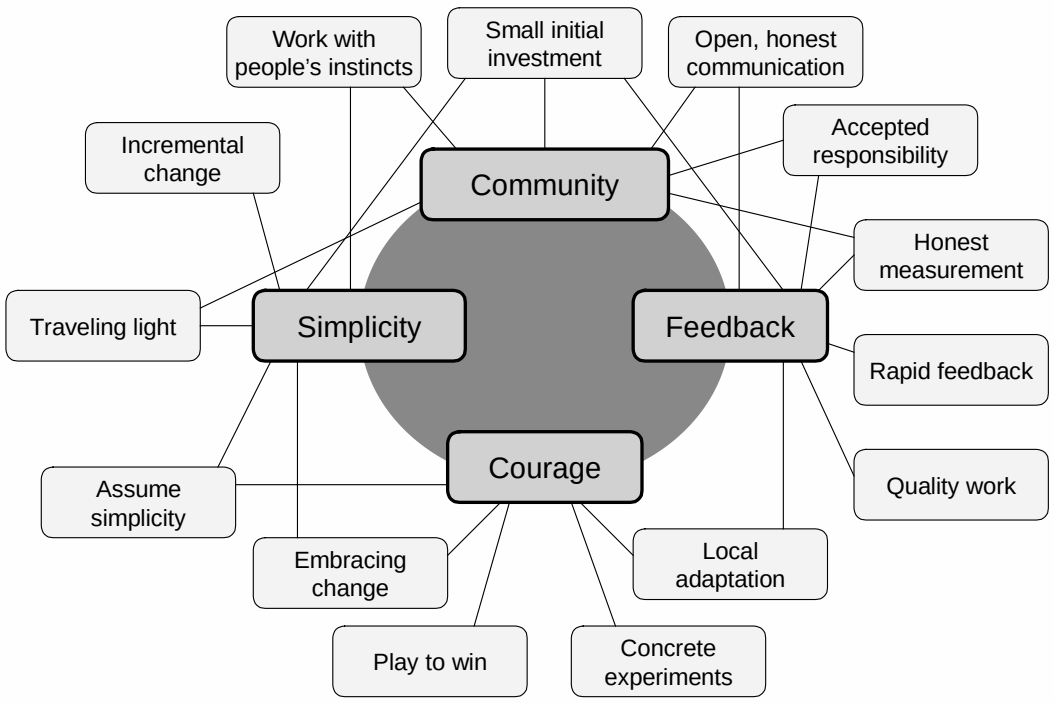

Figure 5.7.: The five values of XP.K (center) and the principles that support them.

These principles are identical to those from XP and therefore already sufficiently described and discussed in the literature (e.g., [8]). However, the following principles are particularly important for knowledge modeling.

Traveling Light. Teams write and maintain as few models as absolutely necessary, in order to reduce the cost of change. For knowledge modeling, XP.K suggests to focus on only one single modeling artifact. This artifact is an object-oriented domain model in the programming language that is also used for the remaining modules. This domain model is enriched by elements that improve its semantic transparency such as declarative constraints and source code documentation. Instead of maintaining multiple models for the different levels of abstraction, XP.K promotes the use of different *views*, in particular a declarative domain view and an executable system view. The principle of Traveling Light thus reduces

the cost of change and supports smooth transitions and interactions between the modeling artifacts for the price of requiring fluent communication links between the different groups.

Rapid Feedback and Concrete Experiments. The notoriously difficult and error-prone knowledge modeling process is supported by automated consistency checks. In detail, this means that the domain experts and knowledge engineers precisely specify the semantics of their ontologies, so that knowledge acquisition tools can validate the knowledge base at design-time. Section 5.4 will present a knowledge acquisition tool that includes a constraint checking mechanism which pro-actively guides the domain modeling activities.

5.2.3. The Practices of XP.K

The values and principles above are put into daily routine by means of various *practices*. As shown in figure 5.8, the practices of XP.K are derived from the original practices of XP (see also subsection 3.2.2).

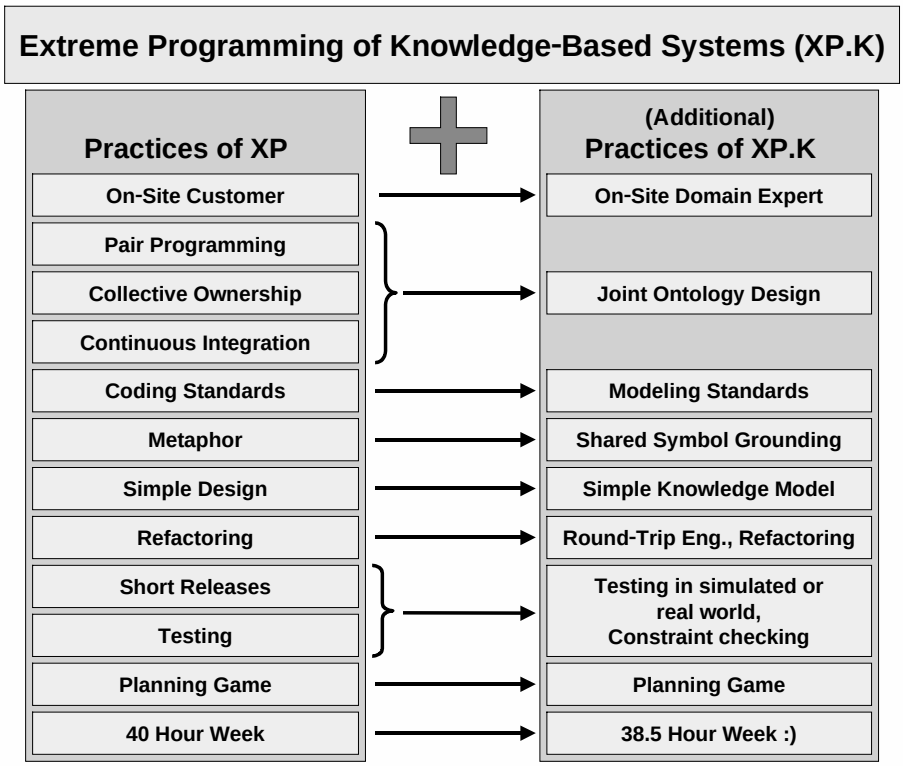

Figure 5.8.: XP.K applies practices from XP to knowledge-based systems.

XP.K relies on the XP practices for those system parts that do not directly involve knowledge modeling, e.g. the development of inference engines, user interfaces, database access, or custom knowledge acquisition tools. For the other parts, in particular ontology

(class) modeling and knowledge (instance) acquisition, the following practices are applied. Note that as already indicated in figure 3.5 (page 35), the practices support each other, so that the weakness of one practice is overcome by the strengths of other practices. This implies that most benefit from using XP.K lies in applying all of the practices in conjunction.

On-Site Domain Expert. XP is performed by small teams of developers (most sources report on 6 to 10 programmers as a suitable team size). The XP.K teams might be slightly larger, because the teams are split into the groups of domain experts, knowledge engineers, tool developers, and system developers (cf. figure 2.3 (page 13)). The groups (which do not necessarily have to be disjoint) can cluster to solve subtasks (e.g., for the development of a particular reasoning engine or agent platform), so that the total team size can be larger than in pure XP. The XP.K team size is limited by the threshold up to which a fluent personal communication is not disturbed. In the ideal environment, the whole team is assembled at a single office site.

Similar to the XP practice of the *On-Site Customer*, XP.K requires that at least one domain expert be available at the developer's site. Due to the aforementioned collaborative nature of knowledge modeling, the task of this expert is to provide information about the domain and end-user world. Knowledge engineers and system developers will frequently encounter situations that they are not sufficiently educated to answer. The domain expert(s) must have the authority to make timely decisions regarding the requirements and prioritization thereof.

Joint Ontology Design and Pair Modeling. The knowledge metamodel (the ontology) is built in a close collaboration between domain experts and knowledge engineers. Both groups represent the (often contradictory) requirements of a knowledge modeling formalism that is at the same time close to the domain terminology and experts' way-of-thinking, and efficiently processable by computers. Compromising on such an ontology will typically be much easier than it sounds, because the number of concepts is usually quite small (e.g., one of our clinical decision-support systems contained about 20 relevant domain concepts). It is therefore frequently possible to build an initial ontology within a day or two. However, ontologies might need to be changed regularly, in particular when the domain experts have gathered experience in building knowledge bases on top of them, identified missing features, inconsistencies, ill-defined concepts, concepts that are not really used, and so on. In those cases, all groups of developers that operate on the ontologies will have to agree on the changes and synchronize their code changes.

Joint Ontology Design has three counterparts among the XP practices. XP suggests *Pair Programming*, in which two programmers collaborate on a single machine, *Collective Code Ownership*, and *Continuous Integration*. In the object-oriented ontology language proposed later, the domain experts will usually not have the know-how to autonomously change ontology classes, so that pair modeling is essential anyway. The close collaboration between experts and engineers also promotes learning from and teaching each other

(cf. [38]).

Joint Ontology Design can significantly reduce development costs: "Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging" [153].

In contrast to ontologies (classes), the knowledge instances can usually be modeled by the domain experts alone, without assistance by engineers, but supported by the later described tools. Following the very idea of XP, I suggest to let domain experts model their knowledge in pairs. Such a practice of *Pair Modeling* can significantly improve model quality, since tacit knowledge and personal preferences are generalized, so that the domain experts' logic approaches the engineers' logic (cf. subsection 2.2.3).

Round-Trip Engineering (see subsection 3.4.2) is a key enabling technology of XP.K. It allows to generate high-level (UML) domain models from low-level (Java) source code and vice-versa, and thus supports Communication, Simplicity, Rapid Prototyping, and Joint Ontology Design. The XP.K knowledge modeling language enables Round-Trip Engineering because it consists only of concepts that are available in both UML and programming languages like Java.

Round-Trip Engineering extends traditional waterfall-based development approaches, in which changes to an implementation must be accompanied by changes to the preceding high-level models such as analysis and design models. In waterfall-based approaches, the dependencies are mostly top-down with no enforced bottom-up correspondence between the implementation and its design. "The practical result of the lack of enforced correspondence between the model and the implementation is that the model becomes increasingly less useful during the software development process" [88]. The consequence of this is that "a key issue will be the mechanism used to establish the correspondence between the abstractions of the modeling language and the constructs of the programming language" [88]. Round-Trip Engineering enforces such a correspondence and thus reduces the need to expend resources on the maintenance of depending documents. Round-Trip Engineering also supports requirements tracing [91].

Modeling Standards. The XP practice of *Coding Standards*, which is a key to Pair Programming and Collective Ownership, is extended in XP.K to *Modeling* Standards. Modeling Standards are agreed-upon naming conventions (e.g, "relations should be verbs"), guidelines for the visual representation of model elements (e.g., UML class diagram layout), and syntax rules (e.g., "classes start with an uppercase letter"). Since our ontology language (see the following section) is very simple, there are only few such standards necessary. Standards are essential to enable model exchange, collaborative modeling, and to minimize the dependency of the team on specific members. Furthermore they contribute to the original vision of ontology research to enable global knowledge sharing.

Shared Symbol Grounding. In XP, the team members are guided by a shared vision, or system *Metaphor*, which can take the role of a system architecture. XP.K extends the

notion of metaphor to a common symbol grounding. *Symbol Grounding* [81] means that the semantic interpretation of a formal symbol system (e.g., the domain terminology) is made intrinsic to the system, for example by means of "iconic" representations. All groups of developers should share an accessible interpretation of the domain concepts and their representation in the knowledge models. For example, during the construction of a decision-support system for anesthesia we found the "pilot" metaphor very helpful: Similar to an airplane's take off, anesthetists have to put the patient asleep. During the airplane's flight, pilots have to monitor the devices for altitude and fuel, just like the anesthetist has to control the patient's respiratory parameters. Finally, the plane's landing can be compared to returning the patient safely back from her artificial sleep. The pilot metaphor allows GUI developers to understand the problems of information overload that are typically connected to anesthesia. Images and metaphors have a much stronger impact on our opinions and the allegedly rationalizing technology than sentences and formal statements [157, as quoted in [152]].

Such a symbol grounding must be shared among developers, so that the domain experts know the engineer's metaphors and vice versa, avoiding misconceptions. Additionally, Symbol Grounding relies on a vocabulary that is well-defined and understandable for all team members (either organized in textual form like the *dictionary* in the Fusion method [40] or on less rigorous media such as speech). Such a shared terminology is essential to enable smooth communication paths.

Testing and Constraint Checking. Beck [8, page 57] states that "any program feature without an automated test simply does not exist". XP.K applies the XP practice of writing unit tests to knowledge modeling. Ontologies and business logic algorithms like Problem-Solving Methods are enriched by test methods which ensure that the intended semantics are indeed implemented by the models. Guidelines on how knowledge bases can be tested systematically can be found in the literature (e.g., [123]). Approaches such as constraints and Design-By-Contract [125] are an alternative (or extension) to unit testing in XP, because they simplify the verification of program states and reduce the coding overhead of test cases. Many errors can be detected early during knowledge acquisition, in particular by constraint checking mechanisms built into knowledge modeling tools. Complex knowledge models can be checked by exposing them to simulated worlds (cf. figure 2.5 (page 17)). "Dynamic analyses have the advantage that detailed information about a single execution is typically much easier to obtain than comparably detailed information that is valid over all executions" [88]. Such automatic tests are crucial to ensure that the models survive changes, in particular that the overall consistency is not broken by local modifications. I argue that tests and constraint checks are nearly as important as the actual models themselves.

Simple Knowledge Model. Simplicity is a key to making models easy to understand, communicate, and maintain. Similar to the XP practice of *Simple Design*, the right knowledge model in XP.K is the model that runs all the tests without constraint violations,

has no duplicate logic, has the fewest possible concepts, slots, and instances, and provides semantic transparency for developers and machines. Knowledge models should only represent what is needed to solve the given tasks, i.e. modeling should be driven by the model's purpose. For example, a clinical expert system does not require a comprehensive model of the human body, but suitable abstractions like observable patient states.

Since ontologies must be understood by domain experts, who are potentially inexperienced computer users, the language of the ontologies themselves must be easy to understand, with as few formal elements as possible. The very simple object-oriented metamodel of XP.K is sufficiently simple and compatible to major industrial standards to allow for smooth transitions between the views and tools, yet sufficiently powerful to allow for the definition of custom knowledge modeling languages for the respective domain.

Simplicity can be regarded as the opposite of the approach taken by many systematic engineering processes, where the design should anticipate future requirements. Instead of spending resources in finding the most elegant design, XP suggests to develop simple models rapidly. The rationale behind this is that simple, agile models are most probably easier to change. The Knowledge Engineering literature contains several examples in which simplicity has lead to astonishing results. For example, the simple mechanism of *Ripple-Down-Rules* [124] works well even without a comprehensive analysis phase. Its main idea is to express knowledge by starting with simple rules and refining them iteratively. Preston et al. [149] report on a knowledge-based system with 1600 rules that was built with such an evolutionary approach.

Finally, simple models are much easier to process by algorithms, for example because they are easier to transform and integrate. "Simple data models have an advantage over complex models when used for integration, since the operations to transform and merge data will be correspondingly simpler" [31].

Refactoring and Design Patterns. The XP practice of Refactoring (i.e., improving the design of existing code without breaking its functionality) can be applied to knowledge modeling as well. On ontology class level, refactoring means to apply the well-known refactoring patterns from object-oriented programming (e.g., "Change bidirectional association to unidirectional" or "Move field" [63]). On knowledge instance level, refactoring can mean to combine duplicate model elements, to improve visual models (e.g., by rearranging nodes in a graph), or to extend the documentation of elements that have proven to be stable for various releases). The *Theory Refinement* [72] community has produced various approaches on how knowledge models can be systematically simplified. The common goal of all these activities is to keep the system as simple and maintainable as possible.

Another important application scenario of refactoring is the generalization of existing models for future reuse. In particular, it may be worthwhile to spend resources on making existing ontologies or knowledge acquisition tools generic. For example, when a tool developer has written a graphical widget to edit process workflows (i.e., process knowledge), she might be able to generalize its underlying ontology, so that other knowledge-based systems can reuse the module. Another example is taken from the ANIS project, in which

I have developed a small rule engine. During and after the project I have generalized the engine's input ontology, e.g. I removed all hard-coded references to medical concepts like `PatientState` and replaced them by generic abstract classes such as `NumericVariable`. These simple refactorings enabled to reuse the same metamodel in different projects (for details, see subsection 8.1.4).

A further potential for improving development efficiency lies in the application of object-oriented *Design Patterns* to knowledge modeling. For example, the *Composite* pattern [67] is applicable to many situations in which domain concepts form a hierarchy (as shown in figure 8.4 (page 133)). Finding and annotating design decisions with Design Patterns can help to improve communication between modelers and support reuse.

Planning Game. The XP practice of the Planning Game suggests that the development process should be guided by simple "story cards", which balance business priorities against the technically possible. In XP.K, the domain experts stack these story cards according to their priorities (i.e., the most valuable features first) and the developers attach estimates about the required resources. This is repeated until a balance is reached. Thus, experts and engineers jointly decide on the project's schedule.

5.2.4. Putting it all Together

The values, principles, and practices described in the previous subsections are the major ingredients of the XP.K methodology and the main things developers should be aware of. XP.K does not prescribe a sequence of modeling activities that should be followed as in systematic engineering methodologies. Approaches such as the ontology development methods from subsection 4.2.1 may be useful during the early stages for the construction of an initial base model, but once a system is implemented and integrated, XP.K does not spend resources on comprehensive analysis and design activities. Instead, it follows the philosophy of agile methodologies, in which agility in the face of changing requirements and knowledge models is a major goal. As illustrated in figure 5.9, XP.K applies various technologies and practices with the explicit goal of reducing the cost of change.

Much of the assumption that the cost of change can be reduced is based on the value of *Simplicity*. I argue that simple object-oriented models are easier to change than complex formal models in traditional ontology languages. The models are integrated to prototypes rapidly and frequently, so that expensive misunderstandings can be uncovered and eliminated early, when the costs of changing them are still low. The principle of *Traveling Light*, which is implemented by practices such as Round-Trip Engineering of ontologies reduces code duplication, conversions, and leads to single-paradigm systems that are easier to maintain. Instead of having to update a sequence of models as in systematic engineering approaches, XP.K teams mainly need to adjust only a single model. The application of design patterns and refactoring can further contribute to model maintainability and future reuse. Knowledge-based systems developed this way have a large potential for reuse of models and code, because the high-level views of ontologies can be extracted from the specific application scenario. Reuse can also take place in the context of tools: XP.K

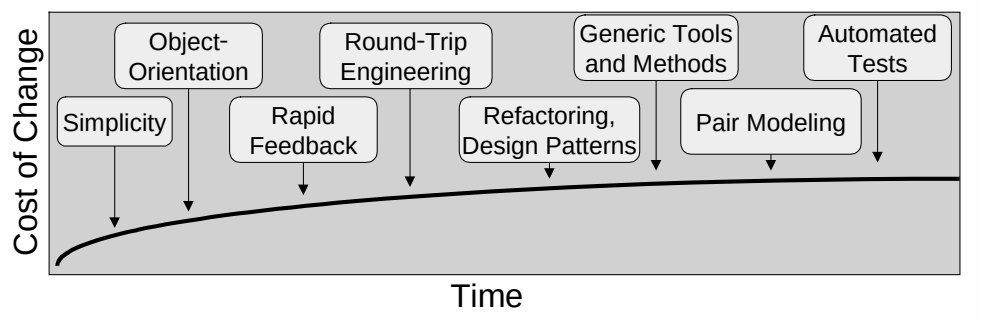

Figure 5.9.: XP.K applies a number of technologies and practices that help to reduce the cost of change.

supports the construction and reuse of generic knowledge acquisition tools. By means of semantic transparency, these tools can operate on any ontology and are therefore robust to ontological changes. Pair Programming and Modeling help to detect errors earlier and spreads knowledge. Finally, comprehensive automated tests and simulations help to build robust and correct knowledge bases.

A word of caution must be brought forth when stating that Extreme Programming flattens the traditional cost curve. Cockburn [36] has shown that the exponential cost curve described by Boehm [18] is still in place. For example, an error hidden in a piece of software that is delivered to customers is far more expensive to fix than if it were detected prior to delivery day. Clearly, the major impact on costs have changes that are motivated by late phases (like end-user tests), but that must be fixed in earlier phases (like requirements). Extreme Programming fights this exponential effect by keeping the iterations between requirements and customer feedback extremely short. For example, the practice of the On-Site Customer makes sure that requirements definition and tests can be performed very rapidly. The practice of Automated Unit Tests ensures that errors are detected prior to delivery.

While all these practices have individual weaknesses when applied in isolation, their main benefits emerge when used together. The practices of XP.K support each other. For example, a weakness of the Round-Trip Engineering approach for knowledge modeling is that changes to one ontology class can also have an impact on several other modules based on the ontology. At the same time, the weakness of writing tests is their additional coding overhead. However, when combined, both weaknesses can compensate each other, because test cases make sure that local changes did not damage other modules, and writing tests is much easier when the ontology is part of the test programming environment. For example, the most frequently used XP testing tool, *JUnit* [66], is written in Java as are our ontologies.

Despite being based on the interplay of various practices, XP.K can and should be adjusted to local settings. In particular, the principle of Traveling Light may be softened to allow for the inclusion of additional modeling artifacts in the process. Medium-sized

knowledge-based systems will certainly require to fix requirements or important knowledge sources in paper-based documents. Examples include interviews with external domain experts (the so-called *knowledge protocols* [4]), clinical statistics, informal treatment plans, etc. For example, prior to building a clinical multi-agent system, we produced a comprehensive model of the clinical processes as they were before we started to modify them. Such artifacts will remain quite stable throughout the process, so that they can be referred to from knowledge models to improve requirements traceability [91].

5.3. Languages

In this section, I will describe a language for the declaration of knowledge models (ontologies and knowledge base instances). In the context of the metamodel hierarchy introduced in figure 3.7 (page 38), this language represents the M2 Layer. In the following, the language will be called XP.KL. I will not rigorously specify the language and its semantics, because (1) the language is very simple, (2) formal specifications are inherently hard to read, and (3) I will provide a detailed Java implementation of it in chapter 6. A formal specification of XP.KL in the Meta Object Facility (MOF, see subsection 3.3.2) is relatively straightforward, because XP.KL is essentially a subset of the UML, which is in turn also defined in MOF. Note that XP.K is not bound to XP.KL, but most of its concepts can also be applied in conjunction with similar languages.

XP.KL is a simple language for the declaration of ontologies and knowledge bases. The main goal of the XP.KL is to integrate smoothly with the values, principles, and practices of XP.K. In particular, XP.KL supports *Simplicity*, *Community*, *Rapid Feedback*, and *Traveling Light*. Figure 5.10 shows the classes of the XP.KL in UML/MOF notation. In the following, I will provide details on the diagram elements, by relating them to the Knowledge Engineering language OKBC (see subsection 4.3.1) and the object-oriented languages UML and Java.

An XP.KL *ontology* encompasses the terminology and structure in which knowledge bases are instantiated. An ontology is a collection of classes. A *class* represents a concept from the domain of interest, comparable to the classes of OKBC, UML, or Java. In terms of the Model-Control-View architecture from figure 3.1 (page 25), classes are purely *model* classes that mainly serve as information containers. Classes have names, which must obey the usual naming conventions from the object-oriented world (i.e., start with an uppercase letter, exclude whitespaces, etc.). Classes can be grouped into named packages and are either abstract or instantiable. Classes can be arranged into hierarchies, but when used in conjunction with Java or C#, only single inheritance is permitted. *Instances* are objects as in the Java/UML world. A *KnowledgeBase* is a collection of instances. The instances are acquired during knowledge acquisition, and usually remain unchanged throughout the execution of the knowledge-based system.

The characteristics of instances are defined by the properties of its classes. The term *Property* is borrowed from the JavaBeans specification [178], details of which will follow in chapter 6. Using Java, properties are implemented by public getter and setter methods.

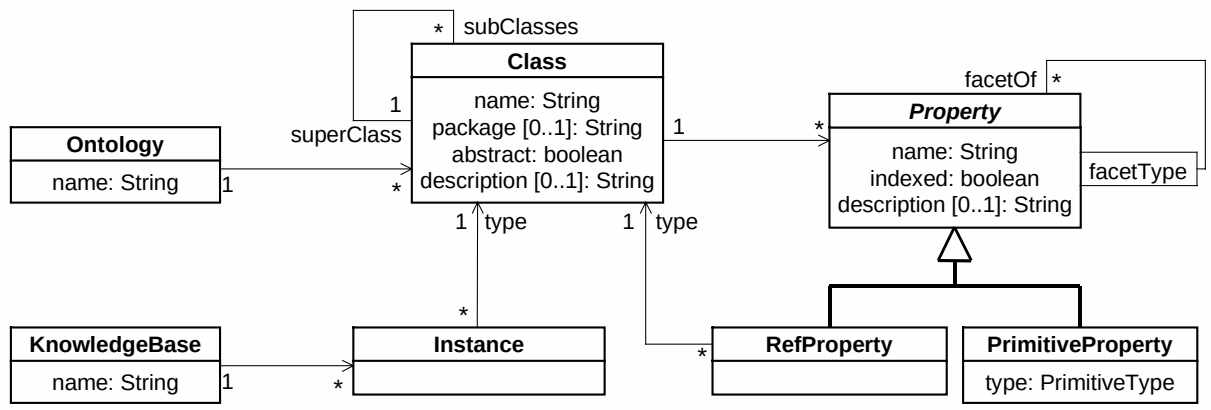

Figure 5.10.: The XP.K knowledge modeling metamodel (in UML notation).

Properties fulfill the role of the template slots in OKBC or public attributes in UML. Properties are either simple or indexed. Simple properties store a single value, whereas indexed properties store arrays of values (although the ordering of elements is seldom important). Properties are either associations (reference properties) or primitive values. A *RefProperty* represents a link from one object to another. Links are directed, because bidirectional links are harder to maintain and reverse-engineer. A *PrimitiveProperty* can take values of predefined types (in particular `int`, `double`, and `boolean`) or of classes that are not ontology classes (e.g., `String` and `Date`).

XP.KL requires that the implementation platform (e.g., Java) supports object-oriented *reflection* [118]. Reflection allows to access metadata about objects, classes, and properties at run-time. Using these metadata, programs can find out the attributes of knowledge base instances and their classes dynamically, so that generic algorithms and tools can be constructed. Reflection is therefore an essential technique for the implementation of semantic transparency. Chapter 6 will provide details on this.

So far, the metamodel is basically an extremely light-weight version of object-oriented domain models without methods. Thus it should be very easy to learn and apply (even) by domain experts. Since the language can be regarded as the common subset of UML and Java, support for Round-Trip Engineering is excellent. At the same time, its intended simplicity and compatibility with object-oriented standards are limitations, because they prevent us from adding new language elements such as those from ontology specification languages. In particular, XP.KL copes with its own primitives to represent semantic transparency. The powerful concept of *facets*, which are used in OKBC to express semantical constraints on the valid states of domain concepts, can therefore only be supported by means of an artifice. The basic idea is to *reify* properties, so that properties are used to describe and constrain other properties. These "meta-properties" take the role of facets without adding new language elements.

Facet properties are associated to a *facetType*, which defines a property's semantics.

A catalog of pre-defined facet types is introduced in chapter 6. This catalog essentially covers the most frequently needed types of constraints (e.g., those from OKBC (table 4.1 (page 53)) such as the minimum value, maximum cardinality, and enumerations of admissible values). Ontology authors are however able to add custom facet types. A class accesses its defined facets through the name of the facet type (e.g., `maxCardinality`). In languages such as UML and Java, the link from a property to one of its facets can be implemented implicitly, by following simple naming conventions (e.g., the property `childrenMaxCardinality` declares the maximum cardinality of the `children` property. Reflection can be used to detect and access the declared facets.

Apart from constraints, some ontology languages support the specification of functions, axioms, and rules. *Functions* specify that the value of a given attribute can (and must) be derived from the values of other attributes. In this view, functions are constraints that restrict an attribute's value to exactly one other value. *Axioms* are complex restrictions on the valid configurations of several attribute values. Thus, axioms are more expressive than constraints. Staab and Maedche [171] have shown that the most common types of axioms can be generalized to *axiom types* which can be converted to constraints. Another mechanism to translate axioms to constraints is to split axioms into multiple constraints, each of which describing the admissible values of the single attributes involved in the axiom. This mechanism, however, can lead to combinatorial explosions, because axioms that put multiple attributes into relation must be resolved for each of these attributes. Where these conversions become inefficient or fail, axiomatic expressions can be moved to the object level by defining specific axiom classes which are instantiated or overloaded. *Rules* can be represented in various ways by means of objects. For example, as shown by Pachet [143], rules can be implemented through specific object-oriented methods. This approach defines one method for each rule, the parameters of which specify the rule's premise or left-hand-side and the body of which performs the conditional check and then the rule's conclusion (which may be to change other attributes). Further axiom and rule declarations could be implemented by means of the UML Constraint Language OCL [194], although in this case individual conventions or tools like OCL compilers need to be introduced to enable their Round-Trip Engineering with Java. The round-trip approach is summarized in figure 5.11.

The presented language is slightly less expressive than traditional ontology representation languages like OKBC. XP.KL neither supports multiple inheritance, nor the construction of slot hierarchies. Rules and axioms are not explicitly supported, but shifted to the object level. This, however is not a serious limitation. As already stated in section 4.1, early expert systems failed exactly due to their trying to the capture all knowledge types by means of uniform logical rules, whereas Knowledge Engineering suggests to express such knowledge in the domain expert's preferred ontology. For example, the expert can formalize rules in terms of symptoms and diseases instead of writing uniform, formal first-order-logic sentences. Furthermore, graphs of related objects are easier to display visually than plain rules. It is questionable whether advanced types of knowledge representation like axioms are needed in average projects [43], so that the benefits of using traditional ontology languages have to be balanced against their aforementioned weaknesses.

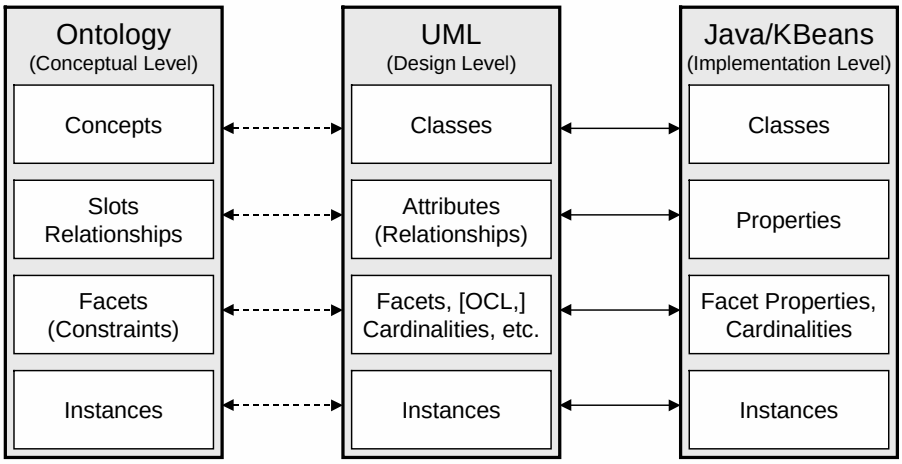

Figure 5.11.: Round-Trip Engineering of knowledge models between the conceptual domain view, a formal visual UML view, and the system view (here with Java classes).

Although XP.KL is less expressive than specialized knowledge representation languages, industrial practice with object-oriented software has shown that its modeling primitives are a suitable way of modeling domains. The central task of ontologies, namely to provide a formal, explicit specification of concepts and relationships, can be fulfilled with XP.KL quite well. XP.KL supports all of the requirements defined in chapter 2, namely that it is adaptable, easy to learn, translatable, incremental, precise, complete, and – in particular – *compatible*. Its compatibility allows to export knowledge models into other object-oriented languages or knowledge exchange formats. Ontologies implemented in XP.KL can be smoothly integrated with other modules, such as reasoning methods, GUIs, and knowledge acquisition tools (as shown in the following section).

5.4. Tools

Each of the roles involved in an XP.K development process should be supported by adequate tools. Developers of system code and knowledge acquisition tools can access the large pool of object-oriented standard development tools such as CASE tools and IDEs. The next two subsections focus on tool support for the two other developer groups, namely knowledge engineers and domain experts.

5.4.1. Ontology (Class) Editors

With the simple object-oriented metamodel from the previous section, ontology classes can be visualized and edited with off-the-shelf UML editors. Probably the most suitable standard tools are modern Round-Trip Engineering tools for UML like Together for Java [184],

a screenshot of which has already been shown in figure 3.8 (page 40). Such tools are mature, robust, and in the case of Together even free. The visual editing facilities of UML tools simplify communication with domain experts. However, CASE tools have an overwhelming number of features and opportunities to make mistakes, so that domain experts will usually not be able to use them on their own. Since XP.K implements the practice of Joint Ontology Design, it is no problem to let a knowledge engineer perform the changes.

An open issue of using standard CASE tools is how the knowledge bases (instances) can survive schema evolution. Although the UML specification includes Object Diagrams, which are equivalent to our knowledge bases, few tools provide suitable support for them. I will describe tool support for these issues in chapter 7, admitting that handling schema evolution is expensive. However, these problems are also present in Knowledge Engineering tools.

5.4.2. Knowledge Base (Instance) Editors

The simplicity and compatibility of the XP.K knowledge metamodel allows to include a huge amount of editor types into the process. For example, it is easy to define mappings between the ontology classes and relational databases, so that off-the-shelf database tools can be used and/or extended [22, 191]. Apart from these tools, which are seldom optimized for knowledge modeling, the Java incarnation of XP.K comes with a specialized tool called *KBeansShell*, which is in detail presented in chapter 7. In order to make the present chapter relatively self-contained, here is already a brief overview of this tool.

The XP.K knowledge metamodel has explicit support for reflection, i.e. the ability to access metadata about the structure of (Java) classes at run-time. I have exploited this feature for the development of various *generic* instance editors, which can operate on any Java data structure that follow certain coding conventions (the JavaBeans [178] specification, which implements XP.KL). These instance editors can visualize knowledge bases as trees, tables, forms, graphs, and others. The editors are part of the KBeansShell platform for the development of knowledge acquisition tools. The shell provides a framework in which the generic editors or customized own editing components can be placed. While the ontology is relatively unstable, the generic standard editors are used, but when the system matures, more and more optimized components can be integrated. Figure 5.12 shows some of the editing facilities that the KBeansShell automatically creates for the example ontology from figure 5.4.

The KBeansShell uses the XP.KL facet mechanism to evaluate constraints, so that the domain experts and knowledge engineers are pro-actively notified about inconsistent knowledge bases and get advice on how to repair them. Custom unit testing and simulation functions can easily be integrated.

The KBeansShell platform is extremely flexible and open. The customization of the knowledge modeling process by means of user-defined components is probably one of the most rewarding features of the whole Round-Trip approach, because Java-based ontology classes can be extended with support code without breaking its semantic transparency.

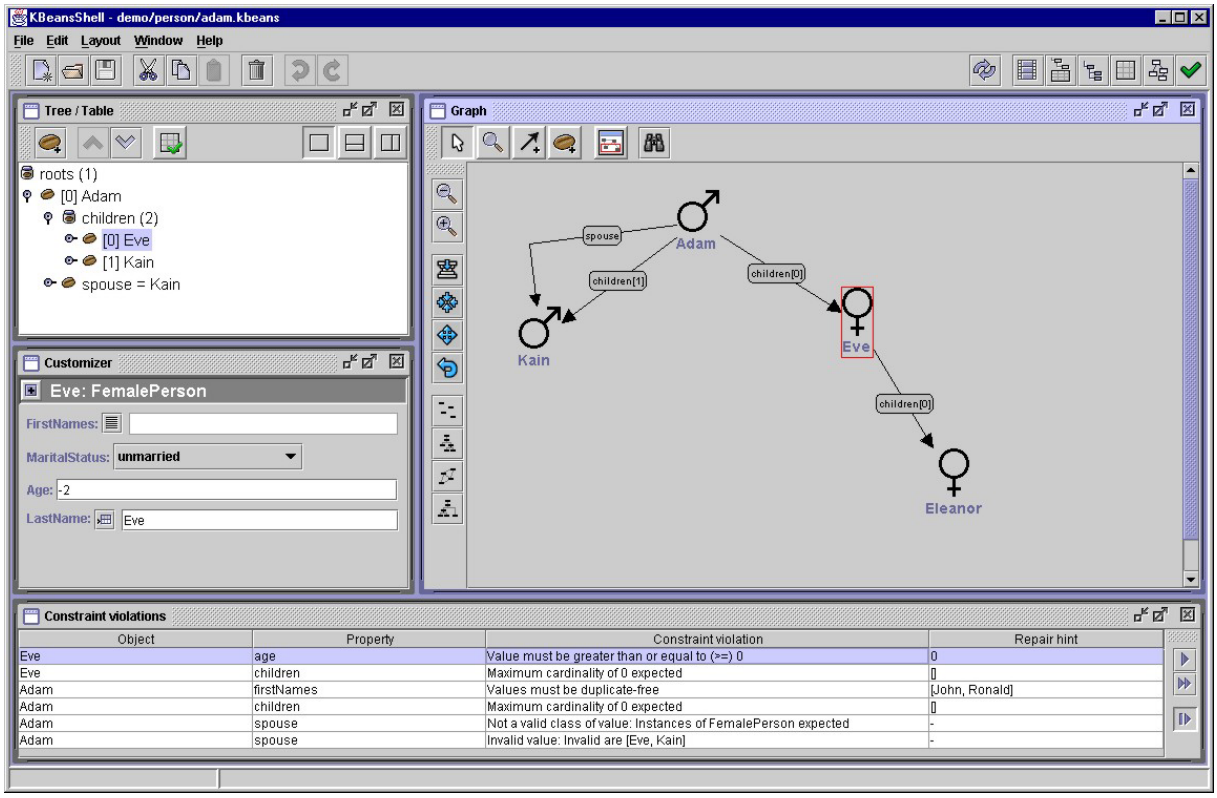

Figure 5.12.: The KBeansShell, a generic JavaBeans instance editor and knowledge acquisition tool.

5.5. Summary

This chapter has provided an overview of XP.K, a pragmatic methodology for the development of knowledge-based systems. As illustrated in figure 5.13, XP.K combines approaches from Software and Knowledge Engineering, so that the weakness of one "world" is covered by the strength of the other.

XP.K extends the usual object-oriented modeling primitives by approaches that increase the model's semantic transparency. It supports smooth transitions between conceptual knowledge models and executable code through a simple metamodel that can easily be engineered in a round-trip style. Furthermore, it allows to use various robust and widely available tools for knowledge modeling and supports reuse of generic knowledge acquisition components. Its compatibility to object-oriented standards makes XK.P a very accessible and flexible approach, because it allows to access mature and future technologies, components, frameworks, architectures, and modeling methods.

The overall process of XP.K is very light-weight and agile, so that the developers get feedback rapidly and frequently, while being able to cope with change efficiently. Instead of requiring the maintenance of comprehensive documentation artifacts, XP.K relies on

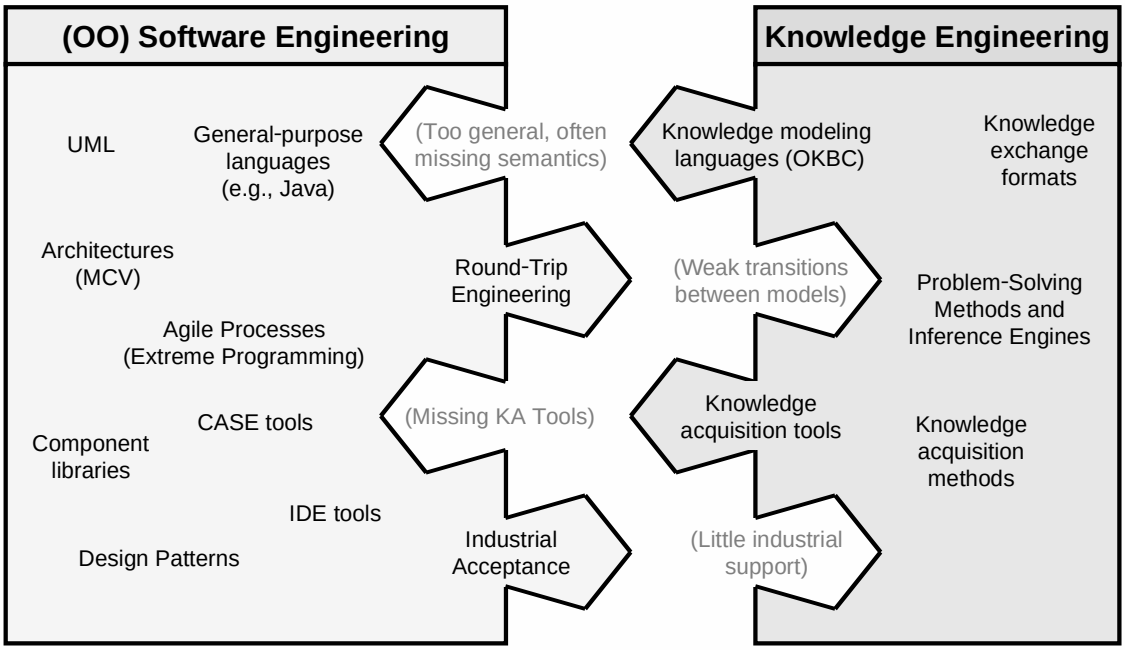

Figure 5.13.: Technology and methods from Object-Oriented Software Engineering and Knowledge Engineering are merged in the XP.K methodology.

communication, automated tests, and semantic transparency of the executable code.

The following chapters will provide details on the language (chapter 6) and tools (chapter 7) of XP.K, and report on our experience with it from various application scenarios (chapter 8). The limitations and problems of XP.K will be discussed in the final chapter 9.

6. KBeans: Implementing XP.K in Java

XP.K is founded on the principle of Traveling Light and practices like Round-Trip Engineering and Joint Ontology Design. For pragmatic reasons, this means to give up the dependency on specific ontology representation languages, but instead to rely on pure Object-Orientation throughout all project phases. In section 5.3, I have discussed the differences between ontology languages and object-oriented programming languages. My conclusion was that objects are missing *declarative* semantics which are necessary to enable ontology sharing, knowledge acquisition, and other reasoning tasks. If we want to employ objects for knowledge modeling, we need to increase their semantic transparency, so that the "meaning" of the knowledge representing objects and classes is accessible to humans and machines both at build-time and run-time. A computer's notion of "meaning" is restricted to *values* that can be related to other values that have a meaning to humans. Therefore, semantic transparency for machines can be reduced to formal information about the elements of an object model and their relationships. Thus, a key to implementing semantically transparent objects is to provide information (or *metadata*), especially about the valid object states. *Constraints* like those provided in the ontology language OKBC are a very important type of metadata. Facets (slots that describe other slots) are the modeling primitive for the representation of constraints.

An implementation technique for these metadata must cope with the language constructs available in the target language. *KBeans* (Knowledge JavaBeans) [101] is a simple implementation technique for semantically transparent class structures in Java. It can be used to implement knowledge models (ontologies and knowledge bases) in pure Java, but is also a valuable programming aid for the development of reusable components in general. KBeans can be regarded as an implementation of the XP.KL language from section 5.3. An earlier version of the approach has been published in [107]. Further information and supporting software are freely available from the KBeans Homepage [51].

KBeans is an extension of the JavaBeans [178] specification, which defines coding conventions and a reflection-based API with supporting classes for the implementation of reusable components. Some background on JavaBeans and its application of reflection is provided by the following section 6.1. After that, section 6.2 introduces the basic idea behind KBeans. Section 6.3 presents a catalog of predefined KBeans facet types. Section 6.4 shows how KBeans can be used in various application domains, in particular for implementing ontologies. Section 6.5 describes related approaches to enriching Java with semantic metadata and discusses benefits and limitations of KBeans. The chapter is summarized in section 6.6.

6.1. Reflection and JavaBeans

The Java *Reflection* [118] API contains – among others – methods to identify the names and types of class attributes and methods. This allows to implement generic algorithms that operate on *any* class structure by dynamically exploring the properties of these classes. For example, as illustrated in figure 6.1, it is possible to write an algorithm that dynamically invokes all methods that are available for a given object (here, the `MalePerson` instance).

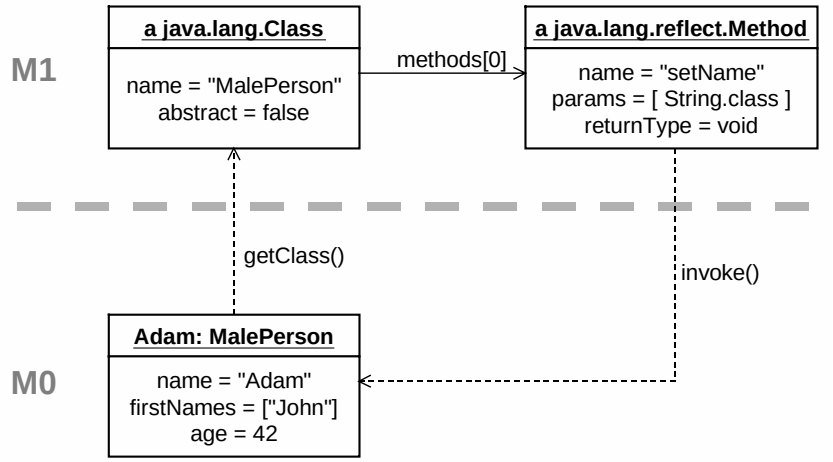

Figure 6.1.: Object-oriented reflection in Java can be used to access metadata from the class (M1) level for objects from the instance (M0) level.

The *JavaBeans* specification [178] defines coding conventions that support the development of such generic algorithms. The main purpose of the JavaBeans specification is to provide a uniform interface for reusable classes (precisely, components), that can be analyzed and mutually connected with generic tools, such as visual GUI builders. A JavaBeans instance (a *bean*) provides metadata about its features, which are the attributes describing its state, the events it fires, and the methods it offers for other components to call. The attributes, which are called *properties*, can be either of a primitive type, references to other objects, or arrays of these types (the *indexed properties*). A JavaBeans class exposes its list of properties by declaring methods that follow pre-defined naming conventions or *design patterns*. For example, the following two *getter* and *setter* methods declare that the class has a property called "age" of type `int`, which can be read and written.

```
public int getAge()             // e.g.  { return age; }
public void setAge(int value)   // e.g.  { age = value; }
```

Note that reflection can be used to look up which getter and setter methods exist in a JavaBeans class, and to derive the class properties from this information. Properties are called *bound* if they fire a `PropertyChangeEvent`, whenever their value has changed, i.e. in their setter method. This follows the Observer design pattern from subsection 3.1.2, insofar

that objects that wish to be notified about changes can register themselves as observers (or *listeners* in the Java terminology) of the target bean or one of its properties. Thus, beans do not only provide static metadata, but can also pro-actively notify their environment about changes.

JavaBeans is particularly suitable for the development of data structures that can be edited with generic instance editing tools, because it provides explicit support for customizing the visual appearance of objects and properties by so-called `Customizer` and `PropertyEditor` objects. Figure 6.2 shows the use of JavaBeans components in a visual GUI builder. Here, the component class `JButton` specifies a `foreground` property and provides a `PropertyEditor` that displays a combobox in which the button's foreground color can be selected conveniently.

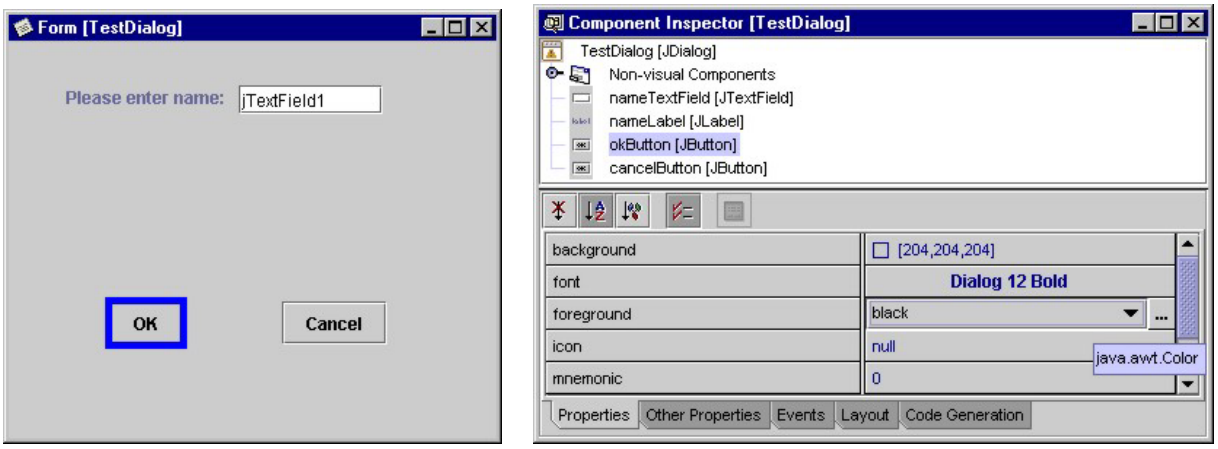

Figure 6.2.: The use of JavaBeans components in a visual GUI editor (here, Forte for Java [180]). The properties of the form elements selected in the left window can be inspected and edited in the customizer on the right.

This attractive metadata support makes JavaBeans a candidate implementation platform for semantically rich data structures that can be processed by generic classes and tools. However, the types of metadata provided by the JavaBeans standard are quite limited. There is little information provided apart from the names, types and access rights of the properties. In the following sections I will show that other useful metadata – especially constraints – can (and should) be supplied by Java classes, by implementing facets with design patterns very similar to those from JavaBeans.

6.2. KBeans: Adding Facets to JavaBeans

This section presents the KBeans approach to define and evaluate facets that declare metadata about JavaBeans properties. An example based on figure 5.4 (page 66) will help to convey the main idea. In order to declare the constraints that the (inclusive) minimum

age of a person is 0 and that a person's spouse may not be one of his or her children, we add the following members to the `Person` class.

```
public final static int ageMinInclusive = 0;
public Person[] getSpouseInvalidValues() { return children; }
```

As illustrated in figure 6.3, these fields and methods can be detected and evaluated by a reflection-based facet engine.

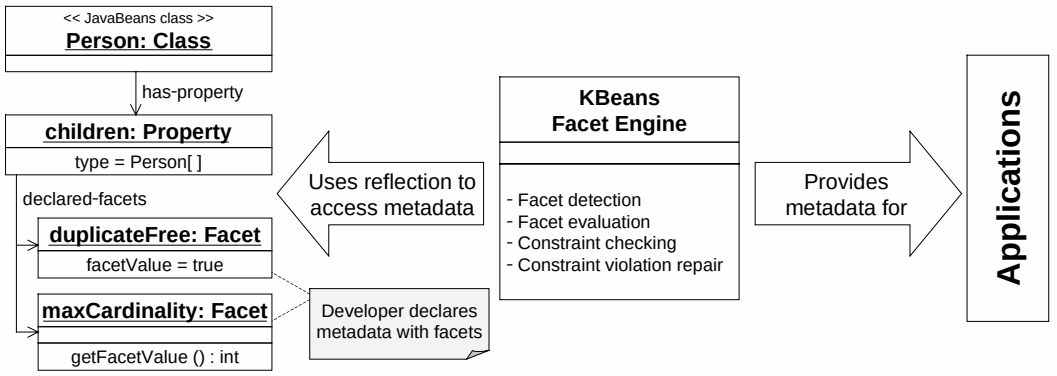

Figure 6.3.: The main idea of KBeans is to attach facets to JavaBeans properties, which can be accessed by means of reflection.

The following semi-formal definitions will lay an ontological foundation for the rest of this section. Here, *property* is a property of a given JavaBean *bean* that is decorated with a facet.

facet: (*bean* × *property* × *facet type*) → *facet value*

facet value: A primitive Java value, an `Object`, or `null`.

facet type: *facet name* → *type mapping*

facet name: A Java identifier which is unique among the facet types that are installed in the Java Virtual Machine.

property type: The `Class` of *property*.

type mapping: *property type* → *facet value type*

facet value type: The `Class` of the facet value or `null`.

constraint type: A *facet type* with a *validation method*.

validation method: (*value* × *facet value*) → `boolean`

In other words, a *facet* declares one Java value for a given property of a given bean. If this value is `null`, then the facet is treated as *not* declared. The class and semantics of the facet value are defined by the *facet type*. Each facet type has a unique name and a *type*

mapping function, that derives the type of the facet value, given the type of the property. The result of the type mapping can be `null` to indicate that the facet type cannot be applied to the given property type. A *constraint type* is a facet type with a *validation method* which can check whether a given value would be a valid property value, given the bean's current facet value.

In order to add facets to a given JavaBeans class, KBeans supports two types of facet declarations:

- *Dynamic facet declarations* are methods that match the signature defined in subsection 6.2.1. These methods compute facet values dynamically, i.e. they may return different results for every bean with every method invokation.
- *Static facet declarations* are fields that match the signature defined in subsection 6.2.2. These fields store a constant facet value for all beans of the class.

6.2.1. Dynamic Facet Declarations

Let's assume we wish to declare a facet of the facet type with the name *facetName* for a property called *propertyName* of the `Class C`. The facet is declared dynamically, if `C` has a method of the following signature

```
public <facetValueType> get<PropertyName><FacetName>()
```

where *facetValueType* is the result of the facet's type mapping for the given property type, and both *propertyName* and *facetName* are capitalized in conformity with the JavaBeans naming rules. The method's prefix can be `is` instead of `get`, if the *facetValueType* is `boolean`. The current facet value is defined by the result of the method call. In the example from the beginning of this section, the property name is `spouse`, the facet name is `invalidValues` and the facet value type is `Person[]`, because the type of `spouse` is `Person`. The facet value is the array of the values of the `children` attribute.

Since the method signature above is compatible to the naming conventions of JavaBeans *getter* methods [178], the facets themselves can be accessed and formally defined as JavaBeans properties:

> **Definition (Dynamic Facet Declaration).** A *dynamic facet declaration* for a property is a readable JavaBeans property with the facet's value type and the name `propertyName + capitalize(facetName)`.

The fact that facets for a property are themselves properties has several consequences. We can read and (if a *setter* method is supplied) write the facet values just like other property values. This includes that we can use existing bean-based tools and methods such as GUI builders and persistence mechanisms to modify them. Furthermore, we can recursively define meta facets that constrain other facet values.

Another interesting issue with facets declared as methods is *inheritance*. Approaches such as *Design-by-Contract* [125] define that class invariants must be inherited from superclasses, so that subclasses can only make the invariant more restrictive. In the case of constraining KBeans facets, this means that a method overloading an inherited facet method must return a value at least as restrictive as the result of the `super` methods. If we funneled all facet access through specific utility classes (such as the `FacetIntrospector` described in subsection 6.2.4), we could implement a mechanism that automatically invokes the super methods and delivers a list of facet values for a given facet type. However, in conformity with the usual semantics of inheritance in Java, where programmers can freely modify an object's behavior by overloading, it appears to be better to put the responsibility for overloading facets into the hands of the programmer. In those cases where overloading is strictly not permitted, developers can declare the facet declaring methods as `final`.

It is important to note that facet methods – similar to *getter* methods in general – must be free of side-effects. In particular, they must not modify any variable beyond their local scope. The reason for this is that the facet methods can not make any assumptions about when they are invoked.

The extreme flexibility of dynamic facet declarations, which is due to their potential to perform arbitrary computations within their method bodies, is at the same time a blessing and a curse. Facet access is expensive, because the methods have to be called each time a facet is being evaluated, and for every object individually. The methods do not provide any information about whether their return value has changed since the last call. Furthermore – and in many cases most importantly – their facet value is the result of a "black box" procedure that reduces its semantic transparency. It is impossible to reason or make any assumptions about the facet value itself at run-time. This also makes it barely possible to export the semantics to other formats such as XML Schema and OKBC. Covering the case that such a stronger notion of semantic transparency is needed, the following subsection introduces the concept of static facet declarations.

6.2.2. Static Facet Declarations

Let's (again) assume we wish to declare a facet of the facet type with the name *facetName* for a property called *propertyName* of the `Class C`. The facet is declared statically, if `C` has a field of the following signature

```
public final static <facetValueType> <propertyName><FacetName>
```

A rather executable definition follows.

Definition (Static Facet Declaration). Let `field` be the result of the call of `getField(propertyName + capitalize(facetName))` for the `Class` that owns a property. `field` is a *static facet declaration* for the property, if it is `public`, `final`, and `static` and has the facet's value type.

The (constant) value of this field is regarded as the facet value for all instances of the class. Similar to their dynamic counterparts, static constraint declarations that are "overloaded" by a namesake field in a subclass should ensure that the overloaded values are more restrictive than the values of the `super` fields.

6.2.3. Coding Conventions

Beside having a meaning in KBeans, both dynamic and static facet declarations can also be treated as normal fields and methods. Thus, the distinction between "normal" fields and methods and those that declare facets is rather blurred. However, semantic transparency requires that other developers and programs are supplied with clear, unambiguous facet declarations. Otherwise, programmers might not recognize existing facet declarations or produce incorrect declarations by accident due to misspelling methods. A solution to these problems could be to extend the Java specification by a new modifier keyword (e.g., `facet`) that would make the declaration of facets explicit. Since such an extension will not be available in the near future, we need to cope with an intermediate compromise. Most of the problems described above can be reduced by an automated facet declaration checker, which uses reflection to detect apparent facet declarations that are misspelled or have illegal value types.

Apart from this tool, the following coding conventions can make facet declarations more readable and reduce oversights. The facet names should convey their semantics and make their role as a facet declaration clear. For example, the purpose of a method called `getLastNameMaxLength` is clearly accessible to readers of both source code and corresponding UML diagrams. Since all facet declarations contain a concatenation of property and facet names, they are easy to relate to their role. In order to make facets even more apparent, the names of the facet types could include predefined keywords, e.g. end with `_FACET`.

On source code level, it is common practice to conglomerate methods that cover similar aspects. In order to enable programmers to quickly browse a JavaBeans property and its facets, I recommend to place getter and setter methods and the facet declarations above each other. The facet values should be precisely described with a commentary block, so that the semantics are transparent to developers from the automatically generated JavaDoc API documentations alone. Using the predefined `@see` tag of JavaDoc, properties should provide cross-references to the facets that are defined on them, as shown in the method definition from figure 6.4. This also allows to automatically detect oversights, because JavaDoc will report misspelled facet declarations.

A technical weakness of the KBeans approach is that programmers could choose to add both static and dynamic facet declarations to the same class. Like facet overloading, such situations will normally occur infrequently, but they are technically possible. Our compromise here is to define that if both types are detected for a class, then the dynamic declaration has priority over the static one.

```
/**
 * Sets the values of the children property.
 * @param values the new children
 * @see #getChildrenMaxCardinality
 * @see #isChildrenDuplicateFree
 * @see #isChildrenOrdered
 */
public void setChildren(Person[] values) { ... }
```

Figure 6.4.: JavaDoc comments can be used to document constraint declarations.

6.2.4. Accessing and Processing Facets

Figure 6.5 displays a UML diagram of Java classes that can be used to access and process facets efficiently. An implementation of this *facet engine* is available from the KBeans Homepage [51].

Facet types are defined by means of classes that implement the `FacetType` interface. Each `FacetType` has methods that can be used to check whether a facet declaration of the given type is admissible for a given property type and to compute the type mapping. If a facet type class also implements the `ConstraintType` interface, it must define a method to check whether a given value would satisfy the semantics behind the constraint. For example, I have defined a class `MaxCardinalityConstraintType`, which implements `ConstraintType` and which can check whether a given indexed property does not have a greater cardinality (array length) than the facet value. This class furthermore implements the `RepairableConstraintType` interface to provide a "repair" method that derives a valid value from an invalid one, by removing those array positions that exceed the maximum cardinality. The source code of the `MaxCardinalityConstraintType` class can be found in Appendix C.

The `FacetType` classes must be centrally registered in the `FacetTypeRegistry`, which maintains a list of facet types so that reflection-based classes can find out what to look for. KBeans supports a number of core facet types (see section 6.3), which are installed by default. However, applications are free to add individual new facet types, preferably in static initializer blocks that are automatically executed when the JavaBeans class that uses the facet type is accessed for the first time.

The main functionality of the facet engine is accessible through the `FacetIntrospector`. This static utility class uses reflection to detect the fields and methods that match the patterns of the installed static and dynamic facet types. The `FacetIntrospector` associates the JavaBeans `PropertyDescriptors` with `FacetDescriptors`, which have links to the respective facet fields and methods and thus store the result of the reflection process for future access.

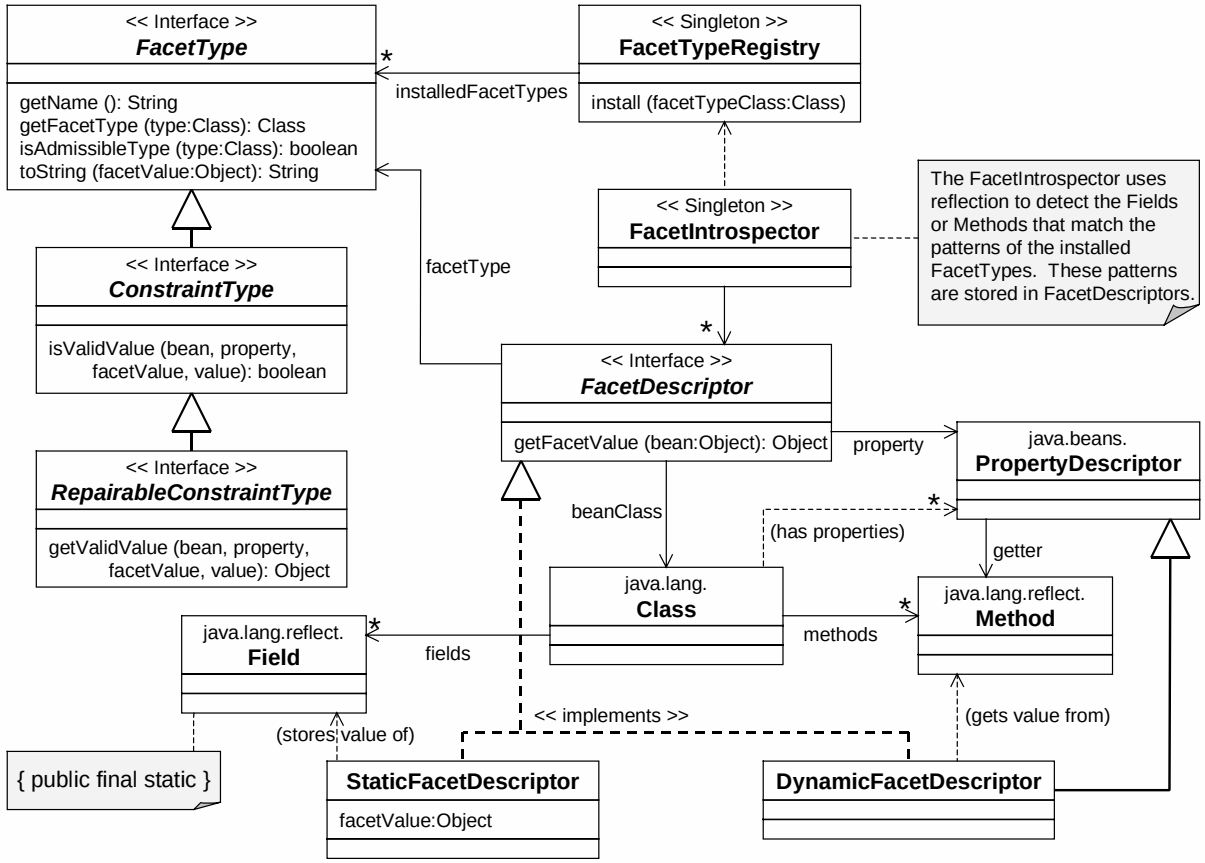

Figure 6.5.: The core classes of the KBeans facet engine.

6.3. A Catalog of KBeans Facets

This section presents the catalog of the standard facet types installed in the `FacetTypeRegistry` of KBeans. The goal of this catalog is to cover the most frequently needed types of metadata to support semantic transparency, while remaining as small and simple as possible. The most frequently needed facet types have been derived from the ontology exchange standard OKBC (table 4.1 (page 53)) and from *XML Schema* [200], an XML-based specification language for describing the syntax and semantics of a class of XML documents. Being an official standard recommendation of the World Wide Web Consortium, XML Schema is rapidly gaining acceptance as a standard for document exchange on the internet. Details on the facet types of XML Schema can be found in Appendix D.

Since XML Schema and OKBC have similar goals, because they both stem from domains that deal with communicating data and knowledge bases between distributed systems, I decided to include all of their standard constraint types into KBeans, apart from a few exceptions which do not make sense in the context of Java. The inclusion of these standards allows to use the KBeans approach to smoothly integrate semantically rich data

structures from other languages into Java programs. In this context, *data binding* [154] involves compiling (XML) schemas or ontologies into Java classes, which handle the translation between external documents and objects. Existing data binding frameworks for XML Schema (e.g., *Enhydra Zeus* [48] and *Castor* [28]) generate Java source code – in particular, JavaBeans classes. The static facet declarations from KBeans can be used to improve data binding and even Round-Trip Engineering of ontologies [103], XML Schema files and JavaBeans classes.

The KBeans constraint types furthermore embrace some elements of UML class diagrams [21], such as cardinalities and inverse relationships. These elements are seldom stated explicitly in Java code, so that reverse engineering from Java to UML is complicated.

A summary of the KBeans facet catalog is shown in table 6.1. A detailed specification of the single facet types can be found in Appendix A. The references to Java types in the table and the rest of this document are as follows.

`<type>`: *any* type

`<simple>`: any non-indexed type

type`[]`: the indexed variant of *type*

`<primitive>`: any primitive type, or `String`

`<object>`: any non-primitive type, except `String`

`<numeric>`: {`byte`, `double`, `float`, `int`, `long`, `short`}, or the respective wrapper types (`Integer`, etc.)

`<comparable>`: a `<numeric>`, or a class implementing `java.lang.Comparable`

The facet names follow the usual naming conventions of Java, so that some XML Schema and OKBC constraint types have been renamed. The three XML Schema facets not implemented as KBeans standards are `whitespace` (which is only relevant for XML parsing), `totalDigits` (which can be expressed in terms of `maxLength` and `maxExclusive`), and `length` (which is the conjunction of `minLength` and `maxLength`). Similarly, the OKBC facet `:CARDINALITY` is not implemented in KBeans, since it is simply an abbreviation of `:MIN-CARDINALITY` and `:MAX-CARDINALITY`. Note that it is possible to dynamically add these facet types to KBeans, by adding appropriate `FacetType` classes to the `FacetRegistry` from figure 6.5.

6.4. Application Scenarios

KBeans has been successfully applied in several development projects [108, 105, 160], in particular from the domains of knowledge-based systems and multi-agent systems. Although its initial intention was to support the implementation of ontologies in Java, it

Table 6.1.: The standard facet types of KBeans.

Facet Type	OKBC equivalent	XML Schema equivalent	Property types	Facet value type
defaultValue	-	-	<type>	<type>
maxLength	-	maxLength/totalDigits	<type>	int (≥0)
minLength	-	minLength	<type>	int (≥0)
fractionDigits	-	fractionDigits	float*, double*	int (≥0)
maxCardinality	:MAXIMUM-CARDINALITY	maxLength/maxOccurs	<simple>[]	int (≥0)
minCardinality	:MINIMUM-CARDINALITY	minLength/minOccurs	<simple>[]	int (≥0)
validClasses	:VALUE-TYPE	-	<object>*	Class[]
invalidClasses	-	-	<object>*	Class[]
maxInclusive	:NUMERIC-MAXIMUM	maxInclusive	<comparable>*	<comparable>
minInclusive	:NUMERIC-MINIMUM	minInclusive	<comparable>*	<comparable>
maxExclusive	-	maxExclusive	<comparable>*	<comparable>
minExclusive	-	minExclusive	<comparable>*	<comparable>
validValues	:SOME-VALUES	enumeration	<type>	<simple>[]
requiredValues	-	-	<simple>[]	<simple>[]
invalidValues	-	-	<type>	<simple>[]
notNull	-	-	<type>	boolean
inverseProperty	:INVERSE	-	<object>*	String
equalProperty	:SAME-VALUES	-	<type>	String[]
unequalProperty	:NOT-SAME-VALUES	-	<type>	String[]
validValuesProperty	:SUBSET-OF-VALUES	-	<type>	String[]
requiredValuesProperty	-	-	<type>[]	String[]
invalidValuesProperty	-	-	<type>	String[]
pattern	-	pattern	<type>	String
ordered	-	-	<comparable>[]	boolean
duplicateFree	:COLLECTION-TYPE	-	<type>[]	boolean
validValuesNames	-	-	<type>	String[]

*Facet type is applicable to both simple and indexed type.

has shown to be very useful in other domains as well. With the metadata provided by KBeans classes, machines and humans are able to *prevent*, *detect*, and eventually *repair* object states that violate the semantics intended by their developers. This section will demonstrate how KBeans technology can be used for these purposes.

6.4.1. Implementation of Semantically Rich Domain Models

Any non-trivial software system will contain a model of the problem domain. For example, a database system might comprise the domain model about persons and family relationships from figure 5.4 (page 66). According to the Model-View-Control architecture (figure 3.1 (page 25)), such domain models should be implemented by non-visual model classes. The container classes typically employed for this in Java are structured like JavaBeans classes, i.e. with little apart from getter and setter methods for each attribute. However, the model's underlying assumption, which are frequently added as constraints or comments to UML class diagrams, are typically not implemented transparently. These assumptions are important, because they constrain the space of valid application scenarios, and thus allow to use and reuse models safely and efficiently. However, when domain models are implemented in Java, their original semantics are usually obscured behind

non-transparent programming primitives. The constraint that a person's age must not be negative might be implemented by `if`-clauses that reject negative values before they are assigned to the `age` attribute (e.g., in the `setAge` method). The constraint that a person's marital status must be one of three predefined constants might be implemented by three appropriate radio buttons. Thus, the semantics are typically implemented *implicitly*, especially by making sure that any execution path through the program will result in valid object states. The complexity of potential execution paths often leads to systems that are error-prone, hard to understand and maintain, and almost impossible to reverse engineer. A KBeans implementation of domain models, such as the `Person` class (which can be found in Appendix B), makes the model's semantics explicit.

6.4.2. Ontology Implementation

KBeans can be used for the implementation of frame-based OKBC ontologies by mapping ontology concepts to JavaBeans classes, slots to properties, and OKBC facets to (static) KBeans facets. However, KBeans is less expressive than OKBC. In particular, KBeans does not support the following:

- Multiple inheritance is not directly supported, because concepts are implemented by Java classes. Java (and C#) classes have exactly one superclass. A certain degree of multiple inheritance can be achieved by representing concepts with Java *interfaces* (abstract classes without method bodies), because each Java class can implement any number of interfaces. Interfaces can even contain static fields, so that static facet declarations are possible.

- Slots are not treated as first-class instances, i.e. slots can neither be put into an inheritance hierarchy, nor enriched by slots themselves. Both aspects can be easily simulated by declaring intermediate *concepts* that represent slots. Note that it is possible to attach facets to slots, and – by means of dynamic facet declarations – facets to facets.

Compared to purely declarative ontology languages, KBeans ontologies and domain models have a much greater flexibility because of the imperative features (methods) and a huge amount of available code libraries that can be directly attached to the semantic objects. For instance, it is possible to add reasoning code of Problem-Solving Methods directly to the method's input and output ontologies. Fensel et al. [56] describe an approach for implementing Problem-Solvers, which are specified in the formal language UPML, in Java (cf. subsection 4.3.2). Similar to our own previous work [103], they even propose to map ontologies to an ordinary class hierarchy (in a JavaBeans style). KBeans facets can be used to specify the semantics of these classes. Furthermore, KBeans can be used to integrate interface code to clinical devices and other data sources much more naturally than if ontology classes from external files were used. It is also possible to implement side effects of method calls (e.g., to update private attributes that depend on other attributes), to compute values like facets dynamically, etc. Despite this mix of implementation details

and structural ontological information, the semantic transparency of the ontologies is not obscured. In particular, it is possible to re-extract an ontology from KBeans classes, in support of model reuse.

The major advantage of implementing ontologies with KBeans is *compatibility* to existing object-oriented standards, especially when the rest of the application is written in Java as well. This allows to implement smooth transitions between the various modeling artifacts. Developers can *travel light* in the sense of Extreme Programming, i.e. they can keep all structural information about the ontologies in Java classes. This allows to implement the interactions between ontologies and other modules very smoothly. Programmers can use the language they are used to and have all information in one and the same model, increasing development speed and reducing code duplication. Furthermore, many powerful tools and auxiliary classes can be used for ontology development. All of the currently most widely used visual Java development tools have explicit support for JavaBeans, so that properties can be added, visualized, and edited conveniently without writing source code. Other generic classes for JavaBeans exist, such as various JavaBeans-to-Database mapping tools (e.g., [179]) and XML-based persistence mechanisms like the JavaBeans Archiver package [182].

6.4.3. Ontology Sharing and Mapping

The persistence mechanisms described above are a central building block for the implementation of ontology sharing and mapping between systems and components. As illustrated in figure 6.6, KBeans ontologies can be translated from a source model (e.g., a domain model) to a target model (e.g., a Problem-Solving Method's input ontology). First, the JavaBeans objects from the source structure are serialized to the JavaBeans XML format [182] by means of the `java.beans.XMLEncoder` class. Then, the XML stream is translated with a declaration of mapping rules in the XML schema translation language XSL [199]. Finally, the resulting XML stream is transformed into JavaBeans objects with the `java.beans.XMLDecoder` class.

KBeans ontologies can also be used in multi-agent systems. Here, an agreement on the semantics and constraints on the possible values of the ontological concepts is necessary to enable inter-agent communication. An agent receiving invalid messages can either reject them or try to use metadata to "repair" them. I have implemented an efficient messaging approach based on the standard agent platform FIPA-OS [60]. Here, agents share ontologies by using common KBeans classes. The single (distributed) agents can have individual implementations of these classes, i.e. they can add own methods or attributes, but agree on common constraints. (This can be implemented by pushing all the property access and facet code into abstract superclasses that can be overloaded with local classes.) Agents communicate by exchanging (FIPA) messages which contain JavaBeans objects that are converted to XML streams. On message receipt, the agents are able to validate the beans with the KBeans constraints. I will return to this issue in section 8.2.

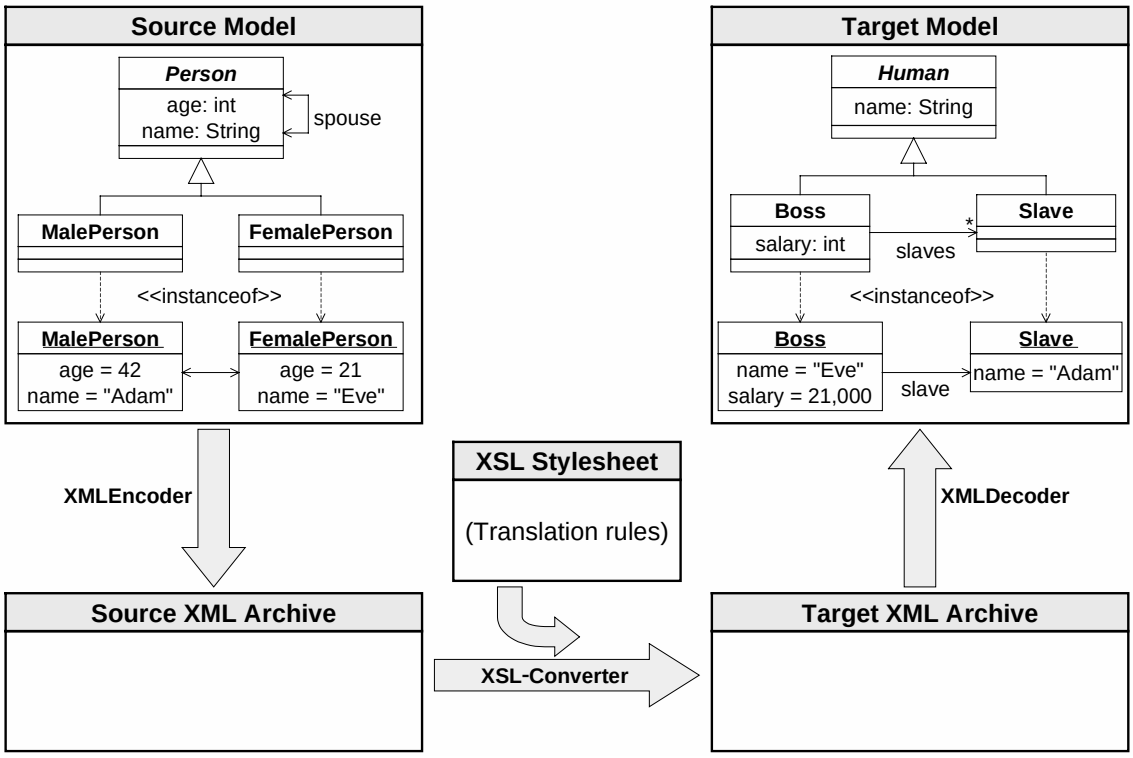

Figure 6.6.: JavaBeans objects can be translated from a source model into a target model, using an XSL-T stylesheet.

6.4.4. Constraint Checking

In the context of modern development processes, in particular the aforementioned lightweight methodologies, self-testing units are an essential means of producing quality software rapidly. The KBeans constraint types can be employed to monitor a given structure of JavaBeans instances to detect constraint violations at run-time. For that purpose, KBeans provides a `ConstraintManager` class, which can be plugged into an existing model of JavaBeans instances, or de-activated, when the program is no longer in development stage.

The `ConstraintManager` checks constraints whenever the value of any property in a given data structure has changed. Here, the fact that each change of a bound JavaBeans property produces a `PropertyChangeEvent` is used for two purposes. First, the `ConstraintManager` only needs a reference to a "root" object, from where it traces the links to all other objects that are accessible from it. The `ConstraintManager` registers itself as a `PropertyChangeListener` on these objects and is thus able to autonomously register or unregister as a listener of the objects that are added to or removed from the data structure. Second, any property change can cause an execution of the constraint checking procedure. This is simply an invokation of the `isValidValue` methods of the `ConstraintTypes` that have been detected by the `FacetIntrospector`.

In the context of knowledge-based systems, the constraint checking mechanism is particularly useful during knowledge acquisition, to test the correctness of knowledge models. Details on this will be elaborated in chapter 7.

The `ConstraintManager` is quite efficient for the evaluation of static constraints, because only those constraints that directly concern the property that has changed need to be checked. However, it can become quite inefficient for dynamic constraints, because the method bodies of dynamic constraint declarations can contain hidden dependencies, so that on each change of a property, all dynamic constraints need to be evaluated. This inefficiency will usually not slow down knowledge acquisition, where property changes are made one-by-one and manually through a user interface.

6.4.5. Semantic Annotation of Reusable Components

The basic idea of component-based development is to build applications by plugging together reusable building blocks (components), which encapsulate their state and expose their services through well-defined interfaces. However, the presence of externally developed components within a system introduces new challenges for software engineering activities, particularly due to hidden dependencies among components, reduced testability, and difficulties in program understanding [141]. The main reason for these problems is the lack of information about components that are not internally developed [141]. Frequently, developers have implemented semantical assumptions that are neither transparent to other developers nor to the software environment in which the components are embedded in. As a result, object-oriented classes (including components) are frequently instantiated in ways that violate these assumptions. KBeans facets can be used to specify the intended semantics and admissible application scenarios to humans and machines.

Graphical widgets such as Java's Swing classes are a very popular type of components. As already shown in figure 6.2, comfortable off-the-shelf visual modeling tools allow to combine simple widgets like buttons and text fields to dialogs and other complex components. Their visual appearance and behavior can be easily modified by changing the components' properties. Visual modeling tools analyze the components' interface, detect the properties and their types, and provide respective editors generically. For example, the `JButton` class from the Swing library has a `verticalTextPosition` property, which specifies whether the button's text shall be aligned to the left, the right, or the center. However, although the `verticalTextPosition` property is of type `int`, only three values are valid for it (namely the predefined constants `LEFT`, `RIGHT`, and `CENTER`). Without metadata in the style of KBeans, generic editors are unable to reject invalid values which counteract the component builder's intended semantics. Adding facets like `defaultValue`, `validValues`, and `validValuesNames` to the `verticalTextPosition` property would significantly improve semantic transparency.

The metadata provided by KBeans facets can be employed by generic user interface components. These components could have labels or "tool-tip" texts that describe the expected values (e.g., "Amount (10..100):"), or change their background color if the user enters an invalid value. If a finite number of values is defined by a `validValues` facet, then

a combo box or a group of radio-buttons can be used. If a `maxLength` facet is present, then a text field can automatically constrain the number of characters. All these components reduce development overhead, because programmers only need to explicate the object model's semantics, and the corresponding input dialogs or forms can be developed rapidly by visually plugging together generic components. The user interface code could even be generated automatically.

Driving the idea of generic components further, I have developed a tool called `KBeans-Shell` that serves as a graphical editor and knowledge acquisition tool of any structure of JavaBeans instances. Details on this tool will be supplied by chapter 7.

6.5. Discussion

6.5.1. Related Work

Other approaches for the implementation of semantic transparency for Java classes exist. One approach (e.g., [121]) is to include external formats such as XML Schema or OKBC documents into the software architecture and rely on the facet services provided by parsers and components for those languages. However, such a mixture of Java and external files complicates software maintenance and contradicts to the paradigm of Traveling Light. Furthermore, it requires an infrastructure to access the objects from the external files as Java objects. Access to data from XML or ontology documents is traditionally implemented by translating them into an intermediate layer of objects, such as DOM trees. This is very flexible, as it allows to access documents the schemas of which are unknown at build-time. However, it complicates access to the data contained in the documents and means an artificial mixture of modeling levels, because entities that are naturally considered as classes are treated as instances.

The lack of an efficient constraint mechanism in Java has been a widely discussed issue for years. The huge demand for an appropriate Java extension has lead to the addition of a new `assert` keyword in Java 1.4 [181]. Here, an *assertion* is understood to be a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed. `assert` statements are useful to check constraint satisfaction at run-time, especially pre- and post-conditions in the sense of Design-by-Contract. The following code snippet shows how assertions – here in conjunction with KBeans constraints – can be used to check a pre-condition to prevent illegal value assignments.

```
public void setAge(int value) {
    assert FacetIntrospector.isValidValue(this, "age", value);
    this.age = value;
}
```

A major limitation of the `assert` statement is that it does not make explicit what type of contract it supports, e.g. there is no way to declare that a given boolean expression

represents a post-condition. This problem is tackled by other approaches, such as *iContract* [49]. iContract uses a precompiler that converts code from predefined commentary blocks into executable Java code, as in the following example:

```
/**
 * @pre value >= 0
 */
public void setAge(int value) { ... }
```

This approach can increase the semantic transparency of classes and methods to humans, because the expressions from the commentary blocks will also appear in the API documentation. However, the need for an additional translation step and tools is a considerable overhead.

A different approach is taken by *jContractor* [95], where class invariants, and pre- and post-conditions can be specified by means of specific methods that match predefined design patterns very similar to the KBeans approach. For example, the following method declares the pre-condition of the `setAge` method.

```
protected boolean setAge_PreCondition(int value) {
    return value >= 0;
}
```

Calls to these boolean methods are inserted into the byte code by means of special class loaders and object factories.

All of these approaches are limited to constraint checking, i.e. they only evaluate boolean expressions, but do not provide any additional metadata about *what* they constrain. The only way to get more information about the constraints – and thus to increase semantic transparency for machines – would be to parse and analyze the source or byte code, which is in general far too complex to be feasible. Facet-based approaches such as KBeans can provide much more metadata, because they are declarative instead of imperative.

Instead of defining facets in special fields and methods like in KBeans, other approaches rely on run-time objects to represent facets. Okunev's [139] draft of a validation approach for JavaBeans properties makes use of the fact that the `PropertyDescriptors`, which are the result of the JavaBeans reflection process, can be decorated with arbitrary data items (using the `setValue` method). Thus, metadata can be associated with predefined constraint types (e.g., `"maxLength"`). This approach is quite complicating to use, because the authors of constraints need to define additional `BeanInfo` classes, which are hard to read and (reverse) engineer. A similar approach is presented by McLaughlin [121], who employs XML Schema files and data binding to express and read model semantics into Java objects. The constraints are translated into one `Constraint` object for each XML Schema type. The `Constraint` class has methods such as `getMinInclusive` that return the facet values defined in the schema. This approach, however, relies on external schema files, and is thus only an option if the object models are specified in XML Schema. Both approaches are also limited to static facet declarations, whereas dynamic KBeans facets can evaluate anything that can be expressed in Java.

6.5.2. Benefits and Limitations of KBeans

KBeans is a highly pragmatic, simple, and natural extension of JavaBeans. Its simplicity lies in the use of clear design patterns that are easy to apply, comprehend, and evaluate. The fields and methods implementing the patterns are based on pure Java code and do not rely on external formats or declarations. Particularly, the metadata that provide semantic transparency remain with the class they describe, so that component exchange is facilitated. The constraint methods do not have any negative impact on system performance, because they only provide optional metadata.

KBeans is fully compatible to the Java specification, i.e. no changes to the compiler are necessary to use it. Only a small library of core classes is required to implement facet access (e.g., the facet engine from subsection 6.2.4). Since each of the facet declarations is optional, and their omission in a class means to apply Java defaults, any existing JavaBeans data structure is backwards compatible to KBeans. Furthermore, KBeans can be flexibly extended by custom facet types. The space of potential application scenarios of facet metadata is vast.

Applying KBeans technology in projects only requires a minor adaptation of the programming style. Whereas programmers would usually implement the semantics of their models implicitly, they need to get used to making constraints and other metadata explicit. This is not necessarily an overhead, because all of the relevant metadata that can be explicated in facets are usually implemented somewhere in the code anyway. For example, the names and valid values of the `maritalStatus` property from figure 5.4 are typically implemented in the corresponding GUI elements. Thus, enriching components and shared object models by facets is mostly a question of reorganizing the classes. In general, developers who increase semantic transparency by explicitly embedding the intended semantics of their classes into design and implementation artifacts (i.e., UML diagrams and Java code) integrate the goal of component reuse early in the development process (cf. [11]).

KBeans can be supported by tools both at run-time and during development. At run-time, reflection can be used to support visual builders and other generic programming and knowledge acquisition tools. During development and modeling, KBeans facets appear just like other class members, i.e. they can be visualized in UML diagrams, edited with off-the-shelf IDEs, and they can even be engineered in a round-trip style. By the way, the concepts of KBeans are not limited to Java, but can be transferred to other reflective languages such as C#.

The major limitation of KBeans, when faced with the problem of semantic transparency in general, is its applicability to JavaBeans only. JavaBeans must fulfill various attributes, in particular, they must be public, and have a list of public properties. This limitation does not concern ontologies and reusable components, the goal of which is to declare explicit, sharable class structures, that are public by their very nature.

As stated above, any approach to adding semantic transparency to object models is necessarily a compromise between what is available and what is desirable in Java. A weakness of KBeans is that the distinction between facet declarations and other fields and methods is relatively weak. I have proposed tool support and coding conventions which

help to resolve this problem. Another issue is the great proliferation of class members, which can make classes hard to read, understand, and change. A pragmatic approach to reduce the complexity of the resulting UML diagrams is to filter all public, final, static fields. However, my opinion is that facet declarations should be considered as an integral part of any JavaBeans class, just like (other) public method declarations.

6.6. Summary

This chapter has introduced a simple and natural extension of the JavaBeans standard called KBeans. The main purpose of KBeans is to support the development of semantically transparent domain models (ontologies) and components. KBeans achieves this by supporting the declaration and evaluation of facets, which provide metadata about the valid values of JavaBeans properties. As illustrated in figure 6.7, KBeans achieves build-time transparency by defining coding conventions that allow source code parsers to extract ontology declarations from ordinary Java classes, and thus enable Round-Trip Engineering between UML and Java ontologies. Source parsers need to look up methods and fields that match the signatures that are predefined by the coding conventions. The KBeans coding conventions are also used to increase an object's semantic transparency at run-time. Here, Java's reflection capabilities can be employed to find out which fields or attributes and methods are defined for an instance.

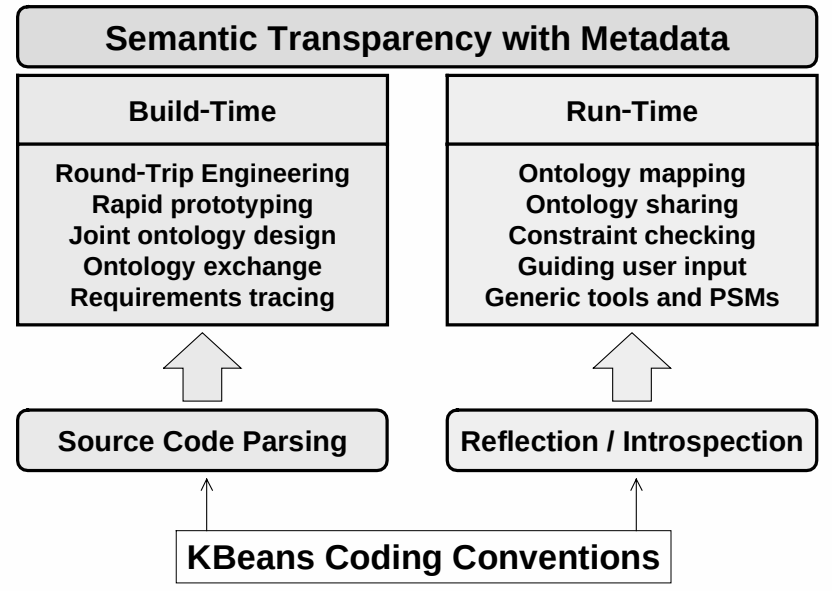

Figure 6.7.: JavaBeans and KBeans technology can be used to increase the semantic transparency of Java classes both at build-time and run-time.

Facets can be used to prevent, detect, and repair invalid object states. KBeans sup-

ports a catalog of facet types, which embraces related standards such as XML Schema and OKBC. From the criteria defined for modeling languages in chapter 2, KBeans therefore scores exceptionally high in compatibility and translatability. KBeans is also adequate and easy to learn, because it relies on Object-Orientation which is equally simple as powerful. The paradigms of encapsulation and the local scope of object definitions supports incremental knowledge acquisition. KBeans is also precise and complete, because Java is a general-purpose language.

I have demonstrated the benefits of KBeans in case studies from the domains of component reuse, ontology sharing, and software testing. The following chapter will introduce a generic knowledge acquisition tool based on KBeans.

7. KBeansShell: A Generic Framework for Knowledge Modeling Tools

After the language aspects from the previous chapter, I will now elaborate on the *tool* dimension of the XP.K methodology. While XP.K and KBeans rely on standard UML tools for ontology editing, specific tool support for the construction of knowledge bases (instances) is required. The Software Engineering world does not focus on instance modeling and although Object Diagrams are an official part of UML, they are barely supported by the current generation of tools. Knowledge Engineering tools like those mentioned in chapter 4 can be employed in conjunction with an Extreme Programming approach, but suffer from relying on specific knowledge acquisition languages which lead to the aforementioned integration problems.

This chapter introduces *KBeansShell*, a Java-based library of generic components that can be efficiently assembled to build customized knowledge acquisition tools (*shells*). In its very basic form, KBeansShell provides a graphical user interface with which domain experts can edit instances of KBeans classes (ontologies) as trees, forms, tables, and graphs. The shell also provides generic services like constraint checking, file management, undo, and clipboard operations. These generic base services and reusable GUI components reduce development costs for tools and thus support rapid evolution of knowledge models in the face of feedback. KBeansShell is therefore particularly geared for agile development processes of knowledge-based systems. Its tight integration of Java-based ontologies enables smooth transitions between the knowledge models and the underlying software infrastructure.

This chapter does not only introduce tool users and developers to the features of the KBeansShell implementation, but also supplies developers with information on which features are commonly found in knowledge acquisition tools and how they can be built in object-oriented languages. Section 7.1 provides an overview of the general architecture of KBeansShell and its implementation as a Java framework. Section 7.2 describes the single reusable components from the module library. Section 7.3 provides practical guidelines how KBeansShell can be used efficiently. Section 7.4 shows how KBeansShell can be used to edit not only instances, but also KBeans *classes*. The chapter is summarized in section 7.5. The application of KBeansShell will be illustrated throughout the sections and also elaborated in chapter 8 in the context of larger case studies. Note that the values, principles, and practices of XP.K do not depend on this specific implementation.

7.1. Architecture

The main goal of KBeansShell is to provide a framework which allows to build knowledge acquisition tools efficiently while remaining open for custom extensions. Efficiency is needed since ontologies might change relatively frequently, so that the overhead of adapting the modeling tool should be minimal. The key to efficient tool development is *reuse*, because it is quite expensive to build comfortable knowledge modeling components from the scratch. Many knowledge acquisition tasks share common structures and reusing standard solutions for these tasks can significantly reduce turn-around times. For example, the instances in many knowledge bases are best navigated with tree-like "explorer" views, whereas the attributes (slots) of the single instances should be edited with forms. KBeansShell provides standard components for these aspects, but enables tool developers to extend or modify the standard solutions where the nature of the domain and the ontology demand for other solutions. The following list enumerates tasks and features that are typically found in knowledge acquisition tools:

- *Model management* includes representing the metamodels (ontologies) and models being edited in the computer's memory and enabling other components of the tool to access and edit the models.
- *Persistence* means storing and retrieving models from a persistent medium.
- *Standard editing operations* on the models are clipboard actions ("copy-and-paste") and undo.
- *Views* display the model elements in suitable ways and accept user input to initiate standard or custom editing operations. Basic types of knowledge editors are forms, trees, graphs, and tables [7].
- *View management* involves displaying and arranging the model views in a layout that suits the users best.
- *Model analysis* assists users in verifying and validating the models they are building.

The KBeansShell architecture provides a simple, light-weight platform into which developers can add *modules*. For the most common tasks, in particular from the above list, KBeansShell provides a library of standard modules. This module-based architecture follows the idea of the generic NetBeans Tools Platform [183], an open framework for Java development tools, in which modules like source code editors, debuggers, and compilers can be integrated. An overview of the KBeansShell architecture is provided in figure 7.1.

As shown in the figure, modeling tools based on KBeansShell technology access knowledge base and ontology through an abstraction layer – called the `BeanModel` – which provides methods to access object properties, class names, and others. This layer serves as a *facade* of the model's implementation details, so that the ontologies do not necessarily have to be implemented as JavaBeans classes. As a result, none of the components of the shell

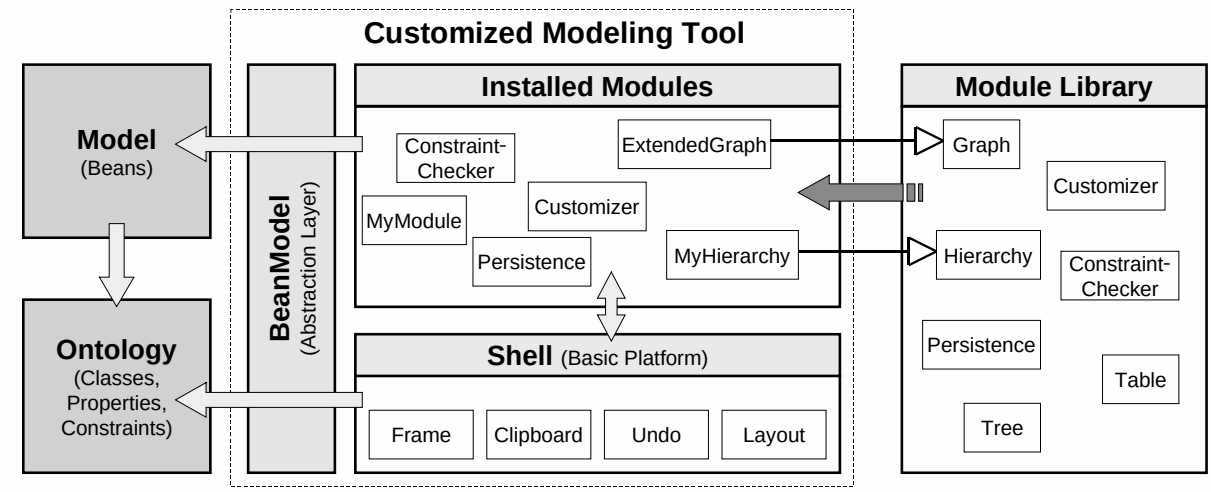

Figure 7.1.: Customizing a knowledge modeling tool by reusing and extending generic modules.

can (or should) make assumptions about the "physical" implementation of the model. In the most common usage scenarios, the shell would implement the model by means of local KBeans objects, but the objects from the model might also originate from an Active Object Model [61] or reside on a remote database with classes defined in an XML schema. The `BeanModel` encapsulates the implementation of the model and can be completely replaced on demand. The `BeanModel`'s interface methods are listed in Appendix E.

Beside the `BeanModel`, the modeling tool consists of a basic *shell*. Think of the shell as a graphical application or frame which contains all other modeling views and manages all types of user interaction with the model. As shown in figure 7.2, the default shell is a top-level window with a menu bar, a tool bar, a desktop area and a status bar.

(A marginal note: In the implementation of the KBeansShell, the `Shell` is a Java *interface* – an abstract class that defines the signatures of the methods that need to be implemented by any shell. Thus, other classes or graphical components than those shown in the figure could implement the `Shell`'s services.)

The `Shell` provides generic clipboard and undo operations on model elements. Additionally, it is the container of model views, which – in the default implementation – are internal windows that can be opened and freely arranged on the shell's main desktop area. Arranging these views is supported by the shell's "intelligent" layout services.

This basic core platform can be extended by installing modules. Modules are Java-based components that are either developed from the scratch, reused as-is from a library of generic modules, or reused and extended from the library by means of object-oriented inheritance.

A rather formal overview of the Java classes and interfaces of `KBeansShell` is provided in figure 7.3. The central interfaces are `Shell`, `BeanModel`, `Module`, and `View`. Each `Shell`

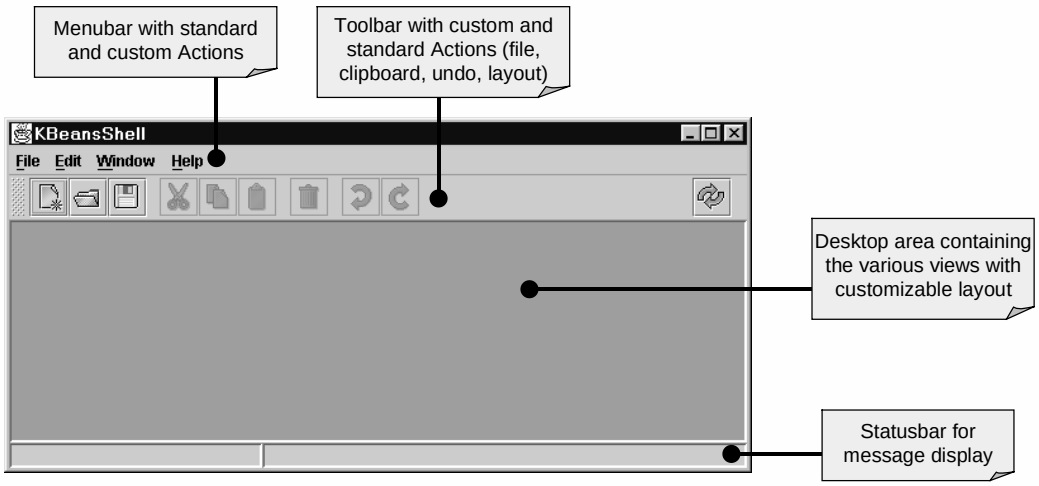

Figure 7.2.: An empty default shell, with only the file persistence module installed.

has exactly one `BeanModel`, which provides access to the ontology and the current model.

`Views` are subclasses of `java.awt.Component`, i.e. visual GUI elements, that can appear in a `Shell`. `Views` typically allow to display or edit the properties of the beans in the model. `BeanViews` are `Views` that allow to select a bean. Among others, this allows the `Shell` to implement generic clipboard mechanisms, which require a certain object to be selected. Figure 5.12 (page 84) shows a sample KBeansShell with various views, such as trees and graphs. Each `Shell` can have several installed `Modules`. `Modules` can be of one of the following types.

- `ViewModules` provide information about a type of `Views`. This includes a method to create a new instance of the `View` and a method that defines whether there are more than one instance of the given `View` type allowed in a single shell. Other methods can specify an icon that can – for example – be used in buttons on the shell's tool bar. `BeanViews` are specialized `Views` that can define a *selected* object or property.

- `ActionModules` provide information about an action. Actions are typically accessible from menus or tool bar buttons. When an `ActionModule` is installed, the shell may generate appropriate menu items or buttons. `BeanActionModules` are specialized `ActionModules` that allow to add actions to specific object types only. The actions that are available for a given object can then be selected by opening a simple pop-up menu in a tree or other `Views`.

- `PersistenceModules` implement a persistence mechanism for model files. Model files may be written in an XML-based format or any other format like OKBC. A `PersistenceModule` must specify a file suffix (e.g., `"*.xml"`) and methods that actually load or save a model. The shell automatically provides appropriate "Open" and "Save" actions to access the installed persistence formats.

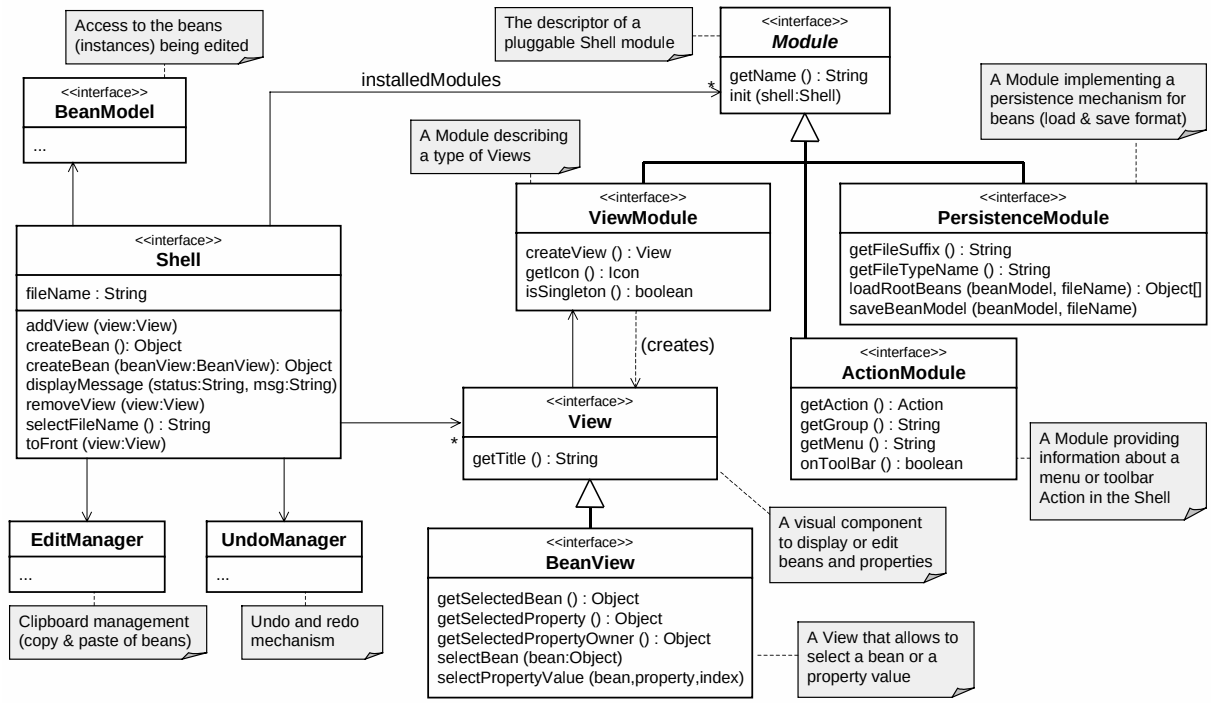

Figure 7.3.: A schematic view of some of the core classes of the KBeansShell library.

KBeansShell provides a collection of *services*, which are described in the following.

Clipboard Operations

Users of modern graphical user interfaces expect to find clipboard actions. KBeansShell provides generic implementations of these services and places global copy-and-paste buttons into the window's toolbar. The shell basically monitors which view has the focus, gets the objects selected by the user, and asks the view to handle the chosen actions. The generic implementations use reflection to access the selected bean's properties. For example, if the user hits the "delete" button while a certain property is selected, it passes `null` to the property's setter method in the selected object.

Undo Mechanism

KBeansShell provides an automated undo mechanism, with which users are able to undo the previous actions and redo them later. These actions are accessible by respective buttons in the shell's toolbar. The implementation of these services makes use of the fact that each change in a KBeans model fires a `PropertyChangeEvent` which encapsulates all information needed to undo an editing step. The event objects particularly store old and new values of the property that has changed. The shell's undo service records all the `PropertyChangeEvents` up to a predefined maximum. Visual GUI elements that allow users to

edit instances thus do not have to worry about implementing their own undo stack. They are, however, able to optimize the generic service by bundling events that logically belong together, so that complex changes can be undone with only a single undo button click.

View Management and Layout

Another global service of the KBeansShell is view management. Users are freely able to open, move, resize, and close the available views of the knowledge model. For each of the installed `ViewModules`, the shell generates corresponding toolbar buttons and menu entries. While this provides much of the flexibility needed during knowledge acquisition, users may typically work with only a selection of the views and arrange them in recurring layout patterns. For that purpose, each KBeansShell has a `ShellLayouter` object and one or more `ShellLayout` instances. Each `ShellLayout` supplies the `ShellLayouter` with information on how the available views should be arranged. All of this information only contains recommendations and relative preferences, which the layouter has to adjust to the actual screen size and number of views. For that purpose, the `ShellLayouter` splits the screen into five regions (`NORTH`, `EAST`, `SOUTH`, `WEST`, and `CENTER`) which can be addressed by the `ShellLayouts`. Each `ShellLayout` provides the following information:

- A list of *default views* which shall be automatically opened when the system starts, and when the user hits the "reset layout" buttons.
- The *relative size* of each region.
- For each type of view a *region*, a *relative size* within the region, and a *priority* which indicates which view shall occupy the center area, if the area is vacant.
- A *fill mode* for each region, which indicates whether the views shall be arranged horizontally, vertically, or stacked (cascaded) within a region.
- A *name* and an *icon*, so that the Shell can create appropriate menu entries and toolbar buttons (e.g., "Activate graph browsing layout").

7.2. Standard Modules

Modules are reusable building blocks of KBeansShell-based tools. This section will introduce the standard modules of the (current) KBeansShell distribution. Each of the modules is introduced with a brief informal description, application scenarios and a sceenshot (if the module is a `ViewModule`).

7.2.1. Persistence Modules

The task of persistence modules is to enable users to open and save knowledge bases in specific file formats.

File Persistence

The `FilePersistenceModule` allows to save object models to the *KBeans file format*, a proprietary language that produces simple, but comprehensible text files. There is an extension of the `FilePersistenceModule`, called `LayoutFilePersistenceModule`, which stores additional `*.layout` files, which specify the layout of the nodes in a BeanGraph (see subsection 7.2.2).

XML Persistence

The `XMLPersistenceModule` exports object models to the XML-based *Java Archiver Format*, the JavaBeans persistence mechanism introduced with the Java 2 platform, Version 1.4 in 2001. This format ensures tool inter-operability, since the leading vendors of Java development tools have committed to it. Furthermore, the XML-based files can be processed further, for example by translating them to other XML languages with the XML transformation language XSL-T.

7.2.2. View Modules

View modules manage a certain type of GUI elements (views), which can be placed onto the shell's desktop area. Table 7.1 provides an overview of the standard view modules from the KBeansShell library.

Table 7.1.: An overview of the standard `View` modules of `KBeansShell`. The last two columns indicate whether the view can visualize primitive properties or relationships between the instances, respectively.

View name	is BeanView	Primitive Properties	Relationships
BeanTree	yes	no	yes
Hierarchy	yes	no	(one property only)
BeanTable	yes	(of selected class only)	no
Tree/Table	yes	(of selected class only)	yes
BeanGraph	yes	no	yes
Customizer	no	yes	(optional)
Constraint Violations	no	no	no

These view classes are completely generic, i.e. they can be used in conjunction with any KBeans ontology. The views make heavy use of object-oriented reflection, which enables them to detect which classes and properties are available and to generate appropriate editors for the current instances. The views furthermore follow the observer pattern, i.e. they register themselves as `PropertyChangeListener` on the instances and update automatically when changes on the model objects occur. This strict separation of model and view

allows to use multiple of the views in parallel without having to hard-code dependencies between them.

BeanTree

The `BeanTree` (figure 7.4) is a hierarchical view that displays objects and their relationships. The objects are represented as tree nodes. A node has one child node for each of the object's reference properties. If the property is indexed, the child node is labeled after the property and has the property values as children. Otherwise, the child node shows the property name and the object being referred to (or a dash, if the reference is `null`).

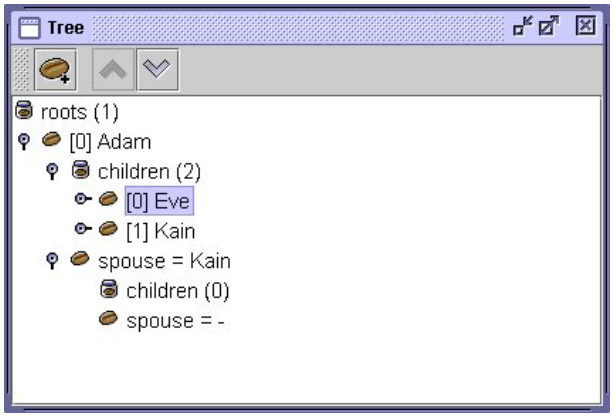

Figure 7.4.: The `BeanTree` view displays objects as tree nodes and their reference properties as child nodes.

Above the tree, three buttons represent standard actions for the selected tree node. The "New object" button allows to insert a new node (i.e., object) at the selected position. The other two buttons allow to change the ordering of values within a reference property.

The `BeanTree` is a very suitable view for object models that are rather hierarchical. It allows to browse and select objects quickly and is best used in conjunction with a `Customizer` (see below), which allows to edit the properties of the selected object. The tree is easy to learn and standard clipboard operations can be intuitively applied to it.

The main weakness of the tree – if used alone – is that it is impossible to add references between existing objects with it. Although it would be technically feasible to implement a "select and assign" functionality to solve this problem, other editing options such as graphs or the tree/table combination from below are much more intuitive. Furthermore the tree might confuse users as it allows to display identical objects more than once and to browse into an infinite depth.

The appearance of `BeanTrees` in a shell can be customized. First, the icons appearing beside each of the tree nodes can be modified. For that purpose, the `BeanTree` analyzes the `BeanInfo` supplied by the object's JavaBeans class and checks whether a custom icon

is supplied. Second, the `BeanTree` can be subclassed, so that, for example, new buttons can be added to its toolbar or specific root nodes can be defined.

Hierarchy

The `Hierarchy` (figure 7.5) is a `BeanView` that displays a tree of objects of a single class. The child node relationship in this tree reflects the values of one explicitly specified reference property. For example, a `Hierarchy` can display the `children` property between `Person` objects. In this case, the tree will only contain the child objects, and – in contrast to the `BeanTree` – ignore any other relationship between `Persons` like `spouse`.

The fact that a `Hierarchy` operates on a specified property of a given class means that it is necessary to parameterize it. Thus, there is no ready-to-use `Hiearchy Module`, but an abstract template implementation of `Module` for which the class and property parameters need to be overloaded by subclasses.

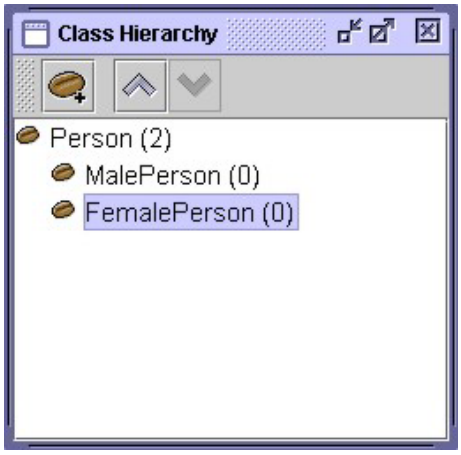

Figure 7.5.: The `Hierarchy` displays a specified reference property by means of tree of objects. Here, the `subClasses` property of Java classes is shown.

BeanTable

The `BeanTableView` (figure 7.6) displays the objects from a knowledge base in a table with one row for each object. A combo-box can be used to select the types of objects that shall appear in the table. If no type has been selected (default), then the table contains only two columns, which display the objects' classes and names. If a certain type has been selected, additional columns show the property values. The objects in the table can be sorted by columns.

The table is well suited to get a complete overview of the objects in the model and to identify which objects exist. However, it is complicating to trace references between the objects, so that the table is incomplete for object editing.

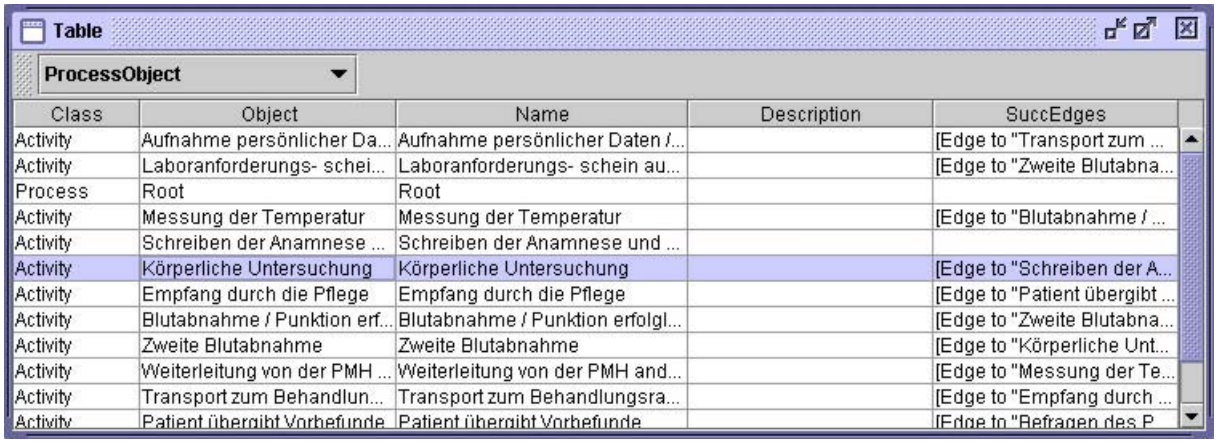

Figure 7.6.: The `BeanTableView` displays all objects in the model in a "flat" table.

The `BeanTableView` is also used in a module called `BeanSelectionDialog` (figure 7.7), which prompts the user to select an object from the current knowledge base. This is a common task in user interfaces. Using the `BeanSelectionDialog` is very simple, as shown in the following code snippet, which displays the dialog from figure 7.7) and assigns the chosen object to the variable `o`.

```
Object o = new NewBeanSelectionDialog(
                    "Select a Medium or create a new one",
                    getShell(),
                    Medium.class
                ).go();
```

Tree/Table

The `TreeTableView` (figure 7.8) meets the recurring demand for a component in which objects can be linked to each other, for example when a `spouse` shall be assigned to a given `Person`. `TreeTableViews` allow to browse the existing objects in a `BeanTree` while showing the potential valid values for the selected node in an associated `BeanTable`. The table can be filtered to display only those values that do not violate any of the constraints defined for the chosen node.

BeanGraph

Graphs are a popular means of visualizing entities and their relationships. The `BeanGraphView` (figure 7.9) is a view that allows to display and edit instances as nodes and their relationships as edges. `BeanGraphViews` provide commands to display a compact overview of the graph, to search for specific nodes, to zoom in and out, and to arrange the nodes by means of various layout algorithms.

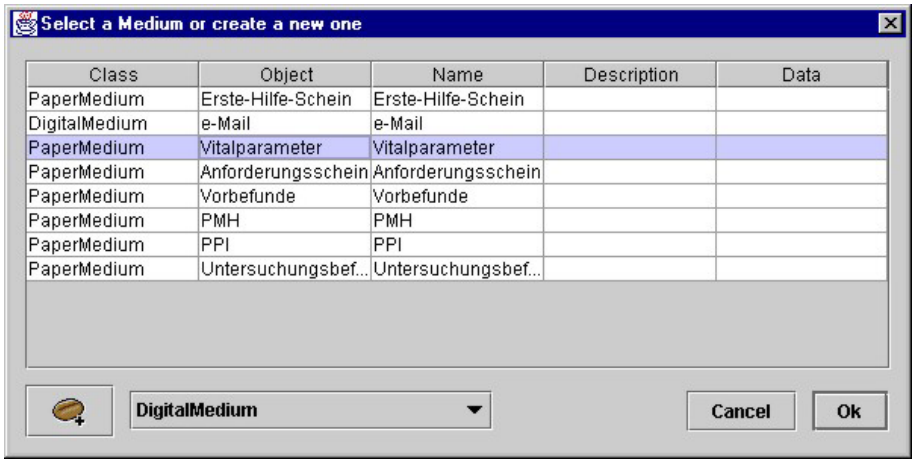

Figure 7.7.: The `BeanSelectionDialog` is a simple dialog that allows to select one object from the current knowledge base, or – optionally – to create a new instance of a given class.

The default implementation of the graph visualizes all instances from the knowledge base by uniform rectangles. This is useful for general overviews and browsing, but the graphs frequently become unintelligible. Therefore, the `BeanGraphView` has an comprehensive adaptation interface. First, the graph classes can be adapted by subclassing them. The graph package that has been developed as a byproduct of the KBeansShell is very modular and has been used in various other projects at the FAW as well (e.g., POWM [158]). Second, ontology classes can supply metadata about how they wish to be displayed. The `BeanGraphView` uses reflection to check whether a `BeanNodeComponent` class is defined for a given ontology class, which defines the component that shall represent an instance of the class. For example, the "female" images in the graph from figure 5.12 (page 84) have been implemented by defining a `FemalePersonNodeComponent` class. Third, the `BeanGraphView` can be customized by means of a complex mapping mechanism, in which tool developers can specify whether instances and relationships shall be displayed as nodes, edges, or not at all. Among others, this allows to visualize and edit objects which represent relationships (association classes) as edges and to mask instances of specific types.

Customizer

Whereas the aforementioned view types focus on the instances and their relationships, the `CustomizerView` allows to edit the values of *primitive* properties. As shown in figure 7.10, `CustomizerViews` are forms which contain editing components for each property of a given object. The object to be edited in the `CustomizerView` must be selected in one of the other `BeanViews` (e.g., a `BeanTreeView`). The type of editing components depends on the type of the associated property, i.e. textfields for `String` properties and checkboxes for `boolean` properties. The default editors exploit the KBeans constraint mechanism by restricting

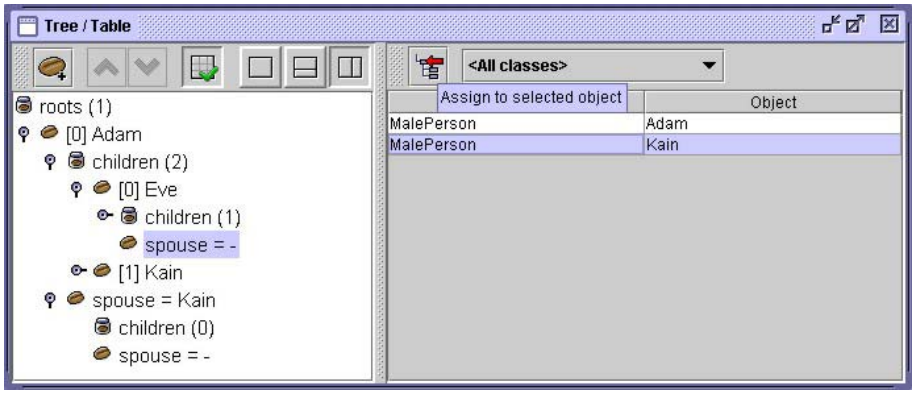

Figure 7.8.: The `TreeTableView` allows to edit the relationships between the model objects in a tree while displaying the potential valid values for the selected node in an associated table.

user input, for example to display a combo-box when only a finite number of values has been declared to be valid (see the `maritalStatus` combo-box in the right part of figure 7.10). `CustomizerViews` make use of JavaBeans's built-in support for `PropertyEditor` classes, so that individual editors can be defined for properties without having to modify the `CustomizerView` class.

Beside primitive properties, `CustomizerViews` can also be used to edit non-primitive relationships. For that purpose, users only need to activate the "plus" button in the upper left corner and appropriate lists or selection panels will appear. As shown in the right part of figure 7.10, referenced objects are recursively displayed by embedded customizers.

The default appearance of the `CustomizerView` can be completely replaced with custom components. KBeansShell automatically uses `java.beans.Customizer` classes, if they have been declared for a given JavaBeans class. These `Customizers` are Java components that have the full control over how an object's properties are presented to the user.

Constraint Violations

Users of KBeansShell-based tools can open a `ConstraintViolationsView` during knowledge acquisition. As shown in figure 7.11, this view lists the currently violated constraints, i.e. invalid property values. Constraint violations are checked after each modification in the knowledge base, so that the list is always up to date. For each violation, the object and property are displayed, together with a descriptive text and a repair hint that instructs users about potentially valid values. A double-click on a constraint violation selects the invalid property in one of the active views, e.g. a `BeanTree` or `Customizer`. Thus, errors can be fixed quickly. The view also provides buttons that allow to assign the proposed repair values directly.

Technically, the `ConstraintViolationsView` is based on the `ConstraintManager` class (subsection 6.4.4). It uses the `toString` method of the `ConstraintType` class (subsec-

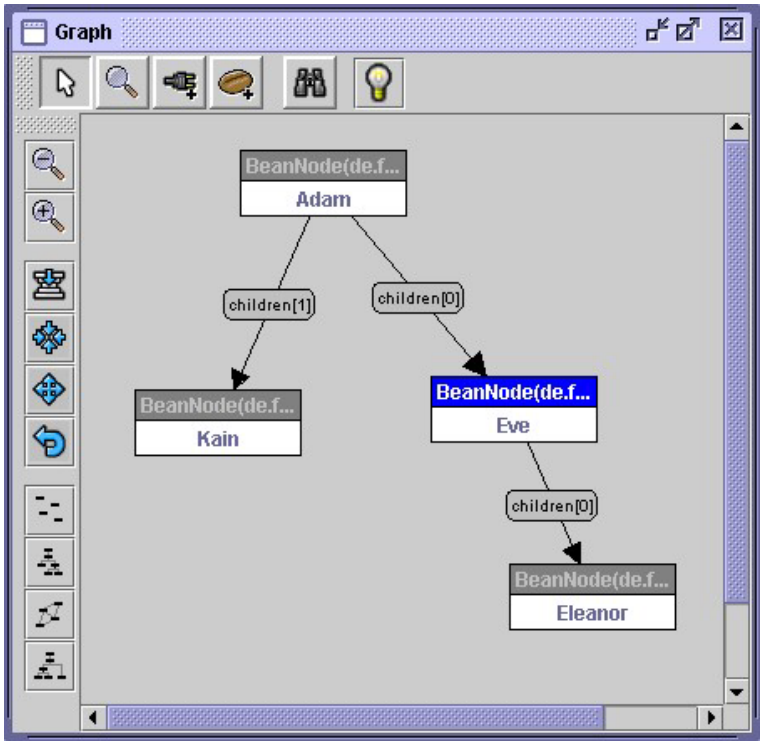

Figure 7.9.: The `BeanGraphView` is a highly configurable display of objects and their relationships. This figure shows a graph that is using the default settings.

tion 6.2.4) to generate descriptions of the constraint violation, and the `getValidValue` method of the `RepairableConstraintType` class to generate potentially valid values.

Other Potential Standard Views

In addition to the standard views described above, other types of generic components can be implemented. Frequently, custom components that have proven to be useful in a given project, can be generalized by replacing the references to specific ontology classes to reflection-based generic classes. For example, the `BeanTree` component was developed by exploiting the underlying design pattern of hierarchical relationships.

Other generic components can be developed to make reasoning modules accessible during knowledge acquisition. For example, a generic query module can be developed in which users are able to request all objects that match pre-defined search criteria. Furthermore, a view to access rule-based reasoning engines like the Java Expert System Shell (JESS, `http://herzberg.ca.sandia.gov/jess`) would be useful. Also, access to generic Problem-Solving Methods can be implemented, for which only a mapping from domain ontology to method ontology needs to be defined. Theoretically, any Java library or component can be installed as a `Module` in the KBeansShell, so that the overhead of integrating external classes is very small.

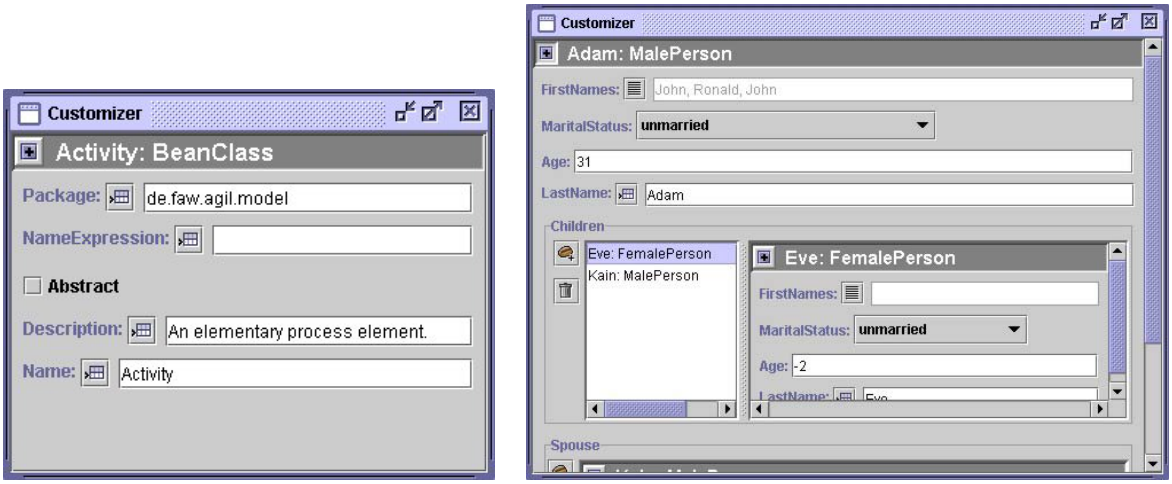

Figure 7.10.: `CustomizerViews` are editors of the properties of the object that is selected in the shell. Each property is automatically represented by an appropriate editor, such as a textfield or a checkbox.

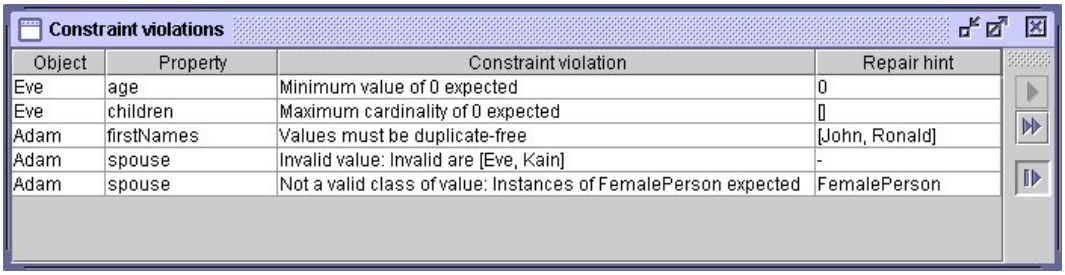

Figure 7.11.: The constraint violations view.

7.3. Using KBeansShell

How can knowledge-based system projects profit from KBeansShell? One of the first activities in an XP.K process will be to define a suitable domain ontology, so that knowledge acquisition can start. With an initial ontology implemented as KBeans classes, the KBeansShell provides a basic user interface with the default views and services described above. The shell can be invoked as-is and the only requirement is to pass the name of the "root" class to it, so that new knowledge bases can be instantiated. KBeansShell encompasses all the standard editors that are implemented in the comparable Protégé system. Our practical experience suggests that the default views are already quite sufficient for many knowledge modeling tasks, so that the initial ontology can be tested efficiently. In this context, KBeansShell can be regarded as a rapid prototyping environment of knowledge acquisition tools.

During the early stages of an ontology's existence, little resources should be spent on

custom modeling components like specific graphical editors, because the ontology might be relatively unstable. The reuse of generic standard components allows to quickly adapt to ontology changes. Successively, tool builders might want to replace the generic components with customized solutions. Light-weight changes can be made by implementing `PropertyEditors`, `Customizers`, or `NodeComponents` for specific properties and classes, without having to modify or overload the KBeansShell base classes. Greater changes can be made by adding custom views to the shell by installing appropriate `ViewModules`.

The KBeansShell supports testing through its built-in constraint checking module. It also allows to integrate customized `ActionModules` which can be used to execute automated tests and simulations at any time. Some examples for this will be provided in chapter 8.

7.4. The MetaKBeansShell

The aforementioned use of KBeansShell technology was focused on the acquisition of instances (knowledge bases). However, it can be applied to edit classes (ontologies) as well. I have implemented an extension of KBeansShell in which the objects are themselves classes, i.e. instances of a meta-ontology. This extension, the *MetaKBeansShell* is shown in figure 7.12.

The basic idea behind this tool is that classes of the metamodel level can be recursively represented as first-level instances from the model level as well. The MetaKBeansShell operates on a KBeans ontology that implements a set of classes similar to the metamodel from figure 5.10 (page 80), including classes like `PrimitiveProperty` and `Facet`. The MetaKBeansShell contains a `CustomizerView` to edit class and property details, displays the inheritance of classes in a `HierarchyView` (figure 7.5), and allows to edit classes in a UML-style diagram. The latter diagram is an application of the `BeanGraphView`, in which objects representing classes are mapped to graph nodes, and objects representing non-primitive properties to edges.

The major benefit of the MetaKBeansShell is its ability to generate Java source code. The classes of the meta-ontology have code generation methods and the MetaKBeansShell offers `ActionModules` which allow to start code generation for all or selected ontology classes. However, this code generation will overwrite any existing previous version of the class and thus also overwrite individual changes made by system developers or knowledge engineers. The tool therefore also supports incremental source generation and reverse engineering of existing Java classes. This Round-Trip Engineering makes use of the open IDE provided by the NetBeans/Forte Tool platform [183]. As illustrated in figure 7.13, NetBeans is a complete Java IDE which allows to add custom actions. The screenshot shows that the MetaKBeansShell (here called Round-Trip Engineering (RTE) Shell) can be opened for any given JavaBeans class from the IDE.

When launched, the shell uses the source code parsing tree supplied by NetBeans to create instances of the KBeans meta-ontology. These instances can be edited by the user and all source code modifications are synchronized with the source code. The NetBeans API

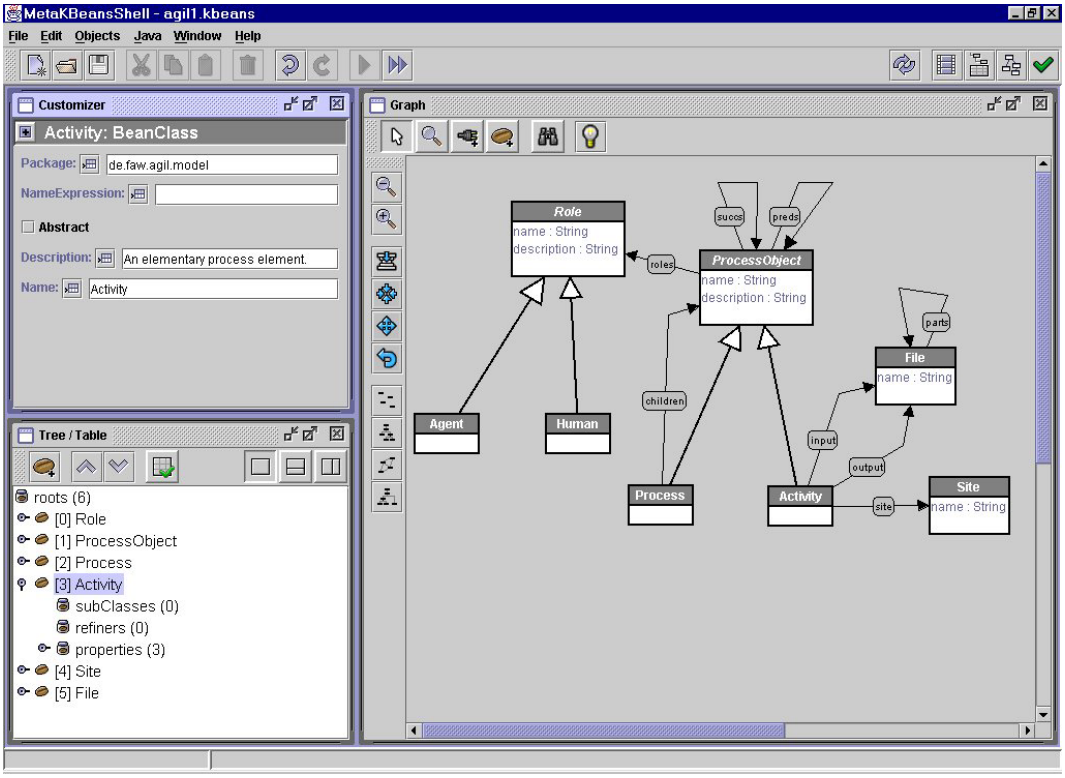

Figure 7.12.: A screenshot of the *MetaKBeansShell*, an extended KBeansShell for editing KBeans classes, including Round-Trip Engineering of Java source code.

supplies appropriate methods with which source code elements can be modified relatively easily. This approach is summarized in figure 7.14.

A clear weakness of KBeans technology compared with Knowledge Engineering approaches like Protégé is that the ontology classes need to be recompiled before changes can take effect. To absorb this limitation, the MetaKBeansShell can also be used to instantiate the classes while they are built but not compiled yet. For that purpose, the MetaKBeansShell provides menu items from which a new KBeansShell can be opened. This shell, however, does not edit ordinary Java instances, but uses a Active Object Model which simulates the Java classes with objects even when the classes are not "physically" existent. This is enabled through the KBeansShell's separation of model and implementation though the `BeanModel` facade. Instead of accessing real Java classes, the instances in this dynamic shell operate on virtual classes. As a valuable auxiliary effect, this allows to modify the classes while objects exist and thus supports schema evolution without having to give up the instances.

The MetaKBeansShell is an impressive demonstration of the flexibility of KBeans technology. It even makes use of the KBeans facet engine. A number of constraints on the classes of the meta-ontology is defined, so that, for example, invalid class and property

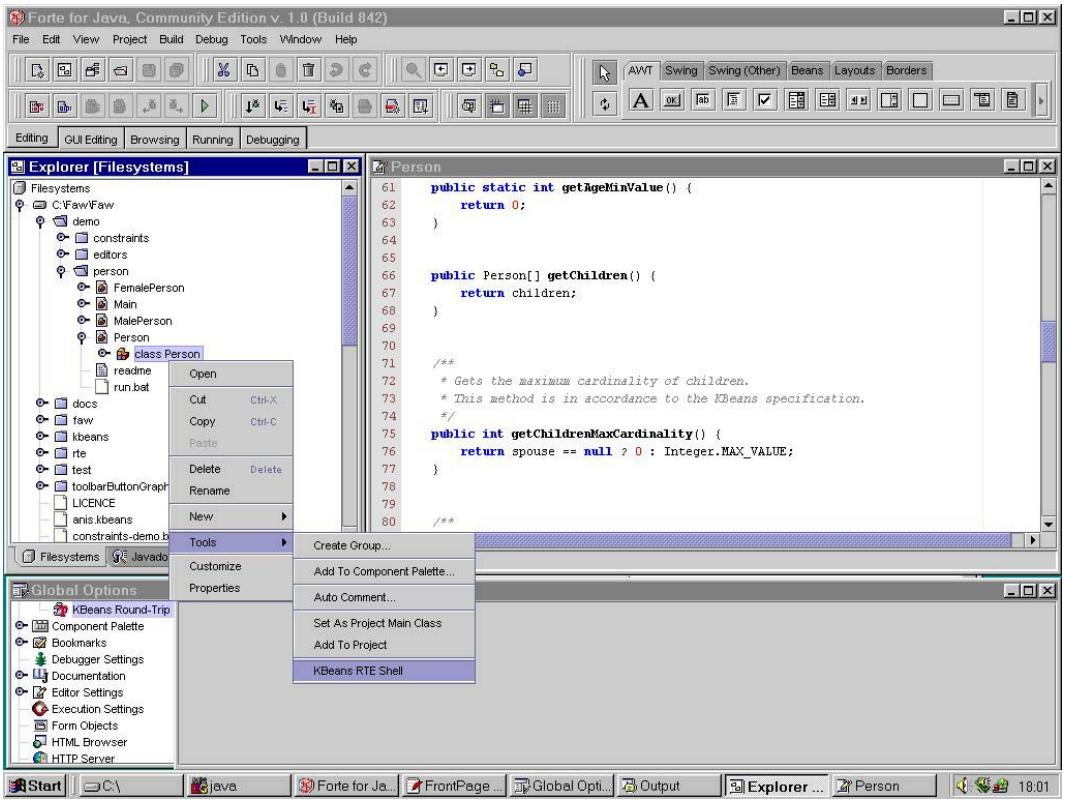

Figure 7.13.: The MetaKBeansShell can be installed as a plug-in for development tools such as Forté for Java. The shell can be invoked to edit the selected KBeans classes.

names are reported during modeling. This does not only point the knowledge engineers to missing or inappropriate model elements, but also makes synchronous Round-Trip Engineering possible, because it is impossible to generate and update valid source code when constraints are violated. For instance, before the MetaKBeansShell can generate code for a newly-added property, it has to wait until the user has specified a valid property type. The shell uses a `ConstraintManager` to halt source code synchronization until all constraints are satisfied.

7.5. Summary

This chapter has introduced a framework for the construction of knowledge modeling tools called KBeansShell. KBeansShell supports the rapid development of such tools by assembling reusable components. Comfortable graphical knowledge editors like trees and forms are immediately available after an ontology has been implemented or changed. The reusable components of KBeansShell rely on reflection to detect the properties of the avail-

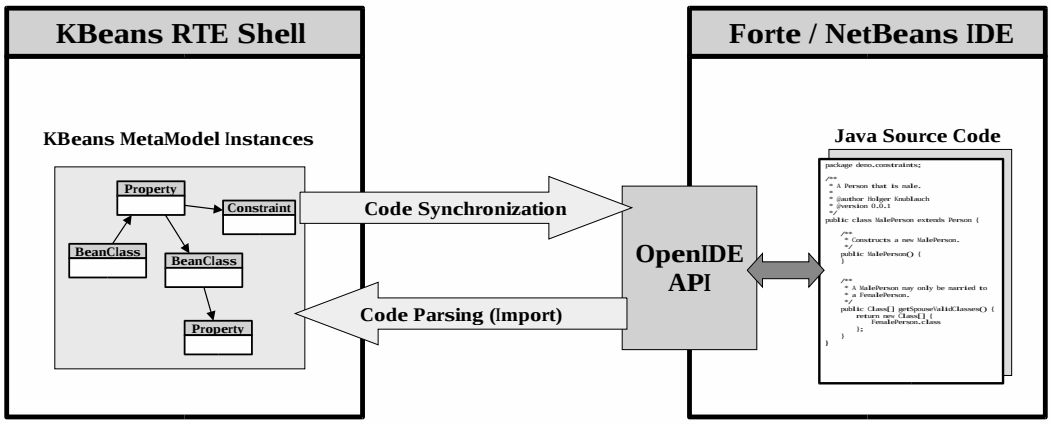

Figure 7.14.: Round-Trip Engineering of Java source code with Forte.

able ontology classes at run-time. The shell provides built-in mechanisms for constraint checking which help to detect modeling mistakes early and to support a test-first modeling style in the sense of Extreme Programming. User-defined extensions like custom editors, simulation modules, and reasoning components can be easily integrated.

In the face of the criteria for modeling tools defined in chapter 2, our users report that KBeansShell is easy and enjoyable to use. Its main strength is its flexibility and adaptability through its open architecture. The shell's generic base services support exploration of ontologies. Additionally, the tool user is supported by mindful constraint checking mechanisms.

The major weakness of KBeansShell and the major obstacle to putting it into practice is that it is – like most of the existing knowledge acquisition tools – not supported by large commercial vendors and does not have a large user group. Instead, it is a research prototype built by a single programmer (me). Although the shell executes very stable, it should not be compared to professional tools. However, the architecture presented in this chapter can serve as a foundation for a professional re-implementation.

8. Case Studies

This chapter reports on application-oriented research projects in which elements of XP.K have been applied. As already stated in the introduction, XP.K could only be partially tested. Since the projects that I have been involved in were quite small (with only up to four people involved), Extreme Programming practices like Pair Programming and the issues involved in communication could only be partially evaluated. Also, spatial distribution of project partners made following the On-Site Customer practice difficult. XP.K grew out of running projects that originally did not have the intention to explore an Extreme Programming approach. However, research projects with only a coarse-grained goal defined from the onset are realistic test fields for agile development processes, because requirements, people, and models typically change frequently. I can therefore report on experiences made with most of the elements of XP.K, being aware that the XP.K practices should not be performed in isolation because they depend on each other. Anyway, this chapter describes several examples in which KBeans technology was applied successfully.

XP.K is basically a byproduct of the two German research projects ANIS (1998–2000) and AGIL (2000–2002) in which I have been the executing researcher. *ANIS* (section 8.1) aimed at developing a prototypical knowledge-based patient monitor for decision support in anesthesia. During the course of this project, KBeans and KBeansShell were developed. Furthermore, a simple rule engine (ontology) emerged from ANIS. *AGIL* (section 8.2) focused on the development of a multi-agent system to optimize the information flow between clinicians. This project made two significant contributions to XP.K. First, KBeans technology was successfully employed for the acquisition of knowledge about clinical processes. Second, based on the resulting process models, the agents of the multi-agent system were designed and partially implemented in an Extreme Programming style by students during a course at the University of Ulm in 2001. Section 8.2 reports on our experiences made during this course. The chapter is summarized in section 8.3.

8.1. The Clinical Decision Support System ANIS

Clinical decision-making is based on various types and sources of knowledge, like raw data about patients, information about the current state of the treatment process, medical knowledge, and general problem-solving knowledge. Anesthetists in modern clinics rely on an armada of clinical devices and other information sources. The information overload caused by the continuous stream and quantity of patient data delivered by clinical

monitoring devices and the inherent complexity of the anesthesia process lead to slips, omissions, and mistakes by medical staff that might result in critical incidents. The main idea behind the research project ANIS (Anesthesia Information System) [108, 76] was that "intelligent" patient monitoring devices should assist in operating rooms in order to reduce the number of these – mostly preventable – critical situations. The project was performed by a clinical partner from the Technical University of Berlin and the FAW Ulm where I was the executing researcher together with a student assistant. The spatial distribution of project partners (600 km) later proved to be an important obstacle to project success.

The project's primary result was a prototypical knowledge-based patient monitor for the induction phase of anesthesia, which was based on a coarse-grained model of the anesthetic processes. For each of the identified process phases (information, monitoring, medication, and ventilation) we have implemented intuitive graphical components and screen layouts [166]. Sample screens for monitoring and medication are shown in figure 8.1. Using touch screen buttons or voice control, clinicians can switch between these screens and access detailed information, such as the patient's history and pharmacokinetic simulations. If the system detects a change in the patient's state, which might indicate an evolving critical situation, the key factors are emphasized and presented to the human decision-maker. In order to be able to detect critical incidents, the system contains an explicit knowledge base.



Figure 8.1.: Screenshots of the ANIS patient monitor.

8.1.1. Domain Knowledge Acquisition in ANIS

Whereas user interface and computing infrastructure posed only technical challenges, the construction of a suitable knowledge model proved to be very difficult. In the early project phases, we relied on approaches described in the Knowledge Engineering literature and used the Protégé system for knowledge modeling. However, we soon recognized that the maintenance of a hybrid system which mixed Java components with models in the Protégé

ontology language was expensive. As already discussed in the previous chapters, the transitions between knowledge models and the remaining software system were not smooth, in particular because dynamic run-time data had to be combined with static domain knowledge. This led to the development of KBeans and an initial version of KBeansShell. We used this technology to design an ontology for anesthetic knowledge. This domain ontology contains concepts like `Drug`, `Observable`, `TreatmentPlan`, and `Trend`, as well as a collection of concepts to build logical rules. This domain ontology was designed to be close to human intuition, so that clinicians could edit knowledge bases in the terminology they are used to. In order to provide credible and reliable assistance for medical staff, any medical information system has to generate recommendations in a transparent and explainable way based on accepted clinical guidelines. We hoped to support this transparency by exposing the system's knowledge with an explicit and traceable model. The knowledge instances themselves were edited with a knowledge editor based on the KBeansShell, which is shown in figure 8.2.

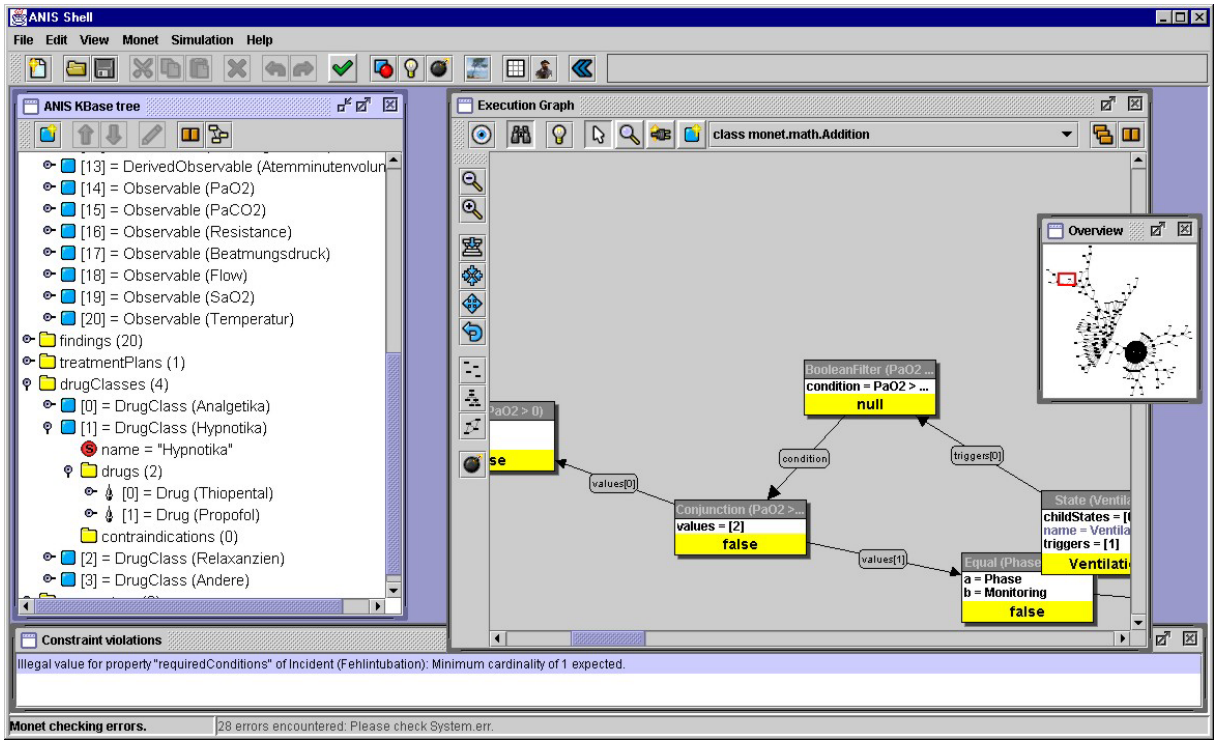

Figure 8.2.: A screenshot of the ANISShell, a knowledge acquisition and test tool for anesthetic domain knowledge. Note that this tool is based on an older version of KBeansShell, so that layout and visual appearance differs from chapter 7.

8.1.2. Problem-Solving Methods in ANIS

The domain knowledge models of ANIS can be translated into an executable form. For that purpose, we have implemented a generic Problem-Solving Method called *Monitoring Network* which is essentially a rule engine for propositional logic. The state of the patient, the operation, and the clinical environment are captured in numeric and discrete variables that are modified by external units like monitoring devices. On arrival of new data, the rules containing the respective variables are evaluated, resulting in the eventual update of other variables. Thus, changes are propagated through the net until no further rules fire. A very similar algorithm is known as the *RETE algorithm* [62] in other expert systems.

The Monitoring Network is a KBeans ontology with embedded reasoning methods. It proved to be very efficient to attach the node evaluation code directly to the ontology classes, because it was thus possible to make use of method overloading and other efficient programming styles. The instances of the Problem-Solver's ontology (the network of rule nodes) are created by mapping the domain ontology to the network's ontology. For example, the domain concept of `Observable` is mapped to the network concept `Numeric-Variable`. This mapping was very easy to implement by attaching mapping methods to the domain ontology Java classes. It would have been very hard to implement these mappings in other languages because we sometimes needed to parse expression strings (e.g., `"BP > 89"`) to create nodes of the Monitoring Network. The flexibility of Java allowed us to implement such parsers efficiently. The mapping process itself is performed when the ANIS system starts. Since the Problem-Solver's ontology was implemented with KBeans classes like the domain ontology, it was possible to reuse the generic view components of KBeansShell for the visualization of the network instances as well. For that purpose, the ANISShell knowledge editor contains a button with which the network instances are generated and visualized as a graph (as shown in figure 8.2). With this graph, knowledge engineers are able to trace the internal reasoning processes and states of the ANIS system. The ANISShell can be executed from within the executing patient monitor application, so that tracing the system's behavior is possible at any time.

Beside access to the reasoning engine, ANISShell also provides complex test and simulation features. First, the domain ontology was enriched by constraints which can be used to detect and visualize inconsistencies at knowledge acquisition time. Second, the shell included a simulation engine with which patient data could be generated, either from mathematical formulae or from recorded patient data streams. Third, small trace windows like an overview of all "observables" in a table and a log of the past blood pressure values were implemented. These features were seamlessly integrated into the ANISShell and contributed to knowledge base maintenance.

8.1.3. The Knowledge Acquisition Bottleneck

Despite the comfortable knowledge editing features, the ANIS project suffered from a rather fundamental problem. The construction of a knowledge base proved to be very difficult, because our clinical partners found it hard to map medical knowledge onto discrete

and precise specifications. For example, the question whether a certain plunge of heart rate indicates a critical incident or is normal in the given context could not be formalized appropriately. The clinicians refused to make definitive statements about thresholds of observables, trends, et cetera. Thus, the amount of tacit knowledge and the complexity of the domain led only to a very shallow knowledge base. During the course of the project, we discovered that the initial goal of implementing intelligent decision support in anesthesia had to be restricted to those aspects where knowledge could be made available in a sufficiently formal form. In succeeding projects like AGIL (see section 8.2), we would focus on the traceable and well-defined *clinical* processes instead of the patient's internal processes.

Another critical point in ANIS was the spatial distribution of domain and technical experts. Communication mostly took place by telephone. Due to the tight traveling budget, only few face-to-face meetings were arranged. Our experiments with an early videoconferencing system failed because of a low internet bandwidth and technical problems. The lack of fluent communication links led to misconceptions, because domain experts and engineers had a different understanding of the meaning and purpose of the various modeling elements. In my opinion, this knowledge acquisition bottleneck could have been widened by following a process like XP.K. If we would have had more time to *jointly* build the knowledge models, a sustainable compromise between tacitness and rigorousness might have been easier to find. By far the largest parts of the ANIS knowledge base were acquired during the few face-to-face meetings. These meetings furthermore significantly increased team motivation and commitment.

8.1.4. A Rule Engine as a Byproduct of ANIS

ANIS was unconsciously developed with some of the Extreme Programming practices. In particular, we complied with the value of Simplicity and started with very simple solutions first. In later projects stages, for example when a certain part of the ontology had proven to be stable over various releases, we performed several medium- to large-scale refactorings, in which we generalized our solutions. For example, the dependency of the KBeansShell tool from the specific ANIS ontology was successively removed, so that the shell could be used with *any* ontology. Furthermore, we eliminated all references from the Monitoring Network classes to the domain ontology classes, so that the Monitoring Network became reusable in other projects. We later showed how this rule engine can be employed for a completely different reasoning task than initially expected, namely for the capture and dissemination of experience about the construction of engineering processes [160, 158].

A part of this rule engine was a collection of classes to represent rules in a simple propositional logic with expressions like `IF N>10 AND B THEN D="Thorax"`. The point is that the construction and evaluation of such rules is a frequently needed feature in knowledge-based systems and relatively easy to comprehend by domain experts, so that it should be implemented in a reusable form and supported by utility classes like parsers and checkers. It was possible to extract and generalize these features from the Monitoring Network classes by refactoring the source code, leading to the classes illustrated in appendix H.

In order to reuse these `kbeans.logic` classes, they must be adapted to the specific ap-

plication domain. The package contains three abstract classes which need to be overloaded. For example, if used for the acquisition of logical rules about patient data, the abstract class `NumericVariable` must be overloaded with `Observable` (or `PatientDate`) and the class `DiscreteVariable` with `PatientState`. All the other classes remain unchanged. Reuse of these other classes is performed automatically, so that a complete rule ontology can be integrated with ease. Domain experts do not have to deal with computational entities (like "numeric variables"), but can define instances of concepts they may find more tangible (like "patient data"). Since ontologies are used to facilitate communication among domain experts and between domain experts and knowledge-based systems the expert's view of the domain should be reflected in the ontology [136].

On a rather general level, this is a good example for successful application of object-oriented *frameworks*. A framework is a collection of components for accomplishing a task (see subsection 3.1.2). Some of these components are designed to be replaceable or refinable (sometimes called the *hot spots* [147]). In the `kbeans.logic` package, the hot spots are the aforementioned three abstract classes. Complex frameworks like KBeansShell may have many more hot spots.

Another general observation from this is that the separation between domain ontologies and Problem-Solving Method ontologies might be much more blurred than anticipated by many Knowledge Engineering researchers. If knowledge is acquired for a specific task (here, the evaluation of rules to update internal states), then the domain ontology (the expressions in the clinical terminology) must already be compatible to the method ontology (the logical rules). Approaches like the `kbeans.logic` engine soften the Knowledge Engineering paradigm of mapping domain ontologies to method ontologies, but rely on object-oriented mechanisms like inheritance instead. This example suggests that in many cases the separation of structure and behavior, as propagated by modern Knowledge Engineering and ontology languages, might be less suitable for knowledge modeling than its integration with objects.

8.2. The Clinical Multi-Agent System AGIL

Following the ANIS project, I was involved in the project "Agent-based Information Logistics" (AGIL) [105, 167, 102]. In this project, a prototypical multi-agent system has been developed using an Extreme Programming approach. The task of this system was to support the clinical information flow, delivering the right information to the right person at the right time. The scenario of AGIL has already been outlined in figure 1.1 (page 2). My partner in this project was a domain expert (an anesthetist) with experience in the analysis of clinical workflows. In AGIL, I was able to test various aspects of XP.K. However, the focus of AGIL was not on the construction of a monolithic knowledge base like in ANIS, but rather concerned the acquisition of knowledge about clinical processes with the goal of identifying, designing, and implementing clinical agents. In other words, the system's knowledge did not reside in the single agents but in their interaction scenarios. For the development of these agents, XP.K proved to be very efficient.

In order to motivate the application of XP to the domain of agents, it is first necessary to clarify what agents are. An *agent* is an encapsulated knowledge-based system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [93]. Agents differ from conventional software particularly in their complex interaction and communication scenarios. While the properties of agents make multi-agent systems a very natural and efficient approach for solving many types of problems, the complex communication scenarios and the emerging system behaviors often lead to situations that are difficult to predict and plan [111]. Any development methodology for multi-agent systems must take these difficulties into account.

The currently discussed *Agent Oriented Software Engineering* (AOSE) [32] approaches face the complexity of agent development by defining modeling languages, processes and tools to systematically divide complexity into a collection of inter-related models. Like most of the Knowledge Engineering methodologies, AOSE approaches follow the traditional Software Engineering paradigm, in which a relatively large chunk of project resources is spent on up-front analysis and design while implementation and evaluation are moved to phases when the requirements are thought to be sufficiently understood. This leads to well-known advantages, e.g. that development processes become reproducible and (at least apparently) planable, but also leads to an overhead when models need to be changed. In AOSE methodologies like *Gaia* [198] the design model is relatively decoupled from the implementation, i.e. the entire design (perhaps even the analysis) has to be revised in order to develop a model that can actually be implemented [111]. Customer feedback is available late, so that systematic AOSE methods are suitable only if requirements are relatively stable. This is often unrealistic, because the complexity of potential agent interaction scenarios and the emerging behaviors within a multi-agent system can make preplanning very difficult [111].

In order to explore alternatives to systematic Agent-Oriented Software Engineering methods, I have adapted XP.K for the development of multi-agent systems. The approach is guided by the XP.K values of Simplicity, Community and Feedback. In particular, it relies on very simple models and metamodels so that agent interactions can be modeled, changed, and communicated quickly. Following the principle of Traveling Light, the approach suggests to build and maintain only two models: The *source code* of the executable agents (including test cases) and a *process model*. The process model represents the domain expert's knowledge about the existing process and the application scenarios of the agents in the visionary new process. Thus, the processes can be regarded as both knowledge base and story cards in the sense of Extreme Programming. As illustrated in figure 8.3, this process model is the starting point of the implementation and test activities. The process model allows to generate parts of the source code (details below) and serves as a requirements statement that can be communicated between the team members. The approach explicitly assumes that the initial process model will be incomplete or wrong, so that it will need to be updated in the face of feedback from the implementation. We therefore yield a cyclic development process that allows to switch between implementation and model updating arbitrarily.

The following subsections will provide details on our process modeling approach (8.2.1)

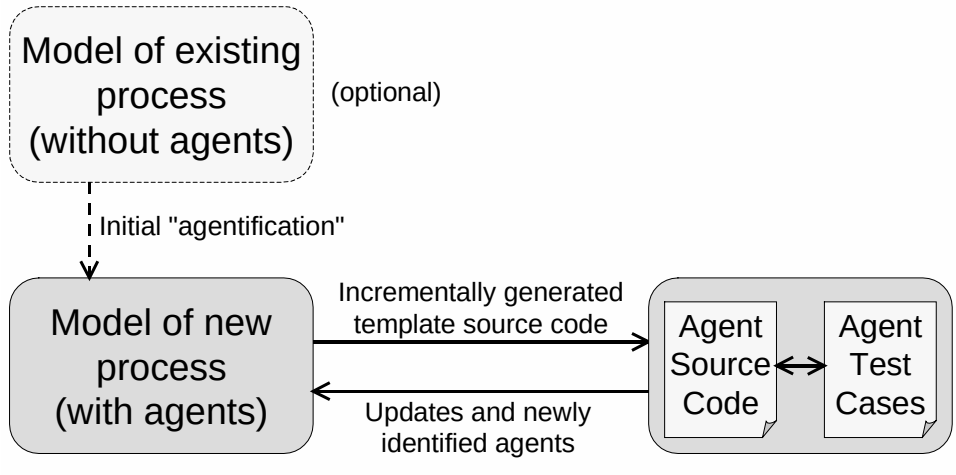

Figure 8.3.: An overview of an XP.K approach for multi-agent systems.

and the XP implementation process (8.2.2). Finally, subsection 8.2.3 discusses strengths and weaknesses of XP.K that only concern multi-agent systems. A general discussion of XP.K will follow in chapter 9.

8.2.1. Process Knowledge Acquisition in AGIL

In the beginning of the AGIL project it was clear that agents, with their ability to fulfill autonomous, pro-active, and distributed tasks, have the potential to optimize clinical information exchange. However, the types of agents, as well as their precise application scenarios were an open issue. Since we were faced with an existing workflow into which agents were to be introduced, we started the project with the acquisition of a model of the clinical processes that we could then modify incrementally by assigning more and more tasks to agents.

Our first task therefore was to define a modeling approach which allowed us to capture the existing processes efficiently and to acquire potential agent application scenarios. Our clinical partners had already used their own process modeling approach in previous projects. They have used the standard tool Microsoft Visio to draw process graphs. These graphs included activities, which are performed by the clinical staff, and documents like patient data sheets, which are produced during the activities. Although this modeling approach was sufficient for basic process analysis, it resulted in models that were inconsistent and which did not allow any formal processing for agent development. We therefore decided to build a customized modeling approach and tool. For that purpose, I met with the domain experts to define an initial process metamodel (or ontology). This metamodel was designed to be close to the experts' existing intuitive view, to be extensible for specific types of agents, and to allow for automatic and semi-automatic transformation into executable code. In support of communication between domain and technology experts, the metamodel was deliberately designed to be extremely simple. The result of this Joint

Ontology Design later proved to be surprisingly stable throughout the project, since it reflected both domain and technical points of view.

A schematic overview of our process metamodel is provided in figure 8.4. A `Process` is a collection of sub-processes and `Activities`. An `Activity` is an atomic unit of work that is performed by an actor who takes a specified `Role`. A role can be taken either by a `Human` or an `Agent`. `Agents` represent types of software agents that can be arranged in an inheritance hierarchy like object-oriented classes. `Activities` and `Processes` can be arranged in arbitrary predecessor/successor sequences, although we have used this ordering mostly for illustration purposes only. `Activities` can read and write `Media`. A `Medium` is either analog, like a phone call or a letter, or an `AgentMessage`. An `AgentMessage` can be characterized by its command (or performative) and its `Ontology`. In AGIL, we relied on KBeans ontologies, so that agents could pass Java objects as message content. This enabled agents to automatically check the consistency of message content on message receipt.

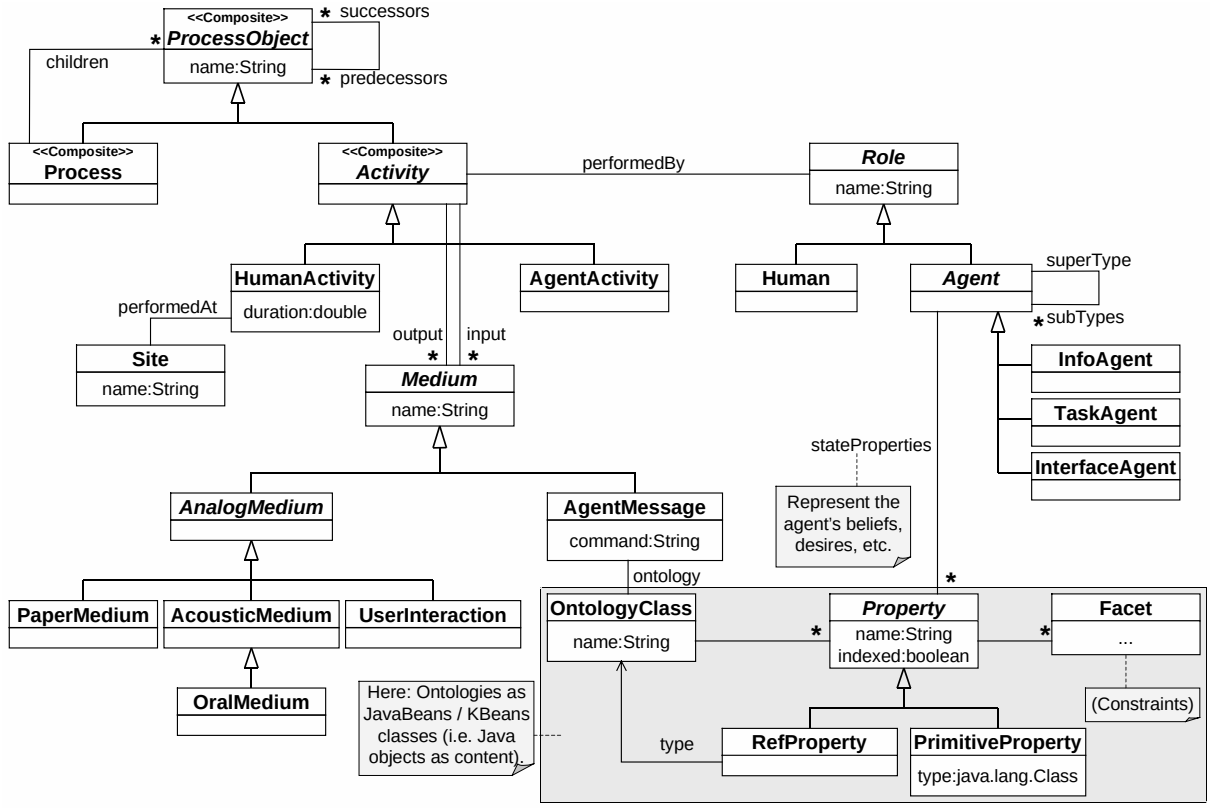

Figure 8.4.: The AGIL metamodel (ontology) for agent-based process models.

This process metamodel was implemented as a KBeans ontology. Several constraints were declared on the ontology concepts, in particular to point the users at missing input and to prevent duplicate identifiers in the code generation. Then I derived a subclass of KBeansShell which allows to acquire some instances of the ontology, i.e. some process

models. I used the default editors of KBeansShell to translate some of our partner's old process models into the new schema, which led to several minor ontology adaptations. After the ontology proved to be sufficiently stable, I implemented a graphical process view, which enabled the domain experts to acquire their process knowledge in a purely visual style by dragging boxes and arrows. Figure 8.5 shows a screenshot of this tool, which is called *AGILShell* [104]. AGILShell is able to automatically generate several types of views in addition to the normal bird's-eye process view. For example, it allows to visualize the message flow between agents in a graph, with agents as nodes and messages as edges. Another view can be activated in which life-cycle, input and output of a single role (agent) are shown.

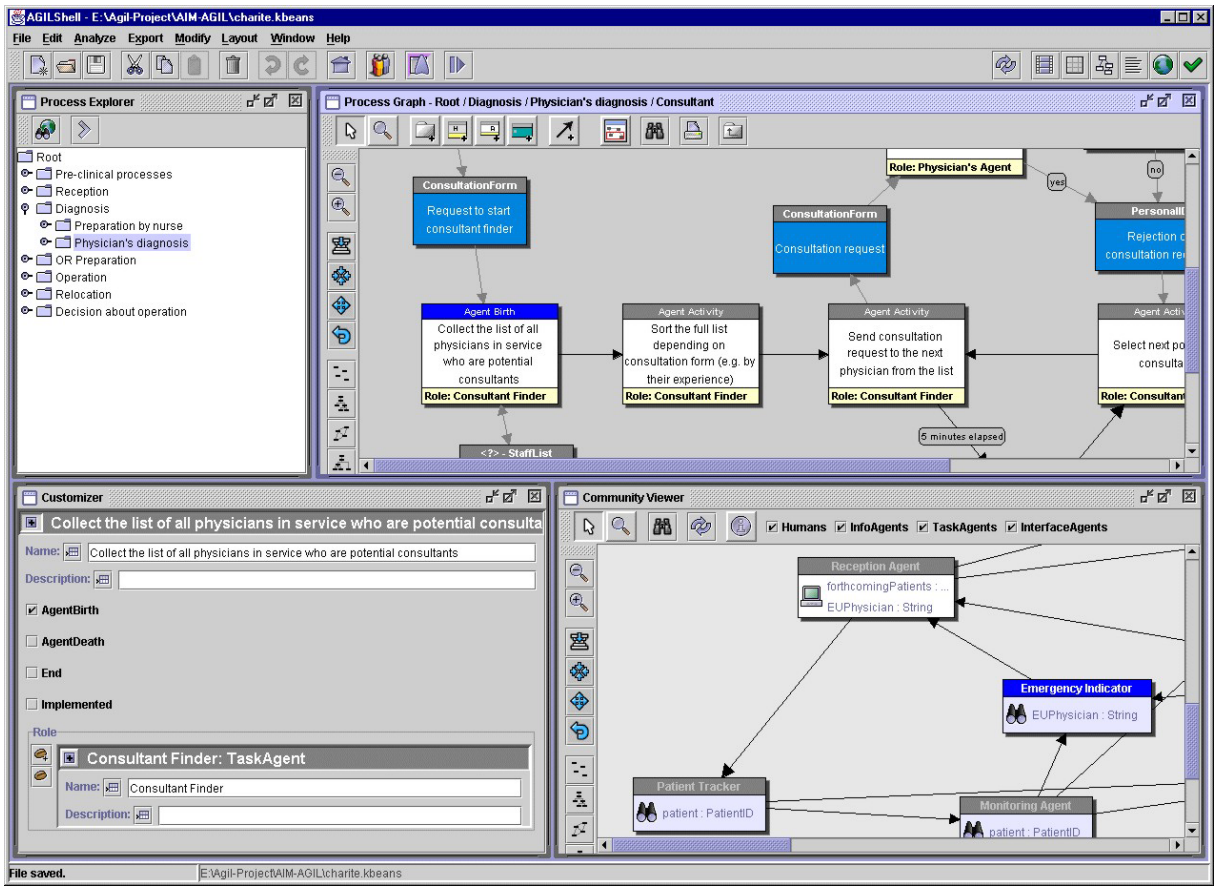

Figure 8.5.: The AGILShell, a tool based on KBeansShell for the acquisition of process knowledge with support for editing and analyzing agent interactions, and code generation features.

Beside the different views, which can help to clarify agent scenarios and to detect bottlenecks, the AGILShell provides code generation features, with which Java source code can be created. This code includes templates for agents and agent test cases, and the source code of the ontology concepts that are used by the agents for communication with

messages. Since agent messages pass KBeans objects, the code generation module reuses the class modeling and code generation approach developed for the MetaKBeansShell class editor (section 7.4). Due to limited time, the available version of the code generator does not provide true Round-Trip Engineering, although this is technically feasible and will be implemented in the follow-up project of AGIL.

Note that the process metamodel above is rather an informal base which can be custom-tailored for other projects with specific needs. It is well possible to add new attributes or concepts that might be needed to express specific agent characteristics (like "beliefs" and "desires"), or to improve code generation. The AGILShell inherits the very open architecture of KBeansShell, so that it automatically provides editors when custom attributes are added to the metamodel classes.

In the AGIL project, the AGILShell was used for the acquisition of two models of process knowledge. The first was more or less autonomously designed by our clinical partners, who have observed the clinical processes on-site and generalized them to a general model of clinical practice. This included 34 sub-processes with 144 activities and 26 human roles, and was designed over a period of several weeks. After that, the model was left unchanged and served as a backbone for the "agentified" process model.

The existing process model was used to derive a new process model that included an initial set of agents and their communication. For that purpose, the domain expert and the leading developer (me) met for a three-day modeling session in which the initial application scenarios were defined. The direct communication and the highly informal modeling approach allowed us to visualize our ideas very quickly. The joint modeling approach furthermore helped to consider both domain and technical points-of-view, although we did not spend much time on formal details. Our visionary model equipped each of the human actors in the original process with graphical interface agents, most of which intended to run on mobile PDA devices. We removed all human actors from the design model in order to reduce complexity and because humans were then represented by their individual interface agents. The new process included 29 sub-processes with 159 activities and 32 agent roles. The resulting "agentified" process model was fed into an XP implementation process, as described next.

8.2.2. Agent Implementation and Test

Our process modeling steps have identified and defined the types of agents, their input and output messages, and an informal description of their behavior. The implementation activities in our XP approach led to executable source code, e.g. Java classes that implement the agents and the remaining modules. In our experiments with an XP process for agents, we used a very simple and light-weight agent platform that focuses on the main characteristics of agents, in particular their communication through messages (see [102] for details). This platform supports the spirit of XP in so far that it is extremely simple and allows to implement and test agents rapidly. In the sense of XP's unit testing, we have developed test cases for all agents individually, whereby all adjacent agents and system

units were simulated by "dummy" objects. Template source code for agents and their test cases could be incrementally generated from the process models by the AGILShell. The developers thus only needed to complete the message handling methods and the test case bodies.

For the implementation, we organized a practical course for third-year Computer Science students at the University of Ulm. The course involved eight students, me, and the (clinical) domain expert. The domain expert was permanently available to answer questions and to clarify and weigh requirements. The setting was a single office with four personal computers arranged in a circle so that the programming pairs could see each other. As shown in figure 8.6, the room was equipped with a beamer that was connected to the coach's laptop, so that prototype demonstrations and process models could be visualized for everyone to see.



Figure 8.6.: Participants of the XP course in which a multi-agent system has been implemented in pair programming style.

None of the students had any prior experience or training in XP, some of them were beginners in Java and none of them had implemented agents before. We therefore used the first two days of the course to provide the students with a theoretical introduction to XP and a practical introduction to the development tools (IntelliJ [86] and a Concurrent Versioning System (CVS)). On the second day, the students learned to implement test cases for some rather simple agents. The agent platform, the modeling framework, and the tools proved to be very easy to comprehend.

40-Hour-Week The practical work itself was done during one 40-hour week. The students were explicitly not encouraged to work overtime. After the course, the students reported that they used to be quite exhausted after a full day of pair programming, but were very disciplined and concentrated while in the office. Nevertheless, the atmosphere was very relaxed and enjoyable and thus stimulated creativity and open, honest communication. We supported this atmosphere by providing free coffee and cookies and by

organizing a social evening during the week.

Planning Game At the beginning of each day, the team jointly followed a planning game approach to define the features that were to be implemented next. Since the process model described the phases of a patient's treatment on her way through the hospital in a rather sequential style, we found it most useful to implement the agents in their order of appearance within the process. We locally focused on those agents that – according to the domain expert – promised the most business value.

Pair Programming The agent implementation itself was then assigned to the developer pairs. The code generator described above was used to create one Java package and template code for each agent. Each pair had to develop and test their individual agent in isolation, using the XP testing framework JUnit [66]. The students found pair programming very enjoyable and productive. One student reported later that he felt much more motivated and concentrated than if he had to work alone. Most students reported that they learned many useful programming techniques from each other during the course. However, due to the different experience and background of the team members (e.g., some were not knowledgeable of GUI programming), not all student combinations proved to work equally well. The student's individual experience had a significant impact on code quality, so that I tended to delegate difficult tasks to the advanced programmers.

Testing Our agents proved to be quite easy to test, because their behavior and state changes mostly depended on incoming messages only. Many tests therefore consisted of sending a test message to the agent and of checking whether the expected reply message was delivered back. The students found testing quite useful to clarify requirements although it was considered to be additional work by some. During the course, the students have implemented 30 `TestCase` classes with 76 test cases, amounting to 4909 lines of code, while the 43 agent source code classes amount to 5673 lines (excluding ontology classes). The students enjoyed using JUnit very much, because the sight of the green bar that indicates correct test runs improved motivation and trust in the code. The main problems with testing were that some refactorings were needed to enable testing of certain functionality, and that automated tests of visual interface agents are difficult.

Collective Ownership We applied a relaxed practice of collective ownership, which allows any team member to modify any piece of code at any time. Since each pair only operated on the source package of a single agent, there was barely any overlapping. Only the ontology classes were shared among agents and thus modified by various teams. Coordination of these changes was accomplished very informally by voice and the CVS.

Coding Standards In the beginning of the project, we defined a project-wide coding standard that was very easy to follow, because the Java tool we used provided automated

code layout features. Thus it was very simple to shift implementation tasks between the pairs and to change pair members regularly.

Simple Design The students were explicitly asked to focus on programming speed instead of comprehensive up-front designs. This seemed to be sufficient because the agents were rather small units with few types of tasks to solve. Despite the focus on simplicity, experienced students almost automatically identified some useful generalizations of agent functionality. Our initial process model underwent several evolutionary changes, in particular we frequently encountered scenarios where agents were unable to fulfill their task because they did not have access to data or information that other agents possessed. We therefore had to add some activities and messages that pass missing data items between agents. When changes were necessary, I first used to modify the process model using AGILShell, which seldom took longer than a minute. Then I used the tool's code generator to update the source code, so that the responsible programmers had less to type. Despite the various small changes, the overall design remained quite stable throughout the project, so that our process modeling framework proved to be sufficient.

Refactoring Since the agents were rather small units, they were very easy to maintain and refactor. Some medium-sized refactorings, like the introduction of new superclasses to generalize functionality, were performed. Smaller refactorings, like the introduction of temporary variables, were performed when code became too hard to read. Some other refactorings were necessary to enable the implementation of automated tests.

Continuous Integration and Short Releases The agents were uploaded onto the CVS server and integrated at least every evening. Since the students were only allowed to upload those agents that passed all test cases, there were almost no integration problems. Agent interactions were tested and presented on the beamer with the help of a small simulation environment that could trigger external events. A student later described the integration shows as the daily highlight, because the agents that were programmed and evaluated in isolation suddenly interacted with real other agents.

On-Site Domain Expert In the questionnaires that were filled out by the students after the course, the presence of the domain expert was very positively evaluated. He was asked to provide clinical knowledge regularly, at least once an hour, so that expensive design mistakes were prevented. His presence did not even mean an overhead for him, because he could use the "spare time" for other types of work on his own laptop. We found that the communication process is characterized by reciprocities between engineers and the experts. As the domain expert got more and more used to the formal view of the developers, he adjusted his modeling style, and vice-versa.

The XP project resulted in the full implementation of 17 agent types, some of which with complex graphical interfaces. These agents cover about the first third of the original process model and solve tasks like notification, information filtering, and patient monitoring

(figure 8.7). The agents make use of 38 KBeans ontology classes. In the implemented excerpt of the overall process, 5 additional agent types were identified in the coding phase, in particular agents that provide generic services for other agents. The agent scenarios, however, underwent much more significant changes. The implemented model includes almost twice as many activities and agent messages as originally designed. The additional activities often concern information transfer between agents, because many agents lacked access to resources that they required to solve their tasks. About half of the additional messages, however, add new application scenarios and functionality that was originally not thought about. This functionality had emerged from the creativity of the team members during the implementation phase.



Figure 8.7.: An interface agent developed during the XP course.

8.2.3. Discussion of XP.K for Agent Development

The significant increase of the process model's size during the implementation phase indicates that our initial agent design was inadequate. This is no surprise, because we intentionally spent only very little time on this pre-modeling process, so that we were able evaluate XP/XP.K with a realistic starting point. However, it is yet unclear whether a traditional Agent-Oriented Software Engineering approach would have produced a better model more efficiently, because we did not have the opportunity to run corresponding modeling experiments yet. I can therefore base my discussion of XP for agents only on my subjective experience, and by drawing parallels between multi-agent systems and reports from mainstream software technology.

Retrospecting on the case study, I am very pleased with this first XP experience. Although neither the team members nor the coach were experienced in XP or agent development, the approach has shown to be quite efficient for the development of a prototypical, medium-sized multi-agent system. All team members reported that XP was in general much more enjoyable than following a strict, rather bureaucratic methodology. Student feedback about the course was overwhelmingly positive. The team worked highly concentrated and disciplined, and the fluent communication within the team fostered creativity. The personal relationships between the students were intensified and the team members

felt responsible for "their" agents. Although not all students were equally well team players, this clearly indicates that XP is quite a natural way of developing agents, which works with people's instincts and not against them.

These observations about XP are consistent with industrial reports on the development of "conventional" software [177]. However, care must be taken to simply generalize those positive reports to the domain of agents, because multi-agent systems have specific properties that should be reflected in the methodology. Therefore, a theoretical assessment of whether and when XP is suitable for multi-agent systems should be derived from the specific attributes of agent systems. Here are some reasons why I believe that Extreme Programming is suitable for agent development.

- The single agents are typically rather small and loosely-coupled systems, which solve their tasks in relative autonomy. As a consequence, writing automated test cases is quite easy for agents, because the single agents have a small, finite number of interaction channels with external system units.

- Simple solutions, like those suggested by XP, appear to be sufficient for most agents, because the agents themselves are quite small. Even if an agent evolves into a performance or quality bottleneck after a series of refactorings, it is possible to completely rewrite it from the scratch (following the XP value of Courage) without risking the functionality of the overall system.

- Agents, like those in our clinical scenario, represent various types of human process participants with individual goals and knowledge. In order to map those multi-facet viewpoints onto a consistent system, people from the domain must be involved very closely into the development process – and they must communicate with each other. In XP processes, domain experts and end users are neatly integrated with rapid feedback, fluent communication, and in the requirements planning cycle.

- The close involvement of real users allows to clarify misunderstandings and terminology much faster than in engineering approaches, in which users are comparably excluded from the analysis and design phases.

- It appears to be very probable that requirements within multi-agent systems will change. The agent modeling process itself often produces new knowledge about potential agent scenarios, and the self-observation performed during analysis of the existing work processes into which agents are to be introduced can lead to new insights [152].

- Systematic engineering approaches are difficult to apply to agents, because the resulting analysis and design models are often based on wrong assumptions. When agents are intended to solve tasks on behalf of human stakeholders with all their individual beliefs, desires, and intentions, the humans are required to transparently

expose their daily practice. However, this "practice necessarily operates with deception" [152], so that mental processes can often only be incrementally translated into agent programs.

- The autonomy and "intelligence" of agents allows to characterize many multi-agent systems as distributed knowledge-based systems. As shown in chapter 2, such systems are best designed in an evolutionary style with rapid feedback.
- Frequently, agents are experimental systems, for which bureaucratic processes are too rigid.

Despite these many arguments in favor of XP for agents, our results must be put into perspective with the context of our specific project. We made some assumptions that do not hold in all projects. In particular, there was complete commitment to XP both within the project management (i.e., I were very motivated to proof the benefits of XP), and among the students. The students may not be representative developers, because they voluntarily chose to participate in the course and were correspondingly motivated and open-minded. Furthermore, the project was very short, so that the long-term impact of the simple agent designs could not be seriously tested. Future XP courses will continue the AGIL project to allow for a rather objective assessment of XP.

8.3. Summary

In this chapter I have reported on my experience with and without Extreme Programming in the two research projects ANIS and AGIL. In the ANIS project, I have evaluated some of the existing Knowledge Engineering approaches, in particular the Protégé system. The difficulties with the transitions between the resulting knowledge models and the remaining system led to the development of the KBeans Round-Trip Engineering approach for knowledge modeling, and the KBeansShell tool. Both proved to be technically very flexible and allowed me to develop a knowledge-base system for anesthesia in a very evolutionary style. As a byproduct of ANIS, I developed a generic rule engine which indicated that the separation of problem-solving and domain knowledge – as advocated by Knowledge Engineering theory – is often neither feasible nor beneficial. In our case, the object-oriented integration of structure and behavior, and the reuse of models through inheritance, proved to work quite well for knowledge modeling.

A major lesson learned during ANIS was the presence of the knowledge acquisition bottleneck in tacit domains like medicine. The spatial distribution between the project partners was a main obstacle to building appropriate knowledge models. The later project AGIL showed that joint knowledge modeling by domain and technology experts can significantly reduce the knowledge acquisition problems. This strengthens the theoretical foundation of agile methodologies, which stress the role of people and communication.

In the AGIL project, I had the opportunity to apply, evaluate, and improve the approach started in ANIS. AGIL showed that KBeans and KBeansShell technologies are quite

efficient for (process) knowledge modeling. KBeans was used for the definition of a process modeling metamodel, and for the implementation of the ontologies of agent messages. The KBeansShell platform allowed to incrementally develop the AGILShell process modeling tool, which enables domain experts to build complex clinical process models, even without technical assistance. The AGILShell will be used in future process modeling projects as well.

The student course in which a large part of the AGIL process model was implemented has provided strong evidence that Extreme Programming and XP.K can be a very attractive alternative to conventional engineering methodologies for agents. The XP.K practices were naturally and eagerly adopted by the students and led to a relatively large system within only five days of programming. In particular the practices of Pair Programming, Testing, and On-Site Domain Expert proved to be very valuable.

While these two case studies have covered only a small section of the domain of knowledge-based systems, in particular multi-agent systems, the following chapter will broaden the view and provide a discussion of the strengths and weaknesses of XP.K in general.

9. Discussion and Conclusion

The preceding four chapters have described how Extreme Programming can be applied to knowledge-based systems. Chapter 5 has provided an overview of the values, principles, and practices of XP.K, and how they are expected to fit together to a consistent methodology. This expectation was based on some assumptions that need further discussion. Now, after chapters 6 and 7 have provided details on languages and tools, and chapter 8 has reported on case studies, it is possible to discuss the critical issues and weaknesses of XP.K.

Note that XP.K is based on Extreme Programming (XP), so that the success of XP.K depends on the success of XP. However, XP is being controversially discussed among academics and people from industry. In a recent technical opinion article, McCormick [120] argues that the whole software community is divided into two opposing belief camps. The group of "programmers" believes that code is easy to change, verbal communication is enjoyable, the code is the design, and that informal requirements suffice. On the other side of the battle line is the group of "software engineers" who believe exactly the opposite. The "battle" is far from being decided, since both groups are able to bring forth valid arguments and practical evidence in favor of their approaches. However, with the rapidly growing interest in XP, it can be expected that there will be a mutual fertilization between the fields. "Even if XP is best suited only to certain projects, it ought to be one of the tools in our bag of tricks" [120].

In this chapter, I will face critical views of XP.K by providing answers to the frequently asked questions about its single practices (section 9.1). Although this will include many arguments for and against XP in general, a comprehensive discussion of XP is beyond the scope of this document. Since some of the arguments against XP.K can not be refuted, it is necessary to make the limits of its application domain explicit. I will for that purpose derive a list of criteria that a knowledge-based system project should fulfill to make XP.K an option (section 9.2). Then, I will point at open issues and potential future work in the area (section 9.3). Finally, I will summarize the main results of this document in a conclusion (section 9.4).

9.1. Discussion

Extreme Programming challenges our basic assumptions about the software development process. Its reliance on simplicity and lack of structure appear counter-intuitive because they contradict the common doctrine taught at university courses [47]. Software engineers

are trimmed towards foresight and tend to build complex designs where XP suggests simple ones. At the same time, XP is unlike plain hacking because it assumes a strong discipline and rigor during coding. Doing XP therefore requires a lot of mental change in the developers. Those who are used to hacking have to get used to testing. Those who are used to designing have to get used to the Planning Game with simple story cards. Those who are used to lonely work have to get used to partnering. Since XP heavily relies on the people involved in the project, i.e. people are less replaceable than in well-structured systematic approaches, it is necessary to *educate* employees and students appropriately. As stated above, software developers should be knowledgeable in both systematic and agile worlds. The increasing availability of XP training courses and the integration of XP techniques into university courses (e.g., [106]) will improve this situation. These types of education must stress social capabilities.

By the way, I argue that XP can activate a large part of the industry's potential workforce and thus reduce the shortage of skilled personnel. In particular, XP might appeal to the highly under-represented group of female IT experts, since women are typically ascribed to better communication and social skills than their male colleagues [192].

The major philosophical question-mark behind XP.K is whether its lightness is sufficient for complex projects like knowledge-based systems, which are traditionally developed with systematic engineering approaches. The major difference between those two philosophies can be summarized by the amount of resources assigned to up-front analysis and design. My main argument against putting large efforts into analysis and design was detailed in chapter 2, where I suggested that requirements in knowledge-based systems are too unstable and tacit to be formally specified in advance. The resulting maintenance problems of the various modeling artifacts are the main argument against systematic engineering approaches. My thesis is that flexible, feedback-driven processes are often a better alternative. Feedback from prototypes is also valuable, because users and domain experts can be involved into the process much better.

The following paragraphs examine frequently mentioned critics to the single practices of XP and XP.K.

Planning Game and project management

The practice of the Planning Game suggests that developers and domain experts jointly determine the next feature to be developed with story cards. A disadvantage of this rather spontaneous process is that it is barely possible to provide the management with long-term cost estimates. Managers and customers usually do not think iteratively, but expect a clear plan with well-defined outcomes. In XP projects, it is hard to estimate when which features will be delivered. A fairly complete set of (static) user stories would be needed prior to development. Producing such a set is typically only ascribed to systematic engineering approaches with a comprehensive requirements analysis up-front. However, surveys of industrial projects [13] suggest that traditional engineering processes still suffer from massive cost and time overruns, so that their potential strength might not apply

in all types of projects. I argue that a flexible process with little pre-planning is often better suited for knowledge-based systems, because of its support for feedback and change. Implementing such flexible XP processes, however, requires to adopt different contracting patterns, so that the problem of missing cost estimates can be reduced. Projects could start with smaller budget and shorter time frames, and redefine contracts after each cycle, when prototypical implementations have been exposed to customer feedback.

On a more general level, the relatively unstructured process model and the lack of high-quality design documents in XP projects means that managers have to pass much of their influence and control to the developers. On the positive side this means that XP teams might be much better motivated, because they can actively participate in decision-making and thus have a high degree of autonomy [33]. The downside is that beside different contracting patterns, this requires a *strong commitment* to XP in general and trust in the team's capabilities. While some success stories (e.g., [70, 25, 145, 133, 163, 196, 77, 98]) on XP projects contribute to achieving this commitment more easily, independent comparisons of projects built with both agile and systematic approaches are still missing. XP proponents argue that the permanent availability of operational prototypes grants a much more direct access to project results and allows for better estimates than analysis models that have not been implemented and tested yet. Reports from industry (e.g., [77]) suggest that release planning in XP style can be very dependable because the single user stories are quite accurate to estimate.

Short Releases

XP suggests to follow a rapid, evolutionary prototyping process, so that feedback can be acquired and put back into the system quickly. Leading Knowledge Engineering methodologists support this idea – at least theoretically. According to Schreiber et al. [164], short releases focus knowledge elicitation and interpretation, motivate the expert, and convince the management. Fensel and van Harmelen [58] argue that rapid prototyping reflects the cyclic nature of the knowledge modeling process. Another reason for rapid feedback is that no matter how rigorous the software practices, the software is dependent on external hardware and software components. Defects in these components are simply unpredictable, but the software will be judged on its ability to respond to them.

There is little doubt about the usefulness of short releases and even the most heavyweight methodologies aim at providing feedback quickly. However, the release cycles of XP are considerably shorter than in RUP or CommonKADS. XP.K supports short releases by fusing high-level knowledge models with (Java) classes that are immediately executable. The disadvantages of this approach are discussed below in the context of Round-Trip Engineering.

Pair Programming / Pair Modeling

The efficiency of Pair Programming is certainly one of the most controversial aspects of XP, because having two people at only one machine might appear to be a waste of re-

sources. In controlled experiments [38], Pair Programming has shown to be very efficient in terms of costs, satisfaction, design quality, continuous review, learning, and team building. Although the pairs did not perform twice as fast as comparable single programmers, the resulting code was of better quality, so that future costs for maintenance were reduced. Griffin [77] also reports that the quality of developer output increased significantly for those teams programming in pairs. One reason for the success of Pair Programming is that it is typically difficult to think at high level at the same time as programming. The synergy between partners can fill an individual's weaknesses [193]. Additionally, two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short, they don't check e-mail messages or favorite web pages, they don't waste each others time. All these observations are consistent with my own experience in the XP course. In knowledge-based systems, many domain experts will find pair modeling very natural. For example, medical schools have a philosophy of "see one, do one, teach one" which supports learning from each other. Other positive reports on Pair Programming can be found in [134, 197]. The frequently mentioned claim that Pair Programming requires additional resources is therefore not true – at least in most cases.

However, pairing raises some requirements on the team. First, team size should be even. Second, teams should consist of at least six programmers, because there is otherwise too little diversification [16]. Third, the teams should not be distributed to multiple sites, so that outsourcing development and international teams are obstructed. However, Kircher et al. [99] describe technological replacements of face-to-face communication that have shown to enable distributed XP. Forth, room planning may be difficult in static office locations. Fifth, not all people are suitable for Pair Programming: Some prefer competition over collaboration and might not be able to program "egolessly". Others might not have appropriate language skills due to their native tongue. Some people simply may not have compatible wavelengths. On the other hand side, Pair Programming might attract rather verbal people, who do not enjoy working in isolation. Anyway, community-building activities like the Shared Symbol Grounding proposed in XP.K and an enjoyable, social working environment are essential in practice.

Coding Standards / Modeling Standards

Project-wide coding standards are an essential means of reducing the cost of change and of implementing Pair Programming. In the Java world, semi-official coding standards exist. For knowledge-based systems, comparable standards are not globally available. Some domains have standardized terminologies and ontologies (e.g., the SNOMED medical terms [150]). The main obstacle to putting these standards into practice is that a rigorous discipline is required among developers. Collective Code Ownership with frequently changing pairs will automatically level minor differences in style. In the Java world, modern IDE's like *IntelliJ* [86] provide features that automatically reformat source code according to predefined standards.

Collective Ownership

In XP.K projects, every developer or domain expert is allowed to make changes to any module. This may result in psychological problems, because developers who have put their whole energy into a certain piece of a model may defend every single part of it [47]. Individualists need to learn how to respect the others' models and to agree that others change theirs too. Another issue with Collective Ownership is that no single team member can be made responsible for errors. For those reasons, a strong sense of community is required in the team and developers need to be convinced that Collective Ownership adds value to the project. Furthermore, automated unit tests and constraint checks must be maintained consequently, so that changes can not accidentially damage other parts.

On-Site Customer / On-Site Domain Expert

An XP.K team involves one or more domain experts, who are available for questions at any time. However, one might object that a domain expert (e.g., a medical doctor) is much too expensive to be in nothing but stand-by-mode, while the programmers are doing their job. The simplest solution to this, of course, is to perform domain modeling and the remaining development tasks in parallel, with all groups of developers interacting with the others when needed. In the AGIL project, we had no problem with this issue, because our domain expert was able to do his normal work while the developers were programming. If this is not possible, the team will necessarily find compromises, e.g. regular meetings every few days. Otherwise, the system might fail to meet the scheduled goals, because developers waste resources by implementing the wrong features at the wrong time. An open-minded management is needed, encouraging domain experts to leave their daily routine. In our work on knowledge-based systems for anesthesia, we found many clinicians were very motivated to contribute to system development, because they were annoyed by the lack of IT infrastructure in their hospital and recognized this as a chance to learn new technology on the leading edge. In conjunction with the XP Planning Game, the domain experts feel that they are truly in control, and that they would receive what they needed when they needed it [77]. However, in some situations, the On-Site Customer is not possible, because a single representative of all customer types does not exist. Note that being on-site is not an absolute requirement, but will typically speed development enormously [92].

Metaphor / Shared Symbol Grounding / Joint Ontology Design

A Shared Symbol Grounding is necessary to enable collaboration between the various developer groups, in particular for Joint Ontology Design. Due to their diverse educational background, it may be questionable whether information technology experts and domain experts will be able to find a shared grounding for abstractions like ontology classes. Industrial experience [77] suggests that XP has the ability to tear down cultural barriers between developers and domain experts. XP has also shown to be useful for knowledge transfer [9]. Our own experience suggests that it is well possible to build a common understanding between domain and computer experts, but we only had to collaborate with technophile

domain experts who already had basic computer knowledge. In knowledge-based systems, the main link between domain and computer experts is the domain ontology. Compared to system implementation and knowledge acquisition, ontology design consumes relatively little time. Therefore, resources should be spent on taking care that the ontology concepts are well-understood by all project participants. In the AGIL project, the Joint Ontology Design was very successful, because we quickly found a sustainable metamodel that was both simple and sufficiently formal.

Simple Design / Simple Knowledge Model / Refactoring

XP.K suggests to build simple models to reduce the investment in potentially wrong or superfluous solutions. The price for simplicity is that refactoring must be performed consequently to improve code design. Languages like Java with KBeans are designed to reduce the cost of refactoring and change, because their object-oriented paradigm helps to manage complexity and facilitates communication (subsection 3.1.1). KBeans has the additional benefit of allowing to mix imperative with declarative elements, so that developers can implement simple solutions that would take much longer to express with traditional ontology languages. XP.K suggests to employ generic tools, like KBeansShell, which allow to experiment with simple solutions without spending much resources if change is necessary. Furthermore, XP.K includes several techniques that make sure that the simple design is not a stupid design. In particular, KBeans ontologies are designed jointly, so that all project stakeholders and roles have to examine the central models from their individual points-of-view.

KBeans was deliberately designed to be simple and misses some of the rather formal features typically available in traditional knowledge representation languages. Jackson [87] argues that the key benefit of object models compared to formal languages is that "an object model is a light-weight formal specification – converting it into a full-fledged formal specification language obscures its simplicity, and eliminates opportunities for analysis that take advantage of its limited expressiveness."

Simplicity is sometimes regarded as an obstacle to *reuse*, because developers might tend to overlook existing standard solutions and waste resources building custom solution instead [47]. Various Java libraries for knowledge-based systems (like those listed in Appendix G) provide generic reasoning services with the explicit goal of being reusable. Extreme Programming teams developing knowledge-based systems should have an overview of these existing libraries. Efforts to categorize Problem-Solving Methods into structured libraries like those around the UPML language [56] may contribute valuable ideas. Note that XP.K is much more general than Knowledge Engineering methods in this regard, because any Java component library is potentially reusable for knowledge-based systems.

Another problem with reuse in XP is that simple models are often not sufficiently flexible to be reused in other projects. XP argues that teams may have to balance whether a certain piece of code may be useful in later projects. Reusable components and frameworks represent long-term investments that only pay off if similar applications are developed over and over again [148]. In Knowledge Engineering, large-scale reuse of domain ontologies

has not proven to work (yet), in particular due to the Interaction Problem (section 4.1). If teams wish to reuse an existing KBeans ontology anyway, they can easily do so by extracting its structure into a UML model and throw away those additional features and methods that are a consequence of simplicity and not reusable.

A greater potential for reuse lies in Problem-Solving Methods. Subsection 8.1.4 has already described the case of a rule engine that was made reusable through refactoring. The question whether refactoring is in general sufficient has not been satisfyingly answered yet in the literature. Experience reports indicate huge benefits of applying refactoring, but there might be equally many unpublished cases in which refactoring was not enough to improve an existing system. Most researchers agree that XP may not be the best choice for the development of very general, reusable components or frameworks, although others report on successful framework development with XP [113].

Round-Trip Engineering / Traveling Light

XP proponents believe that source code, and the documents that can be generated from it (UML diagrams, JavaDoc help files, etc.), are in most cases sufficient for system maintenance. The train of thought behind this is that synchronizing various models is expensive and developers often do not like writing documents that are not directly leading to production code. XP.K and KBeans focus the team on the intended result, namely to deliver an executable system. KBeans supports Round-Trip Engineering with UML and therefore drives the principle of Traveling Light to the extreme. This bidirectional mapping between code and model accommodates non-programmers (domain experts, managers) who are usually unable to understand a system by reading code. By providing different views on the same artifact, KBeans narrows the cognitive gap between mental models and its implementation. Despite this, the proponents of systematic methodologies argue that code and derived models are insufficient to close the gap, because high-level design and analysis documents would be very helpful for conceiving and explaining complex pieces [164]. The only other artifact involved in an XP project beside source code is the collection of story cards, which are supposed to serve as a link between requirements and the implementing classes.

While the KBeans Round-Trip Engineering approach is rapid and agile, it involves the risk that the view of knowledge be determined by the chosen implementation language, resulting in distorted models [164]. Furthermore, the knowledge engineer who implements the system has to perform many tasks at the same time. She has to analyze the given information, as well as to design, implement, and evaluate the system, so that different knowledge aspects have to be considered simultaneously and are therefore mixed [58]. However, Knowledge Engineering methodologies, in which the move from conceptual level to implementation level is delayed until late, suffer from providing little feedback during knowledge acquisition. Thus, domain models have a higher risk of being not implementable, error-prone, or inefficient.

A clear disadvantage of our Traveling Light approach is "that the code – no matter how well documented – does not tell you the problem it solves, it only conveys the so-

lution to the problem" [146]. Instead, the running system is the only documentation of expertise [58]. This makes reuse of intermediate, high-level models as in Requirements Engineering difficult. Furthermore, "the lack of emphasis on documentation within XP does not take into account end-user needs for documentation, including user guides, integration kits, reference texts and fact sheets" [77]. Systematic engineering methodologies that build models on top of each other with traceable cross-references are clearly better in this regard (assuming they are applied correctly). The weakness of Traveling Light surfaces when developer teams change frequently or when all developers are gone after a project. To a certain extent, movement of labor can be absorbed by following Pair Programming consequently, so that new team members can learn from their experienced colleagues and get to see most of the system's architecture through informal communication. Face-to-face communication is certainly much more efficient to integrate new members than letting them browse design models in autonomy. If more or all developers leave, then it is a good idea to ensure that developers leave their code and models in a well-documented state. For example, Kini and Collins [98] report that prior to delivery day, developers created an overview of the design along with some automatically generated UML diagrams, and annotated all of the classes with JavaDoc comments. I could not find conclusive evidence whether this reverse documentation approach is sufficient on a large scale. Some researchers argue that XP's lack of documentation requires stable teams [39].

(Automated) Testing / Constraint Checking

Most practitioners who have introduced automated tests into their code (e.g., [77]) report significant improvements, in particular higher quality, faster integration times, and a higher confidence in the face of changes. This confidence is an important aspect in safety-critical knowledge-based systems, although testing can not prove the absence of errors. The benefits of constraint checking and the related Design-by-Contract have already been covered in chapter 6 and by others [82]. However, the value of writing automated tests appears to be underestimated and few programming courses teach it. As a result, programmers need to rethink and relearn their programming style. In particular, programmers might be reluctant to test their own code and might tend to obnubilate bugs. This problem, however, is faced through Collective Ownership and the strong sense of community built through the intense communication within the XP team. Testing and improving the semantic transparency of code must be supported by the management, so that the developers are supplied with sufficient time to implement tests [16].

Potential pitfalls involved in testing in general (e.g., how to test graphical user interfaces) are sufficiently covered in the literature (e.g., [193, 125]).

9.2. The Application Domain of XP.K

My thesis, as stated in the first chapter, is that a considerable class of small to medium sized knowledge-based systems can be efficiently developed with XP.K. It remains to clarify

this class, because "a one-size-fits-all development process does not exist" [120]. Now after the strengths and weaknesses of XP.K have been identified, it is possible to narrow its application domain and to identify the requirements that a potential XP.K project should fulfill.

A fundamental requirement is the size of the problem. XP is mostly only assumed to work for small to medium sized problems, so that the spontaneous planning process based on user stories is accurate. Furthermore, XP team size should not exceed the threshold up to which a fluent, personal communication is not disturbed. XP will therefore flourish mainly on smaller, less complex projects. Elting and Huber [47] argue that XP is mostly geared for individual, non-generic projects. It is furthermore questionable whether XP should be applied for the development of life-threatening applications, like those found in health care or "Space Shuttles". I have brought forth evidence that XP.K with its support for Unit Tests can well be applied even in safety-critical domains, even if only used for a comprehensive prototyping process that leads to a systematic redesign later. There is hardly any reason why Space Shuttles should be developed in an XP style, because this technology is based on well-defined engineering rules and fixed requirements. "Where the purpose and requirements are clear at the outset, a stage-based approach seems to be more appropriate whereas where no clear purpose has been identified, the evolving prototype model may be more applicable." [94]. Despite this, even Space Shuttle projects can profit from adopting some of the XP practices. For example, the failure diagnosis report of the Mars Climate Orbiter [131] states that a major reason for the failure was bad communication among developers.

Note that XP might be an attractive option for projects that have started with a traditional process but then run into the risk of failure. Such projects can move to a light-weight process, if the source code is brought into the required form (e.g., coding standards, unit tests, and so on). Schuh [165] reports on an actual project that had fallen into developmental disarray, in which a gradual institution of XP facilitated the project's ultimate recovery. Other authors (e.g., [155]) argue that some XP projects should deliberatively be started in a waterfall style to identify user requirements better. Thus, a certain mixture of systematic and agile processes is possible, with a complete analysis and design cycle before an agile implementation.

Implementing XP projects poses some requirements on the people involved. An initial requirement is that the management shows a strong commitment to try XP. Among others, different contracting patterns are required, with shorter pre-planning cycles. Also, projects will profit if employees can be assigned 100 per cent to one project [47]. Furthermore, the customer must show willingness to provide an On-Site Customer. Pair Programming probably enforces most requirements on people, especially an even team size, at least six developers, little spatial distribution, and constraints on room planning. Domain experts need to be open-minded and collaborative. Furthermore, not all people might be suitable for XP, since communication skills are essential. Finally, XP will work best in stable teams. All these issues indicate that XP is particularly attractive for companies that run more than one XP project, so that experience and teams can be nurtured and reused.

Some authors argue that XP is not a complete methodology, because e.g. the roles are ill-defined and XP is not compatible to ISO-9000 quality standards [47]. The very name "Extreme Programming" and the commercialization hype surrounding it repel many people. There is also a strong rejection among professionals who have achieved well-founded knowledge of traditional Software Engineering and fear losing influence. It might therefore be hard to establish XP in existing companies. However, many developers, in particular young IT professionals and students, are very eager to try XP. A recent poll among Java programmers at IBM has shown that 51 percent of Java developers have already tried and loved XP [85]. The supervisors of a practical XP course at a university report that half of their students participated even although they did not require the workshop's credit points [195]. The number of registrations for our own XP course [106] significantly exceeded our expectations. In the near future it should therefore not be too difficult to recruit programmers for XP projects. Existing practitioners, who are not experienced in XP yet, might need to relearn their development style.

Further constraints on the application domain of XP.K arise from its dependency on the simple knowledge representation language, and KBeans in particular. The specific support for KBeans with the freely available KBeansShell tools means that XP.K is most efficiently applicable in Java-based projects. However, Java is becoming more and more widely used [65], in particular in research projects. The simple KBeans ontology language means that certain types of reasoning are not as accessible as with traditional knowledge representation languages. As a consequence, XP.K may not be the best choice when legacy ontologies and components in other languages exist and the team is unwilling to move to pure Object-Orientation.

The following criteria summarize the limits and bottlenecks of XP.K:

- *Size*: XP.K is optimized for small to medium sized projects.
- *Education*: XP.K requires people have communication and testing skills.
- *Management*: XP.K requires management commitment and adequate contracts.
- *Domain expert*: XP.K works best if a motivated domain expert is on-site.
- *Reuse*: XP.K is optimized for one-of-a-kind systems.

9.3. Open Issues and Potential Future Work

A consequence of the discussion above is the following list of open issues and potential future work that may contribute to make XP and XP.K more widely used.

- *More case studies*, in particular independent comparisons, are needed to provide further evidence whether XP works up to its promises, and to identify criteria which dissuade from using XP.

- *A strengthening of the theoretical foundations* of XP, as a decision aid for managers and to guide research and education. Theoretical foundations should aim at bridging the unnecessary gap between the creeds of systematic and agile methodologists. With its reliance on story cards, XP should establish a link to Requirements Engineering approaches which focus on identifying the "right" user stories and scenarios.
- *Better education* both at universities and industry is needed to spread knowledge of XP and practices like Unit Testing and Pair Programming in particular.
- *Methods for building a Shared Symbol Grounding*, especially between domain and computer experts, are required to ensure that collaboration is fruitful.
- *A catalogue of Problem-Solving Methods in Java* and other practically used languages are needed to simplify selection and use of existing reasoning components.
- *The distribution of KBeans as a standard* in the Java world should be attempted. The Java Community Process (JCP) allows user groups to make proposals for extensions to the Java language. I received several very encouraging emails that asked me to initiate a JCP to establish the KBeans facet mechanism in the core language. Formal issues (JCPs are liable for costs) have prevented this so far.
- *An implementation of KBeans in C#* as a consequence of the future demand in this language. This is fairly straight-forward and can be based on the design and implementation described in this document.

9.4. Conclusion

In this document, I have attempted to contribute to make the development of knowledge-based systems more efficient by exploring the benefits and limitations of XP.K, an agile development approach based on Extreme Programming. Backed by the observation that the development of knowledge-based systems must particularly support evolution and collaboration, I have shown that traditional methodologies from Software Engineering and Knowledge Engineering are often inefficient, because they are relatively inflexible and heavy-weight. As an alternative, I have introduced XP.K, which is based on the four values of Simplicity, Community, Feedback, and Courage, and applies object-oriented Round-Trip Engineering to knowledge modeling. I have provided details on how XP.K can be implemented by describing and discussing its principles and practices, by showing how standard technology from Object-Orientation can be used for knowledge modeling, by presenting efficient tool support, and by describing case studies in which essential elements of XP.K have been successfully applied.

During my work on XP.K, I have developed a comprehensive and powerful framework of supporting Java libraries and tools. The KBeans approach for representing semantic constraints on Java objects has demonstrable benefits not only for knowledge modeling but also for component development in general. The large and freely available KBeansShell

tool platform has lead to considerable productivity gains in various projects, because its generic knowledge modeling widgets can be reused for any metamodel. Using the concepts and software architectures presented in this document, this framework can be easily ported to other object-oriented languages with introspection mechanisms.

My main result is that Extreme Programming is a very attractive option for small to medium sized projects in which requirements are uncertain and the project setting allows to implement Pair Modeling and related practices, if the management shows a commitment to agile processes and is able to provide a motivated on-site domain expert. XP.K can be combined with traditional waterfall-based approaches by performing a comprehensive analysis and design prior to an agile iteration cycle. Furthermore, several of the practices, languages, and tools of XP.K are worth being employed in systematic engineering projects as well, in particular Unit Testing and Pair Modeling should be taught as standard tools in the developers' toolbox.

The application of Extreme Programming to knowledge-based systems may therefore be a valuable contribution to both Software Engineering and Knowledge Engineering. The proponents of agile, light-weight Software Engineering methodologies should take a closer look at knowledge-based systems (and in particular multi-agent systems) since these are a fruitful application domain. Inversely, Knowledge Engineering approaches can profit from integrating Extreme Programming techniques. It is my hope that this document will contribute to a cross-fertilization between those domains and thus make the development of knowledge-based systems more efficient.

Bibliography

[1] The Agile Alliance. Manifesto. http://agilealliance.org, 2001.

[2] Sinan Si Alhir. The object-oriented paradigm. http://home.earthlink.net/~salhir/theobjectorientedparadigm.html, 1998.

[3] Scott W. Ambler. Agile modeling homepage. http://www.agilemodeling.com, 2001.

[4] Jürgen Angele, Dieter Fensel, Dieter Landes, and Rudi Studer. Developing knowledge-based systems with MIKE. *Journal of Automated Software Engineering*, 5(4), 1998.

[5] Jürgen Angele, R. Perkuhn, Rudi Studer, A. Oberweis, G. Zimmermann, B. Dellen, F. Maurer, G. Pews, and W. Stein. Abschlußbericht der GI-Arbeitsgruppe "Vergleichende Analyse von Problemstellungen und Lösungsanstzen in den Fachgebieten Information Systems Engineering, Software Engineering und Knowledge Engineering". Report of the EMISA-Forum, http://sunsite.informatik.rwth-aachen.de/Societies/GI-EMISA/Forum, 1997.

[6] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[7] Stefan Bamberger, Ute Gappa, Franziska Klügl, and Frank Puppe. Komplexitätsreduktion durch grafische Wissensabstraktion. In *Proc. of Expertensysteme–97*, Bonn, Germany, 1997.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999.

[9] Luigi Benedicenti and Raman Paranjape. Using Extreme Programming for knowledge transfer. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001. http://www.xp2001.org/xp2001/conference/papers/Chapter18-Benedicenti+al%ii.pdf.

[10] Richard Benjamins, Dieter Fensel, Christine Pierret-Golbreich, Enrico Motta, Rudi Studer, Bob Wielinga, and Marie-Christine Rousset. Making knowledge engineering technology work. In *Proc. of the 9th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE-97)*, Madrid, Spain, 1997.

[11] Sondes Bennasri and Carine Souveyet. Component reuse and component-based methodologies. In *International Workshop on Mechanisms for Enterprise Integration: From Objects to Ontologies (MERIT 2001), at the ECAI 2001*, Budapest, Hungary, 2001.

[12] Ross Bentley. Java skills demand hits all-time high. *Computer Weekly*, August 12, 2000.

[13] Scott Berinato. The secret to software success. *CIO Magazine*, July 1, 2001. `http://www.cio.com/archive/070101`.

[14] Michael Blaha and William Premerlani. A catalog of object model transformations. In *Proc. of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, 1996.

[15] Mercedes Blázquez, Mariano Fernández, Juan Manuel Garcia-Pinar, and Asunción Gómez-Pérez. Building ontologies at the knowledge level using the Ontology Design Environment. In *Proc. of the Knowledge Acquisition Workshop (KAW-98)*, Banff, Canada, 1998.

[16] Andreas Bleul. Programmier-Extremisten. *c't Magazin für Computertechnik*, 2001(3):182–185, 2001.

[17] Daniel G. Bobrow, Sanjay Mittal, and Mark Stefik. Expert systems: Perils and promise. *Communications of the ACM*, 29(9):880–894, 1986.

[18] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[19] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 5:61–72, 1988.

[20] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[21] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.

[22] Pim Borst, Edwin van der Wal, and Piet-Hein Speel. Developing a library of domain models using commercial tools. In *Proc. of the Knowledge Acquisition Workshop (KAW-99)*, Banff, Canada, 1999.

[23] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 3:45–54, 1995.

[24] Tom Bylander and Balakrishnan Chandrasekaran. Generic tasks for knowledge-based reasoning: The "right" level of abstraction for knowledge acquisition. In B. R. Gaines and J. H. Boose, editors, *Knowledge Acquisition for Knowledge-Based Systems, volume 1*, pages 65–77. Academic Press, London, 1988.

[25] C3 Team. Chrysler goes to "extremes". *Distributed Computing Magazine*, October:24–28, 1998.

[26] Colin Cadas and Nikos Parthenios. Catalyst – use of CommonKADS methodology in knowledge based system development. European Systems and Software Initiative (ESSI) Report No. 10327, `http://www.esi.es/ESSI/Reports/All/10327`, 1996.

[27] Martine Callot. Methodology and tools oriented to knowledge based engineering applications. MOKA Public Report No. 2, `http://www.kbe.coventry.ac.uk/MOKA`, 1999.

[28] Castor. Castor. `http://www.castor.org`, 2001.

[29] Balakrishnan Chandrasekaran, Todd R. Johnson, and Jack W. Smith. Task structure analysis for knowledge modeling. *Communications of the ACM*, 35(9):124–137, 1992.

[30] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proc. of the AAAI-98*, Madison, WI, 1998.

[31] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. of the 10th Meeting of the Information Processing Society of Japan*, pages 7–18, 1994.

[32] Paolo Ciancarini and Michael Wooldridge, editors. *Agent-Oriented Software Engineering*. Springer-Verlag, Berlin, Heidelberg, New York, 2001.

[33] Norbert Claudel. Tagungsbericht vom 8. workshop der fachgruppe 5.11 "vorgehensmodelle für die betriebliche anwendungsentwicklung" der gi. `http://www.vorgehensmodelle.de/Ws2001/claudel.htm`, 2001.

[34] Peter Coad and Edward Yourdan. *Object-Oriented Analysis, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[35] Alistair Cockburn. The methodology space. Humans and Technology Technical Report TR.97.03, 1997. `http://members.aol.com/acockburn/papers/methyspace/methyspace.htm`.

[36] Alistair Cockburn. Reexamining the cost of change curve, year 2000. XProgramming.com online magazine, `http://www.xprogramming.com/xpmag/cost_of_change.htm`, 2000.

[37] Alistair Cockburn. *Agile Software Development*. Addison Wesley Longman, Reading, MA, 2001.

[38] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. Humans and Technology Technical Report 2000.01, 2000. `http://members.aol.com/humansandt/papers/pairprogrammingcostbene/pairpr%ogrammingcostbene.htm`.

[39] Jens Coldewey. Über sieben Brücken musst Du geh'n: Eine Kritik am Software Engineering. *OBJEKTspektrum*, 1:22–27, 2001.

[40] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[41] Oscar Corcho and Asunción Gómez Pérez. Evaluating knowledge representation and reasoning capabilities of ontology specification languages. In *Proc. of the ECAI'00 Workshop on Applications of Ontologies and Problem-Solving Methods*, Berlin, Germany, 2000.

[42] Robin Cover. XML and semantic transparency. `http://www.oasis-open.org/cover/xmlAndSemantics.html`, 1998.

[43] Stephen Cranefield and Martin Purvis. UML as an ontology modeling language. In *Proc. of the IJCAI-99 Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

[44] Alan M. Davis. Object-oriented requirements to object-oriented design: An easy transition? *The Journal of Systems Software*, 30:151–159, 1995.

[45] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal: UML shortcomings for coping with round-trip engineering. In *Proc. of the Second Int. Conf. on The Unified Modeling Language (UML'99)*, Fort Collins, CO, 1999.

[46] Arthur Duineveld, Roeland Stoter, Marcel Weiden, Bisente Kenepa, and Richard Benjamins. Wondertools? A comparative study of ontological engineering tools. *International Journal of Human-Computer Studies*, 52(6):1111–1133, 2000.

[47] Andreas Elting and Walter Huber. Schnellverfahren - Mit Extreme Programming immer im Plan? *c't Magazin für Computertechnik*, 2001(3):186–191, 2001.

[48] Enhydra. Zeus. `http://zeus.enhydra.org`, 2001.

[49] Oliver Enseling. iContract: Design by contract in Java. JavaWorld (February), `http://www.javaworld.com/javaworld/jw-02-2001`, 2001.

[50] Evans Data Corporation. North American developer survey. Overviews available at `http://www.evansdata.com`, 2001.

[51] FAW Ulm. KBeans Homepage. `http://www.faw.uni-ulm.de/kbeans`, 2000.

[52] Dieter Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, Berlin, 2000.

[53] Dieter Fensel, Jürgen Angele, and Rudi Studer. The knowledge acquisition and representation language KARL. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):527–550, 1998.

[54] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Stefan Decker, Michael Erdmann, and Michel C. A. Klein. OIL in a nutshell. In *Proc. of the 12th Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW-2000)*, Juan-les-Pins, France, 2000.

[55] Dieter Fensel and Enrico Motta. Structured development of problem solving methods. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[56] Dieter Fensel, Enrico Motta, V. Richard Benjamins, Stefan Decker, Mauro Gaspari, Rix Groenboom, William Grosso, Mark Musen, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The Unified Problem-Solving Method Development Language UPML. ESPRIT Projekt 27169 IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web, Deliverable 1.1, Chapter 1, 1999.

[57] Dieter Fensel and Remco Straatman. The essence of problem-solving methods: Making assumptions to gain efficiency. *International Journal of Human Computer Studies*, 48(2):181–215, 1998.

[58] Dieter Fensel and Frank van Harmelen. A comparison of languages which operationalise and formalise KADS models of expertise. *The Knowledge Engineering Review*, 9(2), 1994.

[59] FIPA. The Foundation for Intelligent Physical Agents specification. `http://www.fipa.org`, 2000.

[60] FIPA-OS. FIPA-OS homepage. `http://fipa-os.sourceforge.net/`, 2001.

[61] Brian Foote and Joe Yoder. Metadata and Active Object-Models. In *OOPSLA-98 Workshop on Metadata and Active Object-Models*, Vancouver, Canada, 1998.

[62] Charles L. Forgy. RETE: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[63] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley Longman, Reading, MA, 1999.

[64] Martin Fowler. *UML Distilled, Second Edition.* Addison Wesley Longman, Reading, MA, 1999.

[65] Peter Galli. Study: Java to overtake C/C++ in 2002. `http://www.zdnet.com/eweek/stories/general/0,11011,2804967,00.html`, 2001. Based on a Study by Evans Data Corp.

[66] Erich Gamma and Kent Beck. JUnit: A regression testing framework. `http://www.junit.org`, 2000.

[67] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addision-Wesley, Reading, MA, 1995.

[68] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I.* World Scientific Publishing Company, NJ, 1993.

[69] Michael R. Genesereth and Richard Fikes. Knowledge Interchange Format, version 3.0, reference manual. Technical Report, Logic-92-1, Computer Science Department, Stanford University, 1992.

[70] Frederik D. George and Robert L. Billington. Extreme Java: Experiences in applying Extreme Programming to a Java-based project. In *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA-99)*, Denver, CO, 1999.

[71] Susan Gerhart, Dan Craigen, and Ted Ralston. Observations on industrial practice using formal methods. In *Proc. of the 15th Int. Conf. on Software Engineering*, Baltimore, MD, 1993.

[72] Allen Ginsberg. Knowledge base reduction: A new approach to checking knowledge bases for inconsistency and redundancy. In *Proc. of the AAAI 1988*, pages 585–589, 1988.

[73] Adele Goldberg. Information models, views and controllers. *Dr. Dobbs Journal*, 9(July):54–61, 1990.

[74] Asunción Gómez-Pérez and V. Richard Benjamins. Overview of knowledge sharing and reuse components: Ontologies and problem-solving methods. In V. R. Benjamins, B. Chandrasekaran, A. Gómez-Pérez, N. Guarino, and M. Uschold, editors, *Proc. of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods*, Stockholm, Sweden, 1999.

[75] Ian Graham, Brian Henderson-Sellers, and Houman Younessi. *The OPEN Process Specification*. Addison-Wesley, Reading, MA, 1997.

[76] Christian Greiner, Holger Knublauch, Martin Sedlmayr, and Thomas Rose. Representing and processing various knowledge types in an anaesthesia information system. *Journal of Clinical Monitoring and Computing*, 14(7/8):541–553, 1998.

[77] L. Ann Griffin. A customer experience: Implementing XP. In *Proc. of the XPUniverse conference*, Raleigh, NC, 2001. `http://www.xpuniverse.com/xpuPapers.htm`.

[78] William Grosso, Henrik Eriksson, Ray Fergerson, John Gennari, Samson Tu, and Mark Musen. Knowledge modeling at the millennium (the design and evolution of Protégé-2000). In *Proc. of the Knowledge Acquisition Workshop (KAW-99)*, Banff, Canada, 1999.

[79] Tom Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[80] Michael Grüninger and Mark S. Fox. Methodology for the design and evaluation of ontologies. In *IJCAI-95 Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995.

[81] Stevan Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

[82] Hasko Heinecke and Christian Noack. Integrating Extreme Programming and contracts. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001. `http://www.xp2001.org/xp2001/conference/papers/Chapter6-Heinecke+alii.p%df`.

[83] James A. Highsmith. *Adaptive Software Development – A Collaborative Approach to Managing Complex Systems*. Dorset House, New York, 2000.

[84] James A. Highsmith. Extreme Programming. *e-business application delivery*, 12(2), 2000.

[85] IBM developerWorks. Java poll results. `http://www-106.ibm.com/developerworks/library/java-pollresults/xp.html`, 2000.

[86] IntelliJ Software. IntelliJ IDEA 2.0. `http://www.intellij.com`, 2001.

[87] Daniel Jackson. Object models as heap invariants. In C. Morgan and A. McIver, editors, *Essays on Programming Methodology*. Springer Verlag, 2000. `http://sdg.lcs.mit.edu/~dnj`.

[88] Daniel Jackson and Martin Rinard. The future of software analysis. In Anthony Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[89] Ivar Jacobson, Grady Booch, and James E. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999.

[90] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.

[91] Matthias Jarke. Requirements tracing. *Communications of the ACM*, 41(12):32–36, 1998.

[92] Ronald E. Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, Reading, MA, 2001.

[93] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Int. Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[94] Dean Jones, Trevor Bench-Capon, and Pepijn Visser. Methodologies for ontology development. In *Proc. of the 15th IFIP World Computer Congress*, Vienna, Austria and Budapest, Hungary, 1998.

[95] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In *Proc. of the Second International Conference on Metalevel Architectures and Reflection (Reflection'99)*, Saint-Malo, France, 1999.

[96] Peter D. Karp, Vinay K. Chaudhri, and Jerome F. Thomere. XOL: An XML-based ontology exchange language. Technical Note 559, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, `http://www.ai.sri.com/~pkarp/xol/`, 1999.

[97] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[98] Natraj Kini and Steve Collins. Steering the car: Lessons learned from an outsourced XP project. In *Proc. of the XPUniverse conference*, Raleigh, NC, 2001. `http://www.xpuniverse.com/xpuPapers.htm`.

[99] Michael Kircher, Prashant Jain, Angelo Corsaro, and David Levine. Distributed eXtreme Programming. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001. `http://www.xp2001.org/xp2001/conference/papers/Chapter16-Kircher+alii.p%df`.

[100] Holger Knublauch. Object-oriented software development applied to the development of a probabilistic knowledge-based system. Master's thesis, University of Ulm, Germany, 1997. `http://www.faw.uni-ulm.de/kbeans/knublauch`.

[101] Holger Knublauch. KBeans: Implementing semantic transparency for components and domain models. Technical Report FAW-TR-01001, FAW Ulm, 2001. `http://www.faw.uni-ulm.de/kbeans`.

[102] Holger Knublauch. Extreme programming of multi-agent systems. In *Proc. of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, Bologna, Italy, 2002.

[103] Holger Knublauch and Thomas Rose. Round-trip engineering of ontologies for knowledge-based systems. In *Proc. of the Twelfth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Chicago, IL, 2000.

[104] Holger Knublauch and Thomas Rose. Werkzeugunterstützte Prozessanalyse zur Identifikation von Anwendungsszenarien für Agenten. In *Proc. Verbundtagung Verteilte Informationssysteme auf der Grundlage von Objekten, Komponenten und Agenten (vertIS 2001)*, Bamberg, Germany, 2001.

[105] Holger Knublauch, Thomas Rose, and Martin Sedlmayr. Towards a multi-agent system for pro-active information management in anesthesia. In *Proc. of the Agents-2000 Workshop on Autonomous Agents in Health Care*, Barcelona, Spain, 2000.

[106] Holger Knublauch, Martin Sedlmayr, Helmuth Partsch, and Franz-Josef Radermacher. Extreme Programming with Java. Practical course at the University of Ulm, 2001. `http://www.faw.uni-ulm.de/kbeans/knublauch/teaching/xpjava`.

[107] Holger Knublauch, Martin Sedlmayr, and Thomas Rose. Design patterns for the implementation of constraints on JavaBeans. In *Proc. of the NetObjectDays2000*, Erfurt, Germany, 2000.

[108] Holger Knublauch, Martin Sedlmayr, and Thomas Rose. Knowledge-based decision support in an anaesthesia information system. *Journal of Clinical Monitoring and Computing*, 16(2):141–158, 2000. From the Proc. of the ESCTAIC Annual Meeting, Glasgow, UK.

[109] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, Reading, MA, 1999.

[110] Philippe Kruchten. From waterfall to iterative lifecycle – a tough transition for project managers. Whitepaper, Rational Software Corporation, 2000.

[111] Jürgen Lind. *Massive: Software Engineering for Multiagent Systems*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[112] Mikael Lindvall and Ioana Rus. Process diversity in software development. *IEEE Software*, 17(4), 2000.

[113] Martin Lippert, Stefan Roock, Henning Wolf, and Heinz Züllinghoven. JWAM and XP: Using XP for framework development. In *Proc. of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Italy, 2000. `http://www.jwam.de/product/pdfs/jwamandxp.pdf`.

[114] Chamond Liu. *Smalltalk, Objects, and Design.* Manning Publications, Greenwich, CT, 1996.

[115] Mariano Fernández López and Asunción Gómez Pérez. Overview of methodologies for building ontologies. In *Proc. of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods*, Stockholm, Sweden, 1999.

[116] Rob MacGregor and Mark H. Burstein. Using a description classifier to enhance knowledge representation. *IEEE Expert*, 6(3):41–46, 1991.

[117] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Object-Oriented Programming in the BETA Programming Language. `http://www.daimi.au.dk/~olm/PUB`, 1999.

[118] Pattie Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. Elsevier Science Publishers, Amsterdam, The Netherlands, 1988.

[119] Steve McConnell. The power of process. *IEEE Computer*, May, 1998.

[120] Michael McCormick. Programming extremism. *Communications of the ACM*, 44(6), 2001.

[121] Brett McLaughlin. Validation with Java and XML Schema. JavaWorld (September-December 2000) `http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html`, 2000.

[122] Deb Melewski. UML gains ground. *Application Development Trends*, October, 1998. `http://www.adtmag.com/Pub/oct98/id4.asp`.

[123] Tim Menzies. *Principles for Generalised Testing of Knowledge Bases.* PhD thesis, University of New South Wales, 1995.

[124] Tim Menzies and Philip Haynes. The methodology of methodologies; or, evaluating current methodologies: Why and how. In *Proc. of TOOLS Pacific '94*, pages 83–92, 1994.

[125] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice-Hall, Englewood Cliffs, NJ, 1997.

[126] Microsoft Corporation. *C# Language Specifications.* Microsoft Corporation, 2001.

[127] Marvin Minsky. A framework for representing knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, 1975.

[128] Riichiro Mizoguchi, Johan Vanwelkenhuysen, and Mitsuru Ikeda. Task ontologies for reuse of problem solving knowledge. In N. J. I. Mars, editor, *Towards Very Large Knowledge Bases*. IOS Press, 1995.

[129] Katharina Morik. Underlying assumptions of knowledge acquisition and machine learning. *Knowledge Acquisition*, 3, 1991.

[130] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proc. of the Conf. on Human Factors and Computing Systems (CHI)*, Monterey, CA, 1992.

[131] NASA. Mars climate orbiter mishap investigation board phase i report. `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`, 1999.

[132] Alan Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[133] James Newkirk and Robert C. Martin. eXtreme Programming in practice. In *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA-00)*, Minneapolis, MN, 2000.

[134] John T. Nosek. The Case for Collaborative Programming. *Communications of the ACM*, 41(3):105–108, 1998.

[135] Natalya Fridman Noy, Ray W. Fergerson, and Mark A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. In *2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, Juan-les-Pins, France, 2000.

[136] Natalya Fridman Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology. Stanford Medical Informatics, Technical report SMI-2001-0880, 2001.

[137] Object Management Group. Meta Object Facility specification 1.3. `http://www.omg.org/technology/documents/formal/meta.htm`, 2000.

[138] Bernd Oestereich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language, 3. Auflage*. Oldenbourg, Munich, Wien, 1997.

[139] Viktor Okunev. Validation with pure Java: Build a solid foundation for the data-validation framework with the core Java API. JavaWorld, December 2000, `http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-valframe.html`, 2000.

[140] OMG. XML Metadata Interchange (XMI). `http://www.omg.org/cgi-bin/doc?ad/98-10-05`, 1998.

[141] Alessandro Orso, Mary Jean Harrold, and David Rosenblum. Component metadata for software engineering tasks. In *Proc. of the 2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*, Davis, CA, 2000.

[142] Yongzhen Ou. On mapping between UML and entity-relationship model. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pages 45–57, Physica-Verlag, Heidelberg, Germany, 1998.

[143] Francois Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24, 1995.

[144] Helmuth Partsch. *Requirements-Engineering systematisch*. Springer Verlag, Berlin, Germany, 1998.

[145] Joseph Pelrine. Modeling infection scenarios: A fixed-price eXtreme Programming success story. In *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA-00)*, Minneapolis, MN, 2000.

[146] Gary Pollice. Using the Rational Unified Process for small projects: Expanding upon eXtreme Programming. Whitepaper, Rational Software Corporation, 2001.

[147] Wolfgang Pree. Component-based software development – a new paradigm in software engineering? *Software – Concepts and Tools*, 18(4):169–174, 1997.

[148] Wolfgang Pree, Fabio Beckenkamp, and Sergio Viademonte. OO design and implementation of a flexible software architecture for decision support systems. In *Proc. of the Int. Conf. on Software Engineering and Knowledge Engineering (SEKE)*, Madrid, Spain, 1997.

[149] Philip Preston, Paul Compton, and Glenn Edwards. A 1600 rule expert system without knowledge engineers. In J. Leibowitz, editor, *Moving Towards Expert Systems Globally in the 21st Century (Proceedings of the Second World Congress on Expert Systems 1993)*. Cognizant Communication Corporation, 1994.

[150] Colin Price and Kent Spackman. SNOMED clinical terms. *British Journal of Healthcare Computing and Information Management*, 17(3):27–31, 2000.

[151] Franz Josef Radermacher, Thomas Kämpke, Thomas Rose, Klaus Tochtermann, and Tillmann Richter. Management von nicht-explizitem Wissen: Noch mehr von der Natur lernen. Studie erstellt vom Forschungsinstitut für anwendungsorientierte Wissensverarbeitung (FAW Ulm) im Auftrag des Bundesministeriums für Bildung und Forschung (bmb+f), `http://www.faw.uni-ulm.de/deutsch/publikationen/bmbf-studie/index.html`, 2001.

[152] Werner Rammert, Michael Schlese, Gerald Wagner, Josef Wehner, and Rüdiger Weingarten. *Wissensmaschinen: Soziale Konstruktion eines technischen Mediums. Das Beispiel Expertensysteme*. Campus Verlag, Frankfurt, Germany, 1998.

[153] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly, Sebastopol, CA, 1999.

[154] Mark Reinhold. Java Architecture for XML Binding (JAXB). Java Community Process (JSR 31), Working draft `http://java.sun.com/xml/jaxb`, 2001.

[155] Christian Reis. A commentary on the Extreme Programming development process. Async Free Software, `http://www.async.com.br/~kiko/papers/xp`, 2000.

[156] Ralf Reißing. Extremes Programmieren. *Informatik-Spektrum*, 23(2):118–121, 2000.

[157] Richard Rorty. *Der Spiegel der Natur. Eine Kritik der Philosophie.* Suhrkamp Verlag, Frankfurt, Germany, 1987.

[158] Thomas Rose, Martin Fünffinger, Holger Knublauch, and Christian Rupprecht. Prozessorientiertes Wissensmanagement. *Künstliche Intelligenz, Schwerpunkt "Intelligente Systeme für das Wissensmanagement"*, 1:19–24, 2002.

[159] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Eaglewood Cliffs, NJ, 1991.

[160] Christian Rupprecht, Martin Fünffinger, Holger Knublauch, and Thomas Rose. Capture and dissemination of experience about the construction of engineering processes. In *Proc. of the 12th Conference on Advanced Information Systems Engineering (CAISE)*, Stockholm, Sweden, 2000.

[161] Gavriel Salomon. *Distributed Cognitions: Psychological and educational considerations.* Cambridge University Press, Cambridge, UK, 1993.

[162] Bernd Schmidt. *Die Modellierung menschlichen Verhaltens.* SCS-Europe Publishing House, Ghent, Belgium, 2000.

[163] Andy Schneider. Remote access to clinical data. In *Proc. of the Int. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA-00)*, Minneapolis, MN, 2000.

[164] Guus Schreiber, Hans Akkermans, Anjo Anjewierden, Robert de Hoog, Nigel Shadbolt, Walter van de Velde, and Bob Wielinga. *Knowledge Engineering and Management: The CommonKADS Methodology.* MIT-Press, Cambridge, MA, 1999.

[165] Peter Schuh. Code redemption: XP as a means of project recovery. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined.* Addison-Wesley, 2001. `http://www.xp2001.org/xp2001/conference/papers/Chapter7-Schuh.pdf`.

[166] Martin Sedlmayr. Integriertes Monitoring mit prozessadäquater Visualisierung in einem Verteilten System zur Entscheidungsunterstützung in der Anästhesie. Master's thesis, University of Ulm, Germany, 1999. (In German).

[167] Martin Sedlmayr, Holger Knublauch, and Thomas Rose. Agent-based theatre management. *Journal of Clinical Monitoring and Computing*, 16(8):617, 2000.

[168] Ian Sommerville. *Software Engineering*. Addison-Wesley, Reading, MA, fifth edition, 1996.

[169] Manfred Spitzer. *Geist im Netz: Modelle für Lernen, Denken und Handeln*. Spektrum, Akademischer Verlag, Heidelberg, Germany, 1996.

[170] Steffen Staab, Michael Erdmann, Alexander Maedche, and Stefan Decker. An extensible approach for modeling ontologies in RDF(S). In R. Grütter et al., editor, *Knowledge Media in Healthcare: Opportunities and Challenges*. Idea Group Publishing, London, UK, 2001.

[171] Steffen Staab and Alexander Maedche. Axioms are objects, too – ontology engineering beyond the modeling of concepts and relations. In N. Guarino V. R. Benjamins, A. Gómez Pérez, editor, *Proceedings of the ECAI 2000 Workshop on Ontologies and Problem-Solving Methods*, Berlin, Germany, 2000.

[172] Hans Joachim Stoerig. *Kleine Weltgeschichte der Philosophie, Erweiterte Neuausgabe*. Fischer Taschenbuch Verlag, Frankfurt, Germany, 1992.

[173] Rudi Studer, Stefan Decker, Dieter Fensel, and Steffen Staab. Situation and perspective of Knowledge Engineering. In *Knowledge Engineering and Agent Technologies*. IOS Press, 2000.

[174] Rudi Studer, Henrik Eriksson, John Gennari, Samson Tu, and Mark Musen. Ontologies and the configuration of problem-solving methods. In *Proc. of the Knowledge Acquisition Workshop (KAW-96)*, Banff, Canada, 1996.

[175] Rudi Studer, Dieter Fensel, and V. Richard Benjamins. Knowledge Engineering: Principles and methods. *Data & Knowledge Engineering*, 25:161–197, 1998.

[176] Rudi Studer, Dieter Fensel, Stefan Decker, and V. Richard Benjamins. Knowledge Engineering: Survey and future directions. In *Proc. of the 5th German Conf. on Knowledge-based Systems*, Würzburg, Germany, 1999.

[177] Giancarlo Succi and Michele Marchesi, editors. *Extreme Programming Examined*. Addison-Wesley, Reading, MA, 2001.

[178] Sun Microsystems. JavaBeans Specification. `http://java.sun.com/beans`, 1997.

[179] Sun Microsystems. JavaBlend. `http://www.sun.com/software/javablend`, 1999.

[180] Sun Microsystems. Forte for Java. `http://www.sun.com/forte`, 2000.

[181] Sun Microsystems. A simple assertion facility. `http://www.javasoft.com/aboutJava/communityprocess/review/jsr041`, 2000.

[182] Sun Microsystems. Long-term persistence of JavaBeans using XML. Will become part of Java2 Platform, Version 1.4, 2001.

[183] Sun Microsystems. The NetBeans tools platform. White Paper, `http://www.sun.com/forte/whitepapers`, 2001.

[184] TogetherSoft. Together/J. `http://www.togethersoft.com`, 2000.

[185] Hans Ulrich and Gilbert J. B. Probst. *Anleitung zum ganzheitlichen Denken und Handeln. Ein Brevier für Führungskräfte.* Paul Haupt, Bern, Switzerland, 1995.

[186] Mike Uschold. Building ontologies: Towards a unified methodology. In *16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems*, Cambridge, UK, 1996.

[187] Mike Uschold and Michael Gruninger. Ontologies: Principles, method and applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.

[188] Pascal A. T. van Eck, Joeri Engelfriet, Dieter Fensel, Frank van Harmelen, Yde Venema, and Mark Willems. A survey of languages for specifying dynamics: A knowledge engineering perspective. *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2001.

[189] Gertjan van Heijst, August Schreiber, and Bob Wielinga. Using explicit ontologies in KBS development. *Int. Journal of Human-Computer Studies*, 46(2/3):183–292, 1997.

[190] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the Object Constraint Language of UML. Response to Object Management Group's Request for Information on UML 2.0, 1999. `http://sdg.lcs.mit.edu/~dnj`.

[191] A. Inkeri Verkamo. Cooperation of kbs development environments and CASE environments. In *Proc. of the 6th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE-94)*, Jurmala, Latvia, 1994.

[192] Liisa von Hellens and Sue Nielsen. Australien women in IT. *Communications of the ACM*, 44(7):46–52, 2001.

[193] William C. Wake. *Extreme Programming Explored.* Addison-Wesley, Reading, MA, 2001.

[194] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addision-Wesley, Reading, MA, 1998.

[195] Christian Wege and Frank Gerhardt. Learn xp: Host a bootcamp. In *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Italy, 2000.

[196] Don Wells and Trish Buckley. The VCAPS project: An example of transitioning to XP. In *Proc. of the International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Italy, 2000. `http://ciclamino.dibe.unige.it/XP2000`.

[197] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, July/August:19–25, 2000.

[198] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[199] World Wide Web Consortium. Extensible Stylesheet Language (XSL). `http://www.w3.org/Style/XSL`, 1999.

[200] World Wide Web Consortium. XML Schema. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/xmlschema-1`, 2001.

A. The Standard Facet Types of KBeans

This appendix provides details on the catalog of facet types supported by the KBeans specification (as summarized in table 6.1 (page 97), see also [101]). Each of the facet types is defined by the following schema.

facet name

Property types: The property types that the facet can be declared for. The types are represented by placeholders such as `<type>`.

Facet value type: The type of the facet value. This value depends on the property type, and identical placeholders in both property and facet value types represent the same type.

Description: A textual description of the facet's semantics and typical application scenarios.

Validation method: If the facet type is a constraint type: Pseudo-Code of a boolean expression or a mathematical expression (in *italics*) to describe the validation method. Note that for the sake of simplicity, not all cases (such as `null` values) are covered here.

Example: A brief example illustrating the facet. Most of these examples refer to the `Person` class from figure 5.4.

Default Value

`defaultValue`

Property types: `<type>`

Facet value type: `<type>`

Description: Specifies the default value of the property. This value should be equal to the property's initial value, and should not violate any of the property's constraints. Default values can be used to repair invalid values, and to implement "reset" functionality.

Example:

```
// The valid values of the maritalStatus property
public final static int UNMARRIED = 0;
public final static int MARRIED = 1;
public final static int WIDOWED = 2;

/**
 * The default maritalStatus of a Person is UNMARRIED.
 */
public final static int maritalStatusDefaultValue = UNMARRIED;
```

Length and Cardinality

`maxLength`, `minLength`

Property types: `<type>`

Facet value type: `int (0 <= minLength <= maxLength)`

Description: Specifies the maximum/minimum length of the result of the `toString` method of property values (`null` values are treated as empty strings). Most frequently, this will be used to constrain the length of `String` properties, or the number of digits of a number. A `maxLength` of `Integer.MAX_VALUE` is treated as infinite.

Validation method:

```
minLength <= ("" + propertyValue).length() <= maxLength
```

Example:

```
/**
 * The maximum length of a Person's last name is 17 chars.
 */
public final static int lastNameMaxLength = 17;
```

`fractionDigits`

Property types: `float, double, float[], double[]`

Facet value type: `int (fractionDigits >= 0)`

Description: Specifies the maximum number of digits of the fractional part of a floating point number.

Validation method:

```
double fraction = (propertyValue - (int)propertyValue);
("" + fraction).length() - 2 <= facetValue
```

Example:

```
/**
 * A price can have at most two digits behind the dot.
 */
public final static int priceFractionDigits = 2;
```

`maxCardinality, minCardinality`

Property types: `<simple>[]`

Facet value type: `int (0 <= minCardinality <= maxCardinality)`

Description: Specifies the maximum/minimum cardinality of indexed property values (the cardinality of `null` arrays is 0). A `maxCardinality` of `Integer.MAX_VALUE` is treated as infinite.

Validation method:

```
minCardinality <= propertyValue.length <= maxCardinality
```

Example:

```
/**
 * A Person that is UNMARRIED cannot have any children.
 */
public int getChildrenMaxCardinality() {
    if(getMaritalStatus() == UNMARRIED) {
        return 0;
```

```
        }
        else {
            return Integer.MAX_VALUE;  // unlimited
        }
    }
```

Type Restriction

`validClasses`

Property types: `<object>`, `<object>[]`

Facet value type: `Class[]`

Description: Constrains the type of valid property values. The facet values must be subclasses of the property type. This constraint is satisfied, if all values of the property are either instances of one of the types declared as facet values, or `null`.

Validation method:

```
forall v=propertyValues: facetValues.contains(v.getClass())
```

Example: (defined in `MalePerson` class)

```
/**
 * The spouse of a MalePerson must be a FemalePerson.
 */
public final static Class[] spouseValidClasses = {
    FemalePerson.class
};
```

`invalidClasses`

Property types: `<object>`, `<object>[]`

Facet value type: `Class[]`

Description: Constrains the type of valid property values. The facet values must be subclasses of the property type. This constraint is satisfied, if none of the values of the property is an instance of one of the types declared as facet values.

Validation method:

```
!(forall v=propertyValues: facetValues.contains(v.getClass()))
```

Example: (defined in `MalePerson` class)

```
/**
 * The spouse of a MalePerson cannot be a MalePerson.
 */
public final static Class[] spouseInvalidClasses = {
    MalePerson.class
};
```

Range by Comparison

`maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`

Property types: `<comparable>`, `<comparable>[]`

Facet value type: `<comparable>`

Description: Specifies the upper/lower bounds of the property value, with regard to `Comparable.compareTo`.

Validation method:

```
maxExclusive: propertyValue < facetValue
maxInclusive: propertyValue <= facetValue
minExclusive: propertyValue > facetValue
minInclusive: propertyValue >= facetValue
```

Example:

```
/**
 * A Person's (inclusive) minimum age is 0 (age >= 0).
 */
public final static int ageMinInclusive = 0;
```

Range by Enumeration

`validValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies all valid values of the property. This constraint is satisfied, if all property values are either equal to one of the facet values (using the `equals` method), or `null` if `null` is one of the facet values.

Validation method: $propertyValues \subseteq facetValues$

Example:

```
/**
 * The maritalStatus must be UNMARRIED, MARRIED, or WIDOWED
 */
public final static int[] maritalStatusValidValues = {
    UNMARRIED,
    MARRIED,
    WIDOWED
};
```

`requiredValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies a set of values that must be part of the set of property values. This constraint is satisfied, if all facet values are equal to one of the property values (using the `equals` method).

Validation method: $propertyValues \supseteq facetValues$

Example:

```
/**
 * The parents of the spouse must be among the guests.
 */
public Person[] getGuestsRequiredValues() {
    return (spouse == null)  ?  null  :  spouse.getParents();
}
```

`invalidValues`

Property types: `<simple>`, `<simple>[]`

Facet value type: `<simple>[]`

Description: Specifies invalid values of the property. None of the property values may be equal to one of the facet values (using the `equals` method), or `null` if `null` is one of the facet values.

Validation method: $propertyValues \cap facetValues = \emptyset$

Example:

```
/**
 * A Person cannot be married to one of his/her children.
 */
public Person[] getSpouseInvalidValues() {
    return children;
}
```

`notNull`

Property types: `<type>`

Facet value type: `boolean`

Description: Declares whether `null` is a valid property value.

Validation method:

```
facetValue == false || propertyValue != null
```

Example:

```
/**
 * The lastName must not be null.
 */
public final static boolean lastNameNotNull = true;
```

Relationship between Properties

Whereas the above constraint types restrict property values by describing valid and invalid values directly, the following constraint types achieve this indirectly by declaring a relationship between the values of the property and those from another property. Since it is very complicating in Java to refer to other properties by objects (this would mean to return `PropertyDescriptors` as facet values), these constraint types declare properties by their names, i.e. as `Strings`. Beside simple references to other properties, they also use *property chains* (cf. the *slot chains* from OKBC). A property chain defines a list of property names `P[0]`, ..., `P[n]`, so that the values of the chain are the values of the `P[n]` property of the values of the `P[n-1]` property... of the values of the `P[0]` property, where `P[0]` is a property of the given class. For example, the values of the property chain `["spouse", "parents"]` are the parents of a person's spouse.

`inverseProperty`

Property types: `<object>`, `<object>[]`

Facet value type: `String` (name of a property of the class `<object>`)

Description: Declares the inverse property in the class that is referenced to by the property type. The property is specified by name. Satisfaction of this constraint means that if `ref` is a property value of the object `obj`, then `obj` is also a value of the inverse property of `ref`.

Validation method:

```
forall v=propertyValues: v.inverseProperty.contains(obj)
```

Example:

```
/**
 * The inverse property of children is parents.
 */
public final static String childrenInverseProperty="parents";
```

`equalProperty`

Property types: `<type>`

Facet value type: `String[]` (a property chain)

Description: Declares that the property values must be equal to the values of another property that is specified by a property chain. This constraint is satisfied if both sets of values are equal, whereby the single elements are compared with `equals`.

Validation method: $propertyValues = chainValueSet(facetValues)$

Example:

```
/**
 * A person's children are equal to the spouse's children.
 */
public final static String[] childrenEqualProperty = {
    "spouse", "children"
};
```

`unequalProperty`

Property types: `<type>`

Facet value type: `String[]` (a property chain)

Description: Declares that the sets of values of the given property and those of a specified property chain must not be completely equal.

Validation method: $propertyValues \neq chainValueSet(facetValues)$

Example:

```
/**
 * A person's parents must be different from the
 * spouse's parents.
 */
public final static String[] parentsUnequalProperty = {
    "spouse", "parents"
};
```

`validValuesProperty`, `requiredValuesProperty`, `invalidValuesProperty`

Property types: `<type>` (or `<type>[]` for `requiredValuesProperty`)

Facet value type: `String[]` (a property chain)

Description: Declares the valid/invalid/required property values by referring to the values of a specified property chain. The semantics are comparable to the related facet types `validValues`, `requiredValues`, and `invalidValues`.

Validation methods:
`valid...`: $propertyValues \subseteq chainValueSet(facetValues)$
`required...`: $propertyValues \supseteq chainValueSet(facetValues)$
`invalid...`: $propertyValues \cap chainValueSet(facetValues) = \emptyset$

Example:

```
/**
 * A person cannot be married to one of his/her children.
 */
public final static String[] spouseInvalidValuesProperty = {
    "children"
};
```

Lexical Pattern Matching

`pattern`

Property types: `<type>`

Facet value type: `String`

Description: Declares a regular expression in the notation of Perl which is used by both the `java.util.regex` package of Java 1.4 and the XML Schema facet `pattern` [200]. The constraint is satisfied, if the `toString` representation of each of the property values matches this expression (whereby `null.toString()` is `""`).

Validation method:

```
forall v=propertyValue: Pattern.matches(facetValue, "" + v);
```

Example:

```
/**
 * The zipCode must match the pattern "[0-9]{5}(-[0-9]{4})?".
 */
public final static String zipCodePattern =
                              "[0-9]{5}(-[0-9]{4})?";
```

Collection Type

`ordered`

Property types: `<comparable>[]`

Facet value type: `boolean`

Description: Declares whether the property values must be in ascending order (using the `Comparable.compareTo` method).

Validation method:

```
!facetValue ||
forall i: propertyValue[i].compareTo(propertyValue[i+1] <= 0
```

Example:

```
/**
 * Children must be ordered by age (using Person.compareTo).
 */
public final static boolean childrenOrdered = true;
```

`duplicateFree`

Property types: `<simple>[]`

Facet value type: `boolean`

Description: Declares whether the property values may contain duplicates (using the `equals` method).

Validation method:

```
!facetValue ||
(forall i!=j: !propertyValue[i].equals(propertyValue[j])
```

Example:

```
/**
 * The list of firstNames must be duplicate-free.
 */
public final static boolean firstNamesDuplicateFree = true;
```

Textual Representation

`validValuesNames`

Property types: `<type>`

Facet value type: `String[]`

Description: Defines a textual representation of all the property value(s) declared with the `validValues` constraint. Each of the results of `validValues` must have one corresponding name element. For example, this facet can be used to define human-readable values of primitive constants for GUI elements (e.g., radio buttons and combo boxes). It can serve as a replacement for the missing support of enumeration types in Java.

Example:

```
public final static String[] maritalStatusValidValuesNames={
    "unmarried", // [0] = UNMARRIED
    "married",   // [1] = MARRIED
    "widowed"    // [2] = WIDOWED
};
```

B. A Sample KBeans Ontology Class

The following Java source code implements the `Person` class from the sample ontology from figure 5.4 (page 66).

```
package demo.person;

import de.faw.kbeans.java.*;
import java.util.*;

/**
 * A simple JavaBeans class to test various KBeans constraints.
 *
 * @author Holger Knublauch
 * @author Martin Sedlmayr
 */
public abstract class Person extends AbstractKBean
    implements Comparable, NamedBean {

    // age property ------------------------------------------------------

    /**
     * The value of the age property
     */
    private int age;

    /**
     * A Person's minimum age is 0
     */
    public final static int ageMinInclusive = 0;


    /**
     * Gets the Person's age.
     * @return the age
     */
```

```
public int getAge() {
    return age;
}


/**
 * Set the value of the age.
 * @param value the value to assign
 * @see #ageMinInclusive
 */
public void setAge(int value) {
    int old = age;
    age = value;
    firePropertyChange("age", old, age);
}


// children property ------------------------------------------------

/**
 * The value of the children property
 */
private Person[] children;

/**
 * The children must be duplicate-free
 */
public final static boolean childrenDuplicateFree = true;

/**
 * A Person's children must be ordered (by age)
 */
public final static boolean childrenOrdered = true;


/**
 * Gets the children.
 * @return the children or null
 */
public Person[] getChildren() {
    return children;
}
```

```
/**
 * A Person that is UNMARRIED cannot have any children.
 */
public int getChildrenMaxCardinality() {
    if(getMaritalStatus() == UNMARRIED) {
        return 0;
    }
    else {
        return Integer.MAX_VALUE;  // unlimited
    }
}


/**
 * Sets the children.
 * @param values the new children or null
 * @see #childrenDuplicateFree
 * @see #childrenOrdered
 * @see #getChildrenMaxCardinality
 */
public void setChildren(Person[] values) {
    Object old = children;
    children = values;
    firePropertyChange("children", old, values);
}


// firstNames property ---------------------------------------------

/**
 * The value of the firstNames property
 */
private String[] firstNames;

/**
 * The firstNames must be distinct
 */
public final static boolean firstNamesDuplicateFree = true;


/**
 * Gets the first names.
```

```
     * @return the first names or null
     */
    public String[] getFirstNames() {
        return firstNames;
    }


    /**
     * Sets the firstNames.
     * @param values the new firstNames or null
     * @see #firstNamesDuplicateFree
     */
    public void setFirstNames(String[] values) {
        Object old = firstNames;
        firstNames = values;
        firePropertyChange("firstNames", old, values);
    }


    // lastName property ------------------------------------------------

    /**
     * The current value of the lastName property
     */
    private String lastName;


    /**
     * The maximum length of a Person's last name is 17 chars.
     */
    public final static int lastNameMaxLength = 17;

    /**
     * The lastName must not be null.
     */
    public final static boolean lastNameNotNull = true;


    /**
     * Gets the value of the lastName property.
     * @return the lastName
     * @see #setLastName
     */
```

```
public String getLastName() {
    return lastName;
}


/**
 * Sets the value of the lastName property.
 * @param value the new lastName
 * @see #lastNameMaxLength
 * @see #lastNameNotNull
 */
public void setLastName(String value) {
    Object old = lastName;
    lastName = value;
    firePropertyChange("lastName", old, value);
}


// maritalStatus property ---------------------------------------------

// The valid values of the maritalStatus property
public final static int UNMARRIED = 0;
public final static int MARRIED = 1;
public final static int WIDOWED = 2;

/**
 * The current value of the maritalStatus property
 */
private int maritalStatus = UNMARRIED;

/**
 * The default maritalStatus of a Person is UNMARRIED.
 */
public final static int maritalStatusDefaultValue = UNMARRIED;

/**
 * The maritalStatus must be UNMARRIED, MARRIED, or WIDOWED
 */
public final static int[] maritalStatusValidValues = {
    UNMARRIED,
    MARRIED,
    WIDOWED
};
```

```
public final static String[] maritalStatusValidValuesNames = {
    "unmarried", // [0] = UNMARRIED
    "married",   // [1] = MARRIED
    "widowed"    // [2] = WIDOWED
};


/**
 * Gets the marital status.
 * @return the marital status
 */
public int getMaritalStatus() {
    return maritalStatus;
}


/**
 * Sets the maritalStatus.
 * @param value the new maritalStatus
 * @see #maritalStatusValidValues
 * @see #maritalStatusValidValuesNames
 */
public void setMaritalStatus(int value) {
    int old = maritalStatus;
    maritalStatus = value;
    firePropertyChange("maritalStatus", old, value);
}



// spouse property ------------------------------------------------

/**
 * The value of the spouse property
 */
private Person spouse;


/**
 * Gets the spouse.
 * @return the spouse
```

```
 */
public Person getSpouse() {
    return spouse;
}


/**
 * A Person's spouse may not be one of the children.
 */
public Person[] getSpouseInvalidValues() {
    return children;
}


/**
 * The spouse must not be null, if the maritalStatus is MARRIED.
 */
public boolean isSpouseNotNull() {
    return getMaritalStatus() == MARRIED;
}


/**
 * Sets the spouse.
 * @param value the new spouse
 * @see #getSpouseInvalidValues
 * @see #isSpouseNotNull
 */
public void setSpouse(Person value) {
    Object old = spouse;
    spouse = value;
    firePropertyChange("spouse", old, value);
}


// constructors & methods -----------------------------------------

/**
 * Constructs a new Person.
 */
public Person() {
}
```

```
    /**
     * Defines an ordering among Persons, with respect to age.
     * @param other the object to compare to (usually a Person)
     * @return the other's age compared to this age
     */
    public int compareTo(Object other) {
        if(other instanceof Person) {
            int oa = ((Person)other).age;
            if(age < oa) {
                return -1;
            }
            else
            if(age > oa) {
                return 1;
            }
        }
        return 0;
    }


    /**
     * Gets an object's name that appears in visual representations (e.g.
     * a KBeansShell). Implements <CODE>NamedBean</CODE>.
     * @return the lastName or null
     */
    public String getBeanName() {
        return lastName;
    }


    /**
     * Used by the KBeans introspection mechanism to locate the subclasses
     * of this class (e.g. in a KBeansShell). Required since there is no
     * Java standard way to forward-reference subclasses.
     */
    static {
        KBeansIntrospector.installClass(MalePerson.class);
        KBeansIntrospector.installClass(FemalePerson.class);
    }
}
```

C. A Sample KBeans Constraint Class

The following Java source code defines the KBeans constraint type “maximum cardinality”. The constraint allows to specify the maximum size of indexed (array) properties.

```
package de.faw.kbeans.facets.core;

import de.faw.kbeans.*;
import de.faw.kbeans.facets.*;

/**
 * A RepairableConstraintType for indexed properties, which specifies
 * the maximum length of the array.
 * @author  Holger Knublauch
 */
public class MaxCardinalityConstraintType extends AbstractFacetType
    implements RepairableConstraintType {

    public MaxCardinalityConstraintType() {
        super("MaxCardinality");
    }

    // Implements ConstraintType
    public String toString(BeanModel beanModel, Object facet) {
        return "Maximum cardinality of " +
               BeanModelTools.getBeanDescription(beanModel, facet) +
               " expected";
    }

    /**
     * The facet type is int.
     * @return <CODE>int.class</CODE>
     */
    public Object getFacetType(BeanModel beanModel, Object valueType) {
        return int.class;
    }
```

```
    /**
     * Removes the end of the array, in order to match the maximum length.
     */
    public Object getValidValue(BeanModel beanModel,
                                Object bean, Object property,
                                Object facet, Object value) {
        if(value == null) {
            return null;
        }
        else {
            int max = ((Number)facet).intValue();
            Object cls = beanModel.getClass(bean);
            Object ct = beanModel.getPropertyType(cls, property);
            Object neu = beanModel.createArray(ct, max);
            beanModel.arraycopy(value, 0, neu, 0, max);
            return neu;
        }
    }

    /**
     * This Constraint is applicable to indexed types only.
     */
    public boolean isAdmissibleValueType(BeanModel beanModel, Object type) {
        return beanModel.isIndexedType(type);
    }

    /**
     * A value is invalid, if it is null, or its length is <= the maximum length.
     */
    public boolean isValidValue(BeanModel beanModel,
                                Object bean, Object property,
                                Object facet, Object value) {
        if(facet instanceof Number) {
            int max = ((Number)facet).intValue();
            int len = (value == null) ? 0 : beanModel.getArrayLength(value);
            return len <= max;
        }
        return true;
    }
}
```

D. The Facets of XML Schema

XML Schema [200] is an XML-based specification language for describing the syntax and semantics of a class of XML documents. XML Schema documents can be used to define simple elements (comparable to the primitive types of Java such as `int` and `boolean`) and complex elements (which can be built by combining other elements, comparable to classes in Java). New simple element types are defined by deriving them from existing simple types, e.g. by building lists of other types. In particular, it is possible to derive a new simple type by restricting an existing simple type. The legal range of values for the new type is a subset of the existing type's range of values. Restrictions (or constraints) are defined by means of so-called *facets*. Every facet has a name that is associated to predefined semantics, and a value of a type that can be uniquely derived from the simple element type that is constrained by the facet. For example, the facet named `minExclusive` defines the exclusive lower bound (using $>$) of a numerical element. The value of the `minExclusive` facet must be of the same type as the simple element, e.g. `int` for `int` elements. In order to cover the most frequently needed types of metadata, XML Schema provides a set of standard facet types, which is summarized in table D.1.

Table D.1.: The constraining facets of XML Schema.

Facet name	**Value type**	**Facet type**	**Semantics**
`length`	`string`, list	`int` ($\geq$0)	(Exact) string or list length.
`minLength`*	`string`, list	`int` ($\geq$0)	Minimum string or list length.
`maxLength`*	`string`, list	`int` ($\geq$0)	Maximum string or list length.
`pattern`	`<type>`	reg. expr.	Lexical space of the values.
`enumeration`	`<type>`	`<type>[]`	Enumeration of valid values.
`whitespace`	`<type>`	(3 options)	Preserve, replace, or collapse whitespaces.
`maxInclusive`	`<ordered>`	`<type>`	Inclusive upper bound ($\leq$).
`maxExclusive`	`<ordered>`	`<type>`	Exclusive upper bound ($<$).
`minInclusive`	`<ordered>`	`<type>`	Inclusive lower bound ($\geq$).
`minExclusive`	`<ordered>`	`<type>`	Exclusive lower bound ($>$).
`totalDigits`	`<decimal>`	`int` ($>$0)	Maximum number of digits.
`fractionDigits`	`<decimal>`	`int` ($\geq$0)	Maximum length of the fractional part.

*Similar to `minLength` and `maxLength`, which constrain simple types, XML Schema supports `minOccurs` and `maxOccurs` declarations for complex types.

E. The BeanModel Java interface

This appendix lists the signatures of the core methods of the Java interface `BeanModel`. The methods are grouped by their functionality.

```
public interface BeanModel {

    // Creation and analysis of beans (objects)
    Object createBean(Object cls);
    String getBeanName(Object bean);
    Object getClass(Object bean);

    // Analysis of classes
    Object getBeanClassType();
    Object getClassByName(String clsName);
    String getClassName(Object cls);
    Class getCustomizerClass(Object cls);
    Object getComponentType(Object type);
    Object[] getKnownSubClasses(Object cls);
    Object[] getPrimitiveProperties(Object cls);
    Object[] getProperties(Object cls);
    Object getPropertyByName(Object cls, String propertyName);
    PropertyEditor getPropertyEditor(Object cls);
    Object[] getRefProperties(Object cls);
    Object getSuperClass(Object cls);
    boolean isAbstractClass(Object cls);
    boolean isComparableType(Object cls);
    boolean isIndexedType(Object cls);
    boolean isRefType(Object cls);
    boolean isSubClass(Object cls, Object subClass);

    // Analysis and manipulation of properties
    String getPropertyName(Object cls, Object property);
    Object getPropertyType(Object cls, Object property);
    Object getPropertyValue(Object bean, Object property);
    Object getPropertyValue(Object bean, Object property, int index);
```

```
    int getPropertyValueCount(Object bean, Object property);
    void insertPropertyValue(Object bean, Object property,
                             int index, Object value);
    boolean isIndexedProperty(Object cls, Object property);
    void removePropertyValue(Object bean, Object property, int index);
    void setPropertyValue(Object bean, Object property, Object value);
    void setPropertyValue(Object bean, Object property,
                          int index, Object value);

    // PropertyChangeListener management
    void addPropertyChangeListener(Object bean, PropertyChangeListener l);
    void addPropertyChangeListener(Object bean, Object property,
                                   PropertyChangeListener l);
    void removePropertyChangeListener(Object bean,
                                      PropertyChangeListener l);
    void removePropertyChangeListener(Object bean, Object property,
                                      PropertyChangeListener l);

    // Constraint checking
    ConstraintViolation createConstraintViolation(Object bean,
             Object property, Object context, Object value, Constraint c);
    Object getConstraintContext(Object bean, Object property, Constraint c);
    Constraint getPropertyConstraint(Object cls, Object property, Class cc);
    Iterator getPropertyConstraints(Object cls, Object property);

    // Object graph management
    Object getRootBean();
    Object getRootBeanProperty();
    BeanModelSet getBeanModelSet();
    void setRootBeanPropertyType(Object cls);
}
```

F. Class Diagram of the Meta Object Facility (MOF)

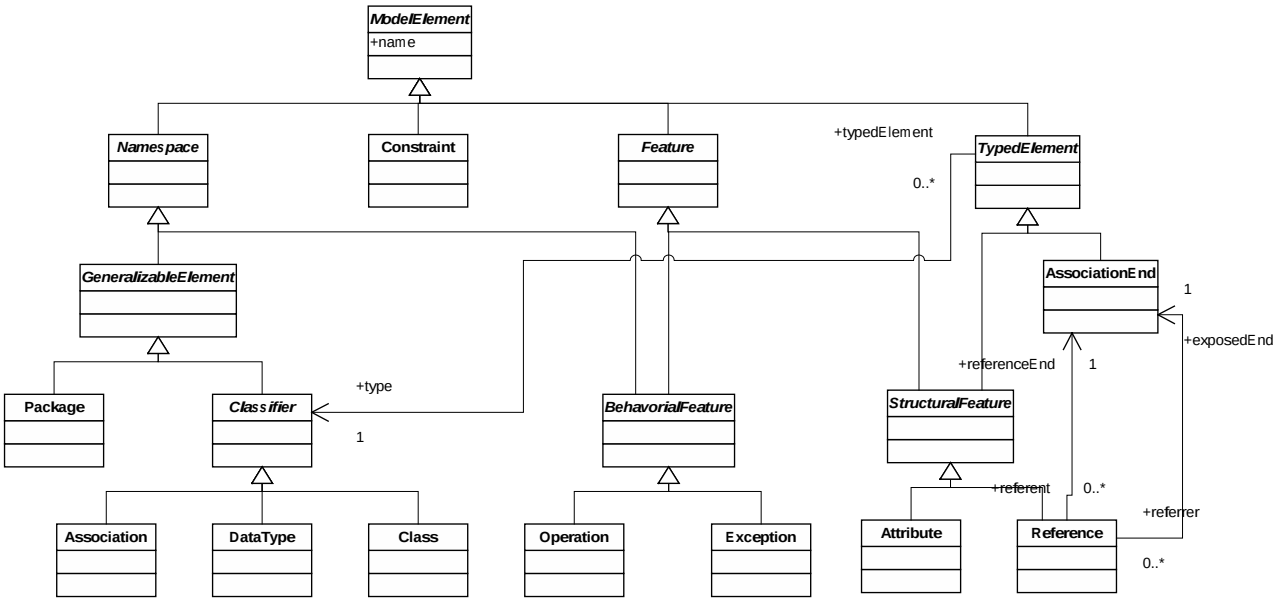

Figure F.1.: Some of the core classes of the Meta Object Facility.

G. Java Libraries for Knowledge-Based Systems and Artificial Intelligence

This appendix provides a catalogue of Java libraries that support functionality which is typically assigned to knowledge-based or Artificial Intelligence systems. Please note that many of these projects are research prototypes, so that the given links might be outdated.

CommonRules

`http://www.research.ibm.com/rules/home.html`
Rule-based business processes for e-commerce.

ECJ

`http://www.cs.umd.edu/projects/plus/ec/ecj`
A Java-based Evolutionary Computation and Genetic Programming Research System.

FIPA-OS

`http://fipa-os.sourceforge.net`
An open-source multi-agent platform in Java.

Java Agent Services

`http://www.java-agent.org`
The (forthcoming) Java standard extension on multi-agent systems.

Java Rule Engine

`http://jcp.org/jsr/detail/94.jsp`
A Java Community Process that defines a Java runtime API for rule engines.

JCL (Java Constraint Library)

`http://liawww.epfl.ch/~torrens/Project/JCL`
Application services related with binary Constraint-Satisfaction Problems.

JESS (Java Expert System Shell)

`http://herzberg.ca.sandia.gov/jess`
A rule engine and scripting environment, based on the RETE algorithm.

SPIRIT

`http://www.xspirit.de`
A Java tool and API providing probabilistic reasoning functionality.

WEKA

`http://www.cs.waikato.ac.nz/~ml/weka`
Machine learning software in Java, particularly data mining algorithms.

H. The KBeans Logic package

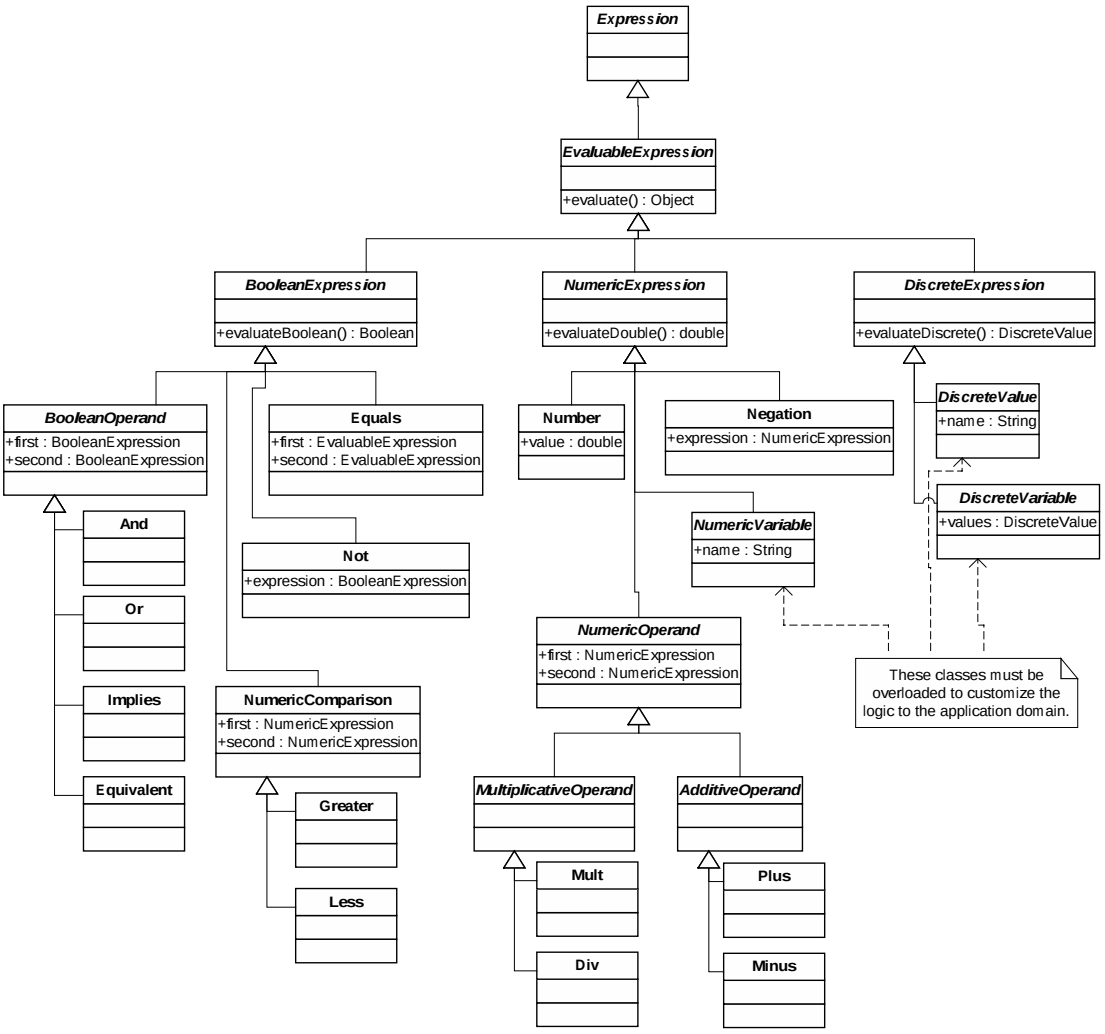

Figure H.1.: A generic library of KBeans classes for the representation and evaluation of logical rules.

Curriculum Vitae

Holger Knublauch was born on July 7th, 1971 in Berlin-Lichterfelde, Germany. After having passed the Abitur (high school graduation) in Hagen, Germany, in 1991, and the mandatory military service, he started studying Computer Science with Economics as minor at the University of Dortmund, Germany, in 1992. Following the completion of his Vordiplom (intermediate diploma) in Dortmund, he moved to the University of Ulm, Germany, where he graduated on a masters thesis on object-oriented software development of a probabilistic knowledge-based system in March, 1997. In 1997, he gathered working experience as software developer in a New Zealand-based teachware company. In 1998, he became a member of the Department of Business Processes and Telematics at the Research Institute for Applied Knowledge Processing (FAW), Ulm, Germany. He worked in various research and development projects with a focus on knowledge and process modeling. The main ideas of this PhD thesis have emerged from the projects ANIS (Anesthesia Information System) and AGIL (Agent-based Information Logistics), where Holger was the principal researcher. In addition to his work at the FAW, in 2001 he started working as a lecturer for application development at the University of Applied Sciences (Fachhochschule) Neu-Ulm, Germany, and as a lecturer and project coach in Extreme Programming courses at the University of Ulm, Germany.

Zusammenfassung

Das Ziel dieser Arbeit ist es die Entwicklung wissensbasierter Systeme effizienter zu machen. Zu diesem Zweck schlägt die Arbeit eine neue, agile Software- und Knowledge Engineering-Methode vor, die mit *XP.K (eXtreme Programming of Knowledge-based systems)* bezeichnet wird. Diese Methode basiert auf den vier Werten *Einfachheit*, *Gemeinschaft*, *Feedback* und *Eigenverantwortung*, und wendet objektorientiertes Round-Trip Engineering auf die Wissensmodellierung an.

Grundlage der Arbeit ist die Beobachtung, dass Wissen für die meisten wissensbasierten Systeme notwendigerweise in einer evolutionären Weise modelliert werden muss, wobei Fachexperten und Ingenieure eng zusammenarbeiten müssen. Der Autor zeigt, dass die existierenden ,,schwergewichtigen" Entwicklungsmethoden der objektorientierten Softwaretechnik und aus dem Knowledge Engineering oft ineffizient sind, weil sie hohe Kosten für Änderungen in den Wissensmodellen nach sich ziehen. Außerdem stellen sie wenig Unterstützung für die Übergänge zwischen Wissen, Wissensmodellen und dem restlichen ausführbaren System bereit. Der Ausgangspunkt von XP.K ist die Hypothese, dass ,,leichtgewichtige" – oder *agile* – Entwicklungsprozesse (wie z.B. Extreme Programming) für die Wissensmodellierung geeignet sind, weil sie für Projekte mit häufig sich ändernden Anforderungen optimiert sind, und zudem mit einem Minimum an Modellierungsartefakten auskommen zwischen denen recht reibungslose Übergänge möglich sind.

XP.K wendet die wesentlichen Prinzipien von Extreme Programming auf das Feld der Wissensmodellierung an. Der Entwicklungsprozess legt großen Wert auf Kommunikation. Fachexperten und Wissensingenieure arbeiten eng bei der Definition von Metamodellen (Ontologien) zusammen, und Wissen wird jeweils zu zweit akquiriert und getestet. Ontologien werden in derjenigen (objektorientierten) Sprache implementiert, die auch für die restlichen Systemmodule verwendet wird. Diese Ontologien machen ihre Struktur sowohl zur Laufzeit als auch zur Entwurfszeit auf transparente Art und Weise nach außen hin sichtbar. Zur Laufzeit wird Reflection verwendet um die Ontologie zu analysieren, sodass generische Wissensmodellierungswerkzeuge und Inferenzmaschinen (wieder-) verwendet werden können. Zur Entwurfszeit wird Round-Trip Engineering eingesetzt um die Struktur der Ontologie nach UML umzuwandeln, sodass die gesamten Informationen und Dokumentationen zur Ontologie zentral im Quellcode abgelegt werden können. Ausführbare Prototypen werden schnell erzeugt und in kurzen Zyklen mit Hilfe von automatischen Tests und Constraint-Checks evaluiert.

XP.K wird durch einen umfangreichen, direkt einsetzbaren Satz an Basissoftware unterstützt. Dieser beinhaltet eine Erweiterung des Java-Komponentenmodelles JavaBeans zur Darstellung von semantischen Constraints über Objektzustände, und eine generische Plattform für Wissensmodellierungswerkzeuge. Die Effizienz von XP.K wird durch Fallstudien untermauert, z.B. anhand eines klinischen entscheidungsunterstützenden Systems und eines Prozessmodellierungsansatzes für Multi-Agenten-Systeme.